Multiplicative Coding and Factorization in Vector Symbolic Models of Cognition

Spencer Kent

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-215
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-215.html

December 18, 2020

Multiplicative coding and factorization in vector symbolic models of cognition

by

Spencer Kent

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Bruno A. Olshausen, Co-chair
Professor Alexei A. Efros, Co-chair
Assistant Professor Steven T. Piantadosi

Fall 2020

Multiplicative coding and factorization in vector symbolic models of cognition

Abstract

Multiplicative coding and factorization in vector symbolic models of cognition

by

Spencer Kent

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Bruno A. Olshausen, Co-chair

Professor Alexei A. Efros, Co-chair

This dissertation covers my attempts to confront the challenge and promise of multiplicative representations, and their attendant factorization problems, in the brain. This is grounded in a paradigm for modeling cognition that defines an algebra over high-dimensional vectors and presents a compelling factorization problem. The proposed solution to this problem, a recurrent neural network architecture called Resonator Networks, has several interesting properties that make it uniquely effective on this problem and may provide some principles for designing a new class of neural network models. I show some applications of multiplicative distributed codes for representing visual scenes and suggest how such representations may be a useful tool for unifying symbolic and connectionist theories of intelligence.

To Edie.

Contents

List of Figures

List of Tables

Acknowledgments

The ideas in this thesis have been shaped most of all by my advisor Bruno Olshausen, whose sustained patience and support has made this work possible. Almost without exception over the course of 5 years, I left meetings with Bruno more excited about my own research than I had been the day before. This is a standard of mentorship that I hope to live up to in my own professional life. Among Bruno's many talents, he is a great communicator of science, and I have been consistently inspired to improve my own writing and speaking by the example he sets for his students.

The Redwood Center for Theoretical Neuroscience is a strange and wonderful collection of misfits who I've been lucky to share ideas and Friday morning pastries with over the last 5 years. It's free-wheeling style is not for everyone. Sometimes I wondered whether it was even for me. In the end, I come back around to the reason I joined in early 2016: the intellectual freedom offered to researchers, from top to bottom, is extraordinary. This, coupled with a certain rejection of conventional wisdom and a scrappy ethos, has made the Redwood Center a wonderful place to spend my PhD. Among my many colleagues there I want to especially thank Pentti Kanerva, whose work on Sparse Distributed Memory and then Vector Symbolic Architectures set me on a fun and unusual path when I read about it early on in my time here. The way Pentti approaches science, and people, makes me want to keep doing research. Thanks also to my officemates Michael Fang and Connor Bybee who shared countless hours–mostly of quiet focus but a few of needed distraction–in Evans 563. Yubei Chen, Vasha DuTell, and Mayur Mudigonda provided intellectual and emotional stimulation in ways that made things more meaningful. My longtime collaboration with Paxon Frady taught me a great deal about the culture of academia, and about myself. Thanks also to Navneedh Maudgalya, with whom I started a short but productive collaboration in the final year of my program.

Lifelong friendships built outside of my working life are what I cherish most about the last 5 years. Anusha, Matt, Brian, Chris, and Gideon are among those who have made it all so interesting. Thanks to them for indulging my eccentricities and sharing their many talents.

Lastly, thanks are due to my family. With the aunts, uncles, and cousins of my Albuquerque tribe I've been set up with a springboard for life that not every young person is lucky enough to get. My brother Carson is my biggest inspiration, closest confidant, and strongest ally. Thanks most of all to Mom and Dad, whose unconditional love has always supplied me with the strength to keep going.

Chapter 1

Introduction

Intelligence research today is faced with the eighty-year-old challenge of explaining how rich and flexible cognitive data structures can arise from the wetware of the brain. Theories that build bottom-up from highly simplified models of neural networks enjoy widespread popularity and some impressive capabilities. Theories that build top down have more to say about what actually makes a system intelligent. Where they will ultimately meet, no one knows. This dissertation is about a computation which I believe to be fundamental for both perspectives and perhaps a key to moving between the two.

A brief note on the notation to be used: bolded lower-case letters ($\mathbf{x}$) are vectors, bolded upper-case letters ($\mathbf{X}$) are matrices, and words or letters in `typewriter` font are *cognitive concepts*. A cognitive concept is a concept for which I am making no particular choice of representation. `Denmark`, `bicycle`, `noun`, `red`, and `42` are all perfectly valid cognitive concepts. What this thesis (and indeed much of intelligence research) is concerned with is how to best *represent* these concepts. Throughout this work, concepts are represented with high-dimensional vectors. What is meant by high dimensional is ≥ 1000. I write $\texttt{Denmark} \sim \mathbf{d}$ to indicate that the cognitive concept `Denmark` gets represented by vector $\mathbf{d}$, a mapping referred to as *encoding*. Inferring the concept associated to $\mathbf{d}$ is then *decoding*.

Reasoning requires methods for combining cognitive concepts. One of the the most important is *conjunction*. Conjunction allows us to represent a `red bicycle`, using the more fundamental concepts `red` and `bicycle`. We will notate the conjunction of two concepts `A` and `B` as $\texttt{A} \wedge \texttt{B}$ or $(\texttt{A}, \texttt{B})$. It is *critical* that we are able to form conjunctions–this is the gateway to building more complex concepts out of simple parts. In our space of vector representations a conjunction can be produced via an operation that "multiplies" vectors in a particular way. Multiplication in this context shares essentially all of the properties of multiplication applied to real numbers. In the case that the cognitive concepts obey some multiplicative group structure, this may be part of a literal isomorphism between the space of cognitive concepts and the space of their vector representations.

What is meant by "multiplicative codes"–in the title of and throughout this dissertation–is simply vector representations formed by a multiplicative combination of other vectors. We might have called these "conjunctive codes" were it not for the fact that this has historically

had a rather specific meaning as *tensor* representations formed via a generic outer product. Vectors produced by a multiplicative operation are *composite* while the vectors that were combined multiplicatively are its *factors*. When a representation has been generated via a multiplicative operation, we can call the inverse of this–the task of decomposing a composite into its constituent atoms–factorization. I will argue that the brain needs to generate and factor multiplicative codes.

1.1 Levels of intelligence

For as long as researchers have been studying intelligence, they have been confronted by the enormous variety of ways in which a system can be said to behave intelligently. How is it that sensory processing, which seems "analog" and largely statistical, can be reconciled with the decidely structured and rule-based properties of languague and other forms of cognition? The apparent span between different levels of intelligence can be captured by analogy to the research of Walter Pitts, arguably the first modern theoretical neuroscientist.

In a landmark paper with Warren McCulloch published in 1943 (when he was merely 20 years old), Pitts and McCulloch developed a theory of neural network computation based on a calculus of logical propositions (McCulloch & Pitts, 1943). In their model, neurons came to represent individual propositions, and the connections between neurons captured conjunction, union, negation, and other operations one might find defined in the Principia Mathematica (Whitehead & Russell, 1925). Never mind that this theory turned out to be largely wrong (as far as we know, individual neurons are not primarily responsible for encoding propositions in a logical calculus, and most neurons do not simply sum up, but rather compute more complex nonlinear functions of, their presynaptic inputs); it set in motion the whole field of cybernetics. In the act of trying to model somewhat high-level cogntive concepts, McCulloch and Pitts put forth a simplified model of the neuron that dominates bottom-up connectionism to this day.

The demands of perception, namely a robustness to changes in viewpoint or lighting, make such a model hard to reconcile with biological brains. In a second paper, Pitts' focus shifted to more graded and statistical computations, namely the issue of how to build invariant representations of physical stimuli (Pitts & McCulloch, 1947). Finally in 1959, and now fully contending with the messy details of neurobiology, Lettvin et al. (1959) reported the computations performed in Frog retinas, which are fundamentally analog and probabilistic. Pitts' later results seemed to rule out his 1943 theory of neural networks, and yet this paper had already set in motion, to some extent, *both* the dominant perspectives in intelligence research.

The rules-and-symbols perspective, which came to be called the Language of Thought Hypothesis (Fodor, 1975; Fodor & Pylyshyn, 1988), and now is often simply referred to as classical cognitive science, held that cognitive computation was best modeled at a symbolic and mathematically abstract level. It was not so much that Language of Thought (LOT) proponents proposed constructive theories for how cognition could arise from neurons as it

was their claim that *whatever the physical basis for intelligence*, it ultimately required some amount of rule and symbol manipulation. On the other hand, neural network modeling–which was first called cybernetics, then connectionism, and these days deep learning–stressed that large parallel networks of simple processing units could perform surprisingly complicated computations and were ultimately a better explanation for the physical basis of intelligence. In this article we will refer to these as the symbolic and connectionist approaches to modeling intelligence. Both of these terms, especially connectionism, are somewhat old-fashioned. We use them simply because they are compact and descriptive.

Current intelligence research enjoys a healthy contribution from both symbolic and connectionist theories. However, there has been a tendency to treat neural networks as self-contained modules which can be cobbled together in a larger system that uses symbolic structures (Mnih et al., 2015). It has been far less common to consider how symbolic computation, namely the use of rich *data structures*, can be *built in* to neural network representations. This article suggests how and why we should want to do this. While the basic building blocks have been around for some time, the computational hurdle of efficient factorization methods have kept some of these ideas from finding wider application. In proposing a new method for factorization in Chapter 2, we make connectionist symbol processing more practical.

1.2 The primacy of factorization

This section will argue that factorization arises everywhere in perception and cognition and that in order to build intelligent systems one must model it explicitly. Factorization problems may be found at every level of the conceptual hierarchy, but we will start with visual perception.

The signal measured by an optical sensor includes the conjunction of many physical factors interacting in the world. For instance, luminance at a point on the sensor is a function of physical properties in the scene, such as the reflectance and orientation of object surfaces, as well as the direction and intensity of incident illumination. An early idea in computer vision was the theory of "intrinsic images," which suggested that these factors might be effectively represented by a series of more basic images, one each for object range, surface reflectance, surface orientation, and incident illumination (Barrow & Tenenbaum, 1978). The physics of light interacting with object surfaces can be fairly complex, but under certain simplifying assumptions it is fundamentally multiplicative, and computer vision researchers have often abstracted this with an intrinsic image called "shading." In Barrow and Tenenbaum (1978) and most subsequent work, the apparent luminance of a point on an object's surface is a *literal product* of its reflectance and shading images.

From among the many follow-on works, we highlight Adelson and Pentland (1996), which took this idea, specifically the factorization of measured luminance into reflectance and shading, and put it on the firmer conceptual footing of Bayesian statistical inference. Their examples also help to illustrate just how fundamental and unconscious this type of perceptual

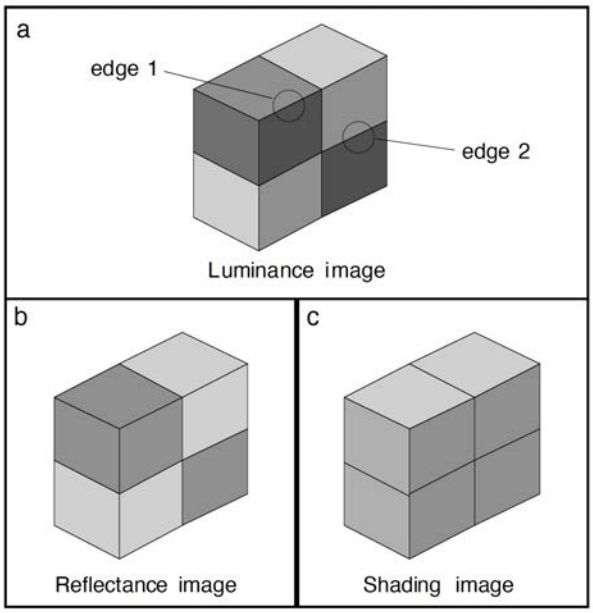


Figure 1.1: Luminance factorization (from Adelson and Pentland (1996))

factorization is for humans. See Figure (1.1), which shows a simple three-dimensional shape rendered as if it is illuminated from above and to its left. The main point of this figure is that two prominent edges in the scene are interpeted to the be the result of very different physical properties of the object–edge 1 is the result of a change in object shape whereas edge 2 is the result of a change in object reflectance. The richness of this interpretation comes in spite of the fact that each edge has *precisely the same* luminance levels on either side–the reader should verify this for themselves. This difference in interpretation of the low-level luminance properties can be explained by the shown *factorization* into intrinsic images for reflectance and shading. The fact that luminance above edge 1 is perceived to be darker than luminance above edge 2 might be explained by a preference for extracting reflectance information from a visual scene. This fits with the longstanding idea that brains have learned to capture, and prefer, certain regularites in visual scenes. Adelson and Pentland modeled this as probabilistic "priors" over certain configurations of lighting, shape, and reflectance.

This general idea, of modeling certain scenes using a decomposable (i.e factored) model in which specific regularities are enforced, has been very influential in the field of computer vision, where intrinsic images are often referred to as a "2.5-D sketch" Marr (1982). See Blanz and Vetter (1999) or Barron and Malik (2015) for particularly interesting applications and methods. We do feel that the intrinsic images formalism places a misplaced emphasis on two-dimensional projections of a fundamentally three-dimensional scene, but it can still be a stepping stone to a full three-dimensional description. In general, uncovering an accurate set of intrinsic images from a single input image is a challenging and unsolved problem. We will outline in Section (1.4) how others have attempted to solve it so far.

At another level of abstraction, one must consider the separation of visual object features defined in some canonical object-based reference frame from the pose of objects in a larger scene. Robust object recognition depends on the ability to separate these two factors. Pitts and McCulloch (1947) examined the related problem of building invariance into a system, which they called representation of "universals." One should not, however, ignore the transformations that define a particular observation of the universal, but rather model this seprately in support of novel inferences. This fuller picture was first sketched by Hinton (1981b) and later fleshed out by Olshausen et al. (1993) in a model that is known as "dynamic routing." The dynamic routing model contains a set of control neurons which multiplicatively gate the transmission of feature activations between successive layers in a multi-layer neural network. This is used to build into the final layer a scale-and-translation-invariant representation of whole objects, while the control neurons maintain a representation of the transformation necessary to bring the original input into a canonical frame of reference. It was hypothesized that the pulvinar nucleus of the thalamus is involved in such a representation of transformations (Olshausen et al., 1993).

Another important notion of factorization is that, in dynamic scenes, humans compute a natural factorization of object features (also called object 'form') and object motion. This allows one to understand dynamic scenes in terms of discrete objects which roughly retain a particular physical form as they undergo coherent transformations, rather than as a soup of moving and unrelated features. A simple demonstration of this factorization can be found in the striking psychophysical percepts induced by random dot stereograms and kinematograms (Julesz, 1971; Milne et al., 2002). One example of how this factorization has been modelled is Cadieu and Olshausen (2012), which proposes a hierarchical, probabilistic generative model in which motion interacts multiplicatively with form in a high-dimensional polar representation.

One longstanding challenge problem in theoretical neuroscience has been the so-called "neural binding problem" (Treisman & Gelade, 1980; Von der Malsburg, 1995; Wolfe & Cave, 1999), which is often meant to loosely capture the above problems of feature grouping–to which objects do particular features belong? Interestingly, this has been posed as almost the *inverse* of a factorization problem. It is said to be one of binding together (read: multiplying) features so as to maintain their association in support of further inference. It almost assumes that atomic features are available at the outset, but this is not quite right. Due to the underlying physics, the signal *comes in* as a multplicative combination, and the job of perception is to factor it first. The perceptual system has to effectively do *both*. It is more about recoding the data–first making the factorization explicit, but then generating a new (and presumably better in some way) multiplicative representation that supports higher-level inferences.

Somewhere above the conceptual level of perception is the uniquely human facility for language. So powerful are the structures of language that many cognitive scientists have suggested that, to some extent, cognition *is* language. This is what came to be known as the Language of Thought Hypothesis (Fodor, 1975). We will illustate how multiplicative combinations arise at every turn in a Language of Thought.

Among the most commonly studied objects in classical cognitive science are propositional phrases like "Mary is the mother of John", or the following:

"Spot bit Jane"
"Jane bit Spot"
"Fido bit John"
"John bit Fido"

Propositions like these contain a predicate `bite` which has two arguments–the `agent` of the bite and the `object` of the bite. To capture any of these particular instances, we make a conceptual distinction between what are called "roles" and "fillers." Fillers are instance-specific cognitive concepts, whereas roles indicate the function of these fillers in a larger data structure. A role concept is used to label the conceptual role of a filler. In the four phrases above, `Spot`, `Fido`, `Jane`, and `John` are filler concepts, while `agent` and `object` roles serve to specify which role any of these fillers play in a particular instance of these propositions. By associating the `agent` role to `Spot`, which we notate $(\texttt{agent}, \texttt{Spot})$, we have a representation of the first phrase, $\{\texttt{bite}, (\texttt{agent}, \texttt{Spot}), (\texttt{object}, \texttt{Jane})\}$ which is distinguishable from the the second phrase $\{\texttt{bite}, (\texttt{agent}, \texttt{Jane}), (\texttt{object}, \texttt{Spot})\}$. The third and fourth phrases have the same *structure* as the first two, but different fillers. This separation of roles from fillers allows us to model the concept of a standalone bite predicate $\texttt{bite}(\texttt{agent}, \texttt{object})$ for which values can be filled in for any particular *instance* of this concept. This is precisely the separation of *variables* from specific *values* that one finds in mathematics or programming languages and has come to be known in cognitive science as "systematicity" (Fodor, 1975; Fodor & Pylyshyn, 1988).

The association of filler concepts to role concepts is a logical conjunction. Moreover, we can build arbitrarily complex cognitive concepts via further conjunction, union, and composition. The result is a set of composite concepts that are formed by two or more–sometimes many more–primitive concepts. Cognition in this framework amounts to constructing and decomposing (multiplying and factoring) these data structures. We will see how these ideas can be represented with an algebra over high-dimensional vectors and how this involves vector multiplication and factorization.

1.3 Vector Symbolic Architectures (VSAs)

This section introduces a framework for cognitive representation, called Vector Symbolic Architectures, that motivates the factorization problem we study in Chapter 2. More than that, it orients much of our past and ongoing work on how to build data structures into neural network representations. We will demonstrate the application of these ideas to modeling visual scenes in Chapter 3.

1.3.1 Building up to VSAs

While connectionists of the 1980s extolled certain properties of neural networks (namely parallel processing with many simple computational units and an ability to learn patterns from data), they did not completely jettison what had been previously developed from the Language of Thought perspective. Particularly, Hinton notes in a 1981 paper the importance of role-filler distinctions and sketches how one might represent these with separate groups of neurons (Hinton, 1981a). A more fully-formed idea comes in 1986, where he advocates capturing role-filler binding via the outer product of two vectors: $(\mathbf{x}\mathbf{y}^\top)_{ij} = x_i y_j$. In connectionist circles this came to be called *conjunctive coding* (Hinton et al., 1986; McClelland & Kawamoto, 1986).

In addition to conjunctive coding, connectionists wrestled throughout the 1980's with mathematical structures such as hierachy and sequence, as this was thought to be among the elements of a Language of Thought which should be explained by connectionism. In some ways, this work culiminated in a special issue on "Connectionist Symbol Processing" in the journal *Artificial Intelligence*. Three papers from this issue set the stage for Vector Symbolic Architectures (VSAs).

In a theory he called Tensor Product Representations, Paul Smolensky fleshed out a connectionist theory of conjunctive coding that was based on the generalized outer product, also called the tensor product (Smolensky, 1990). Notable about this work was its ambition for specifying a full symbolic system as well as its mathematical rigor relative to most connectionist works of the time. It focused on representing two fundamental data structures, strings and trees, and showed that this allowed one to represent operations from the LISP programming language with *superpositions of tensor products*. A physicist by training, Smolensky also made an interesting analogy between connectionist representation of symbolic structures and the study of group representation theory, particularly as it is applied to modeling quantum physics. The terminology and intuition used by Smolensky, ourselves, and other researchers in this area when talking about connectionist symbol systems maintains a kind of loose correspondence to quantum physics to this day. Tensor Product Representations are still being used by Smolensky and collaborators; see, for example, McCoy et al. (2019) and Moradshahi et al. (2019) for more modern applications. However, tensor products applied to vectors have a conceptual and practical limitation, namely that of *closure*. This manifests in the difficulty of superimposing tensor representations of different orders–a remedy for this is to use a kind of placeholder vector to expand the tensor order artificially, but this solution is fairly kludgy (Smolensky, 1992). A more practical issue with the lack of closure for outer products is that the number of elements in a tensor product representation scales exponentially in the order of the tensor. Because the tensor order corresponds to the number of simultaneous conjunctions, this places a real limitation on simply being able to represent complex structures in neurons or on a computer. For instance, the representation of a binary tree with depth k ultimately is represented by a tensor of order k. Smolensky has argued for using specialized low-dimensional vector representations for certain role vectors in these types of data structures, but this removes the robustness and modeling power of

high-dimensional distributed representations, among other isssues.

The second and third in this series of papers were written by Geoff Hinton and Jordan Pollack, respectively. Hinton (1990) outlined desiderata for what he called "reduced descriptions," which, roughly speaking, was meant to suggest a representation of hierarchical structures in which partial information about each layer of the hierachy was immediately decodable, but where full descriptions were compressed and required further sequential processing to uncover. This involves some notion of superposition and composition of concepts. A system that can recursively encode these concepts in fixed-width vectors, like we will introduce in the next section, is one realization of reduced descriptions. Pollack (1990) dealt with the representation of binary trees, but where the encodings of nodes in the tree are *learned* via the backpropagation algorithm. These networks were reportedly hard to train, used local, rather than distributed, representations, and were relatively hard to interpet. However, they did to some extent satisfy the properties of Hinton's reduced descriptions.

1.3.2 Vector Symbolic Architectures

In the 1990's, a family of connectionist models that has come to be called Vector Symbolic Architectures (VSAs) materialized. First proposed in 1991 and then more fully elaborated in his PhD thesis, Tony Plate's Holographic Reduced Representations was the first of model of Vector Symbolic Architecture (Plate, 1991, 1994). In addition to the aforementioned connectionists, Plate was influenced by models that used *convolution* as a mechanism of association, namely those of Willshaw, Murdock, and Metcalf (Willshaw, 1981; Metcalf Eich, 1982; Murdock, 1982, 1983; Metcalf Eich, 1985). With Holographic Reduced Representations (HRRs) Plate elaborated a *framework for building data structures with high-dimensional vectors*. In contrast to prior works, HRR data structures are represented simply with vectors, making them a kind of "reduced" description. One of Plate's many insights in this work was that by *embracing high-dimensionality*, one could use a set of representations which remained closed under the operations of conjunction and union. Plate's work, which was also published in 1995 and later made into a book (Plate, 1995, 2003), characterized the capacity properties of such reduced representations and also emphasized the Fourier-domain duality of convolution and elementwise multiplication, which suggests an even simpler operation for representing conjunctions.

The vectors in HRRs are either real or complex-valued, but it turns out that vectors whose components assume just two states ($\{0, 1\}$ or $\{-1, 1\}$) can also be used in a VSA. This fact was explored by both Pentti Kanerva (Kanerva, 1996, 2009) and Ross Gayler (Gayler, 1998, 2004) under various names, including "Binary Spatter Codes" and the "Multiply, Add, Permute" architecture. N-dimensional vectors which are in $\{0, 1\}^N$ are said to be "binary", while those that are in $\{-1, 1\}^N$ are "bipolar." Among the reasons one might prefer these types of vectors over HRRs (particularly the real-valued variant) is that they more closely align with the discreteness of representation in biological neural networks and lend themselves to scalable and low-power implementation in hardware. For examples of how these types

of VSAs may be a good fit for implementation with emerging device nanotechnologies, see Rahimi et al. (2017), Gupta et al. (2018), and Wu et al. (2018).

There is a qualitative equivalence among all VSAs, allowing our own work to somewhat fluidly cross between the formalisms of each, depending on the application. Therefore we will distill the essential characteristics of a VSA into the following definition:

Definition. A Vector Symbolic Architecture is defined by a set of vectors $\mathcal{V}$ which encode a set of cognitive concepts $\mathcal{C}$, coupled with a similarity metric for comparing any two vectors, and three algebraic structures on this set.

- Cognitive concepts have a corresponding vector representation given by the mapping $\sim\colon \mathcal{C} \to \mathcal{V}$, which is called *encoding*. This mapping can be deterministic, random, or some combination of both. Going from the space of vectors to cognitive concepts (inverting this mapping) is called *decoding*. Most cognitive concepts are "compound" in that they can be decomposed into a set of more fundamental concepts, while a smaller number are "atomic" in that they are irreducible. Vector representations of compound concepts can be constructed by applying the VSA operations to representations of the atomic concepts.

- A similarity metric $sim(\mathbf{x}, \mathbf{y})$ defines how similar two concepts are in the encoded space of vectors. It has the following properties:

  - For atomic concepts that are unrelated, the similarity is close to 0. When the set of concepts is large and diverse, the distribution of vectors chosen randomly from $\mathcal{V}$ is expected to be *highly* peaked at 0. It may be, however, that the encoding of some atomic concepts is designed to capture an underlying structure such that this structure is reflected by vector similarity.
  - For a given vector dimensionality N one typically defines a threshold γ and says that any two vectors whose similarity is $> \gamma$ are similar (notated $\mathbf{x} \approx \mathbf{y}$). This threshold can vary depending on the application. In some applications one may just care about identifying the vector which is *most* similar to a given vector without explicit reference to a threshold.

- The algebraic structures on the set of vectors $\mathcal{V}$ have the follow properties:

  1. Superposition (addition) $\oplus\colon \mathcal{V} \times \mathcal{V} \to \mathcal{V}$

     This is a binary structure on the set $\mathcal{V}$ producing a vector that is **similar** to each of its inputs ($\mathbf{x} \approx \mathbf{x} \oplus \mathbf{y}$, $\mathbf{y} \approx \mathbf{x} \oplus \mathbf{y}$). Superposition is associative and commutative.

  2. Binding (multiplication) $\otimes\colon \mathcal{V} \times \mathcal{V} \to \mathcal{V}$

     This is a binary structure on the set $\mathcal{V}$ producing a vector that is **dissimilar** to each of its inputs ($\mathbf{x} \not\approx \mathbf{x}\otimes\mathbf{y}$, $\mathbf{y} \not\approx \mathbf{x}\otimes\mathbf{y}$) Binding is associative, commutative, and

distributes over superpositions. It is either exactly or approximately invertible. Approximate invertibility means that

$$\forall \mathbf{x}, \mathbf{y} \in \mathcal{V}, \quad \mathbf{x}^{-1} \otimes \mathbf{x} \otimes \mathbf{y} \approx \mathbf{y} \quad \wedge \quad \mathbf{y}^{-1} \otimes \mathbf{x} \otimes \mathbf{y} \approx \mathbf{x}$$

While the output $\mathbf{x} \otimes \mathbf{y}$ is dissimilar to either $\mathbf{x}$ or $\mathbf{y}$, the binding operation **preserves similarity** in a different sense: $\mathbf{x} \approx \mathbf{z} \iff \mathbf{x} \otimes \mathbf{y} \approx \mathbf{z} \otimes \mathbf{y}$, which we comment on in Chapter 3.

3. Permutation $\rho(\cdot)$: $\mathcal{V} \to \mathcal{V}$

   This is a unary structure on the set $\mathcal{V}$ that permutes the elements of a vector. It is often chosen randomly from the symmetric group $\mathcal{S}_n$. Permutation distributes over binding *and* superposition.

Superposition is used to store sets of concepts. For concepts in which an ordering doesn't have any particular meaning, we often store them in superposition. Binding is used to store conjunctions of concepts. This might be labeling fillers with specific roles, or it might be binding fillers together directly. Permutation is used typically to assign an ordering to bound or superimposed elements. When we use a combination of all three operations to build symbolic vectors, we are encoding structure as a **"sum of products of permutations"** (of vectors).

The specific operations used in each Vector Symbolic Architecture differ slightly, and the notations $\oplus$, $\otimes$, and $\rho(\cdot)$ are meant as placeholders. All the operations, however, share the properties specified above. It is the complementary nature of these properties that confers surprising representational richness to every Vector Symbolic Architecture. Superposition in all VSAs involves pointwise addition: $\big(\mathbf{x} + \mathbf{y}\big)_i = x_i y_i$, but some VSAs follow this with an additional thresholding operation. Binding gets slightly fancier for HRRs, which come in two variants. The variant with real-valued vectors uses circular convolution $\circledast$ to bind vectors: $\big(\mathbf{x} \circledast \mathbf{y}\big)_i = \sum_{k=1}^{N} x_k y_{(i-k+1)\%N}$, where $\%N$ means "modulo-N". The variant of HRRs with complex-valued vectors uses the Hadamard product $\big(\mathbf{x} \odot \mathbf{y}\big)_i = x_i y_i$, as does Gayler's Multiply-Add-Permute (MAP) architecture. The MAP and complex-valued HRR architectures will feature most prominently in the rest of this dissertation.

We may refer to VSA representations as living in a vector space–after all they are just high-dimensional vectors–but the binding operation induces a nonlinear structure that is quite a bit richer than what comes from the narrow technical definition of a vector space. Certainly we can add vectors and multiply them by scalars (which we did not mention above but sometimes comes up in applications). However, the fact that we can multiply the vectors themselves is an entirely different beast. In addition, permutation adds a complement to both addition and multiplication that is not present for a generic algebraic field.

The VSA framework depends on relatively high dimensionality, roughly speaking $N \geq 1000$, where N is the number of elements in each vector. In an N-dimensional vector space the number of exactly-orthogonal vectors is always N, but the number of *approximately orthogonal* vectors is exponential in N. This is sometimes refered to as the quasi-orthogonal

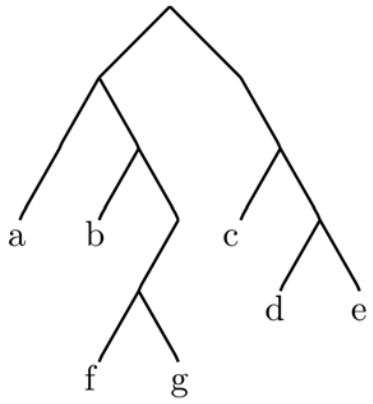


Figure 1.2: A binary tree

dimension of the space (Kainen & Kurkova, 1993); it can be explained by the Johnson-Lindenstrauss lemma and is ultimately a result of concentration of measure phenomena (Boucheron et al., 2013). Plate addresses this analytically and through simulation in the appendix of his thesis (Plate, 1994). The similarity metric of a VSA is in every case related to angles, and therefore quasi-orthogonality, between vectors. One can say that there is plenty of "room" to place vectors which are dissimilar. The effect is that vectors–even those in $\mathbb{R}^N$ and $\mathbb{C}^N$–start to behave in an almost symbolic or discrete way.

1.3.3 A VSA factorization problem

We now explain how a tree data structure can be encoded into a single high-dimensional vector using VSAs. Consider the tree depicted in Figure 1.2. First, each leaf in the tree is assigned a random vector $\mathbf{a}, \mathbf{b}, \ldots, \mathbf{g} \in \mathcal{V}$. We also assign random vectors $\mathbf{q}_{\texttt{left}}$ and $\mathbf{q}_{\texttt{right}}$ that are used to describe position in the tree. Moving from the root of the tree to a particular leaf involves a sequence of `left` and `right` turns. The order of these turns is represented by permutation $\rho(\cdot)$. The number of times permutation is applied indicates depth within the tree: $\mathbf{q}_{\texttt{left}}$ is a `left` turn at depth 0, $\rho(\mathbf{q}_{\texttt{left}})$ is a `left` turn at depth 1, $\rho^2(\mathbf{q}_{\texttt{left}})$ is a `left` turn at depth 2, and so on. A sequence of turns is represented by the *binding* of these vectors, e.g., $\mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{left}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}})$ corresponds to three left turns. We can then attach to each leaf its position in the tree, again with binding, e.g., $\mathbf{a} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{left}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}})$. Finally,

the representation for the whole tree is collapsed into a single vector, $\mathbf{t}$, via superposition:

$$\begin{aligned}\mathbf{t} \;=\;& \mathbf{a} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{left}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \\ &\oplus \mathbf{b} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \\ &\oplus \mathbf{c} \otimes \mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \\ &\oplus \mathbf{d} \otimes \mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{right}}) \otimes \rho^3(\mathbf{q}_{\texttt{left}}) \\ &\oplus \mathbf{e} \otimes \mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{right}}) \otimes \rho^3(\mathbf{q}_{\texttt{right}}) \\ &\oplus \mathbf{f} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{right}}) \otimes \rho^3(\mathbf{q}_{\texttt{left}}) \otimes \rho^4(\mathbf{q}_{\texttt{left}}) \\ &\oplus \mathbf{g} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{right}}) \otimes \rho^3(\mathbf{q}_{\texttt{left}}) \otimes \rho^4(\mathbf{q}_{\texttt{right}})\end{aligned} \tag{1.1}$$

The vector $\mathbf{t}$ encodes the information of this entire tree so that we can flexibly query the data structure using VSA operations. For instance, we can find the identity of the leaf located at position `left`, `right`, `left` by "unbinding" the representation of this location from $\mathbf{t}$. When we unbind with this vector (call it a query), it distributes through the superposition and cancels out with itself, leaving the atomic vector attached to that location "exposed":

$$\left(\mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}})\right)^{-1} \otimes \mathbf{tree} = \mathbf{b} \oplus \mathit{noise} \tag{1.2}$$

The noise term arises because the query distributes through the sum. The other terms combine with the query, but remain quasi-orthogonal to the atomic vectors we're looking for $(\mathbf{a}, \mathbf{b}, \ldots, \mathbf{g})$. The vector $\mathbf{b} \oplus \mathit{noise}$ will have high similarity with atom $\mathbf{b}$ and will be decoded with high probability by nearest neighbor or associative memory lookup among the atoms. According to the normal VSA setup, the scaling of how many terms can be in superposition while still decoding $\mathbf{b}$ correctly scales linearly in N (Plate, 1994). Increasing the storage capacity of such structures is an ongoing topic of research (Frady, Kleyko, et al., 2018), and various encoding techniques such as "chunking" help as well. Under the set of normal VSA assumptions it is typical that several hundred nodes could be stored in superposition this way.

The above encoding of the tree data structure is flexible in the following sense: instead of asking for the label at a specific position, we can ask for the position of a specific label (essentially the problem of tree search). For instance, the query that exposes the position of leaf `c` is simply:

$$\mathbf{c}^{-1} \otimes \mathbf{t} = \mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \oplus \mathit{noise} \tag{1.3}$$

This presents a new challenge, however, because we still need to decode the composite vector $\mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \oplus \mathit{noise}$ into its atomic parts. Ignoring the permutation for now (we will show later how to deal with it), it is hopefully clear that this is a factorization problem. More generally, this is a structure (a bound combination of three or more atomic vectors) that shows up throughout Vector Symbolic Architectures. Let us call this the VSA Factorization Problem, which we will formalize in Chapter 2.

Until recently, there has been no good solution to this factorization problem. Past applications of VSAs have tried to sidestep this issue by limiting the hierarchical depth of the

data structures or by using a brute force approach to consider all possible combinations when necessary (Plate, 2000; Cox et al., 2011). In the tree example above this is tantamount to exhaustively enumerating all possible traversals of the tree. We will demonstrate a much better solution in Chapters 2 and 3.

1.4 Previous approaches to factorization

We return to some of the other factorization problems covered in the previous section. Most approaches to solving these problems have had essentially one thing in common–they have treated factorization as an *optimization problem*. There is nothing wrong with this per se, but it contrasts with how we will approach factorization in Chapter 2.

Adelson and Pentland (1996) considered an intrinsic images factorization problem which we illustrated in Figure 1.1. Their solution estimates lighting, shading, and reflectance in an iterative algorithm which is essentially computing maximum a posteriori estimates for each factor in turn. Without any constraints on the factors–the "priors" they enforce–this setup would be fundamentally ill-posed. However, by heavily constraining valid solutions for lighting, shape and reflectance, their algorithm can generate estimates which seem to align with subjective percepts of these factors–consider Figure 1.3. The algorithm decomposes shading into a representation of 3D shape and an incident light source. Following an initialization in which reflectance is made to completely explain the input image, each factor is updated in turn (first shape, then lighting, then reflectance) and this continues until a convergence criterion is met. See in Figure 1.3 two equally-valid factorizations of the scene. It is good to remember that this kind of symmetry often pops up in factorization problems (in what amounts to essentially a sign-ambiguity).

This is a bit of a toy problem, and Adelson and Pentland specifically note that their cost function has no local minima and only two global minima (the ones shown). But take note of the fact that they have a cost function at all. While they do not give detail sufficient to reimplement their algorithm, it is still clear that the updates to each factor are determined by trying to minimize a cost. It is this carefully-designed cost and the fact that updates *descend* this cost that gives a guarantee of convergence to reasonable solutions.

A scaled-up realization of Adelson and Pentland's idea comes in the form of Barron and Malik (2015). This highly-impressive paper seeks to generate intrinsic images for a much more challenging dataset of images by using significantly more complicated representation of, and priors imposed on, each factor. Ultimately however, the authors are solving an optimization problem. They go to great lengths to condition the problem (representing shape in a multiscale Gaussian pyramid, "whitening" the reflectance representation, and numerous other tricks) because otherwise it is so poorly behaved. It is also relatively slow–the algorithm takes between 2 and 10 minutes on a color 1 Megapixel image, according to the authors. This should suggest the inherent difficulty of the intrinsic images factorization problem. We also suggest, however, that casting this as an intricate optimization is partially to blame.

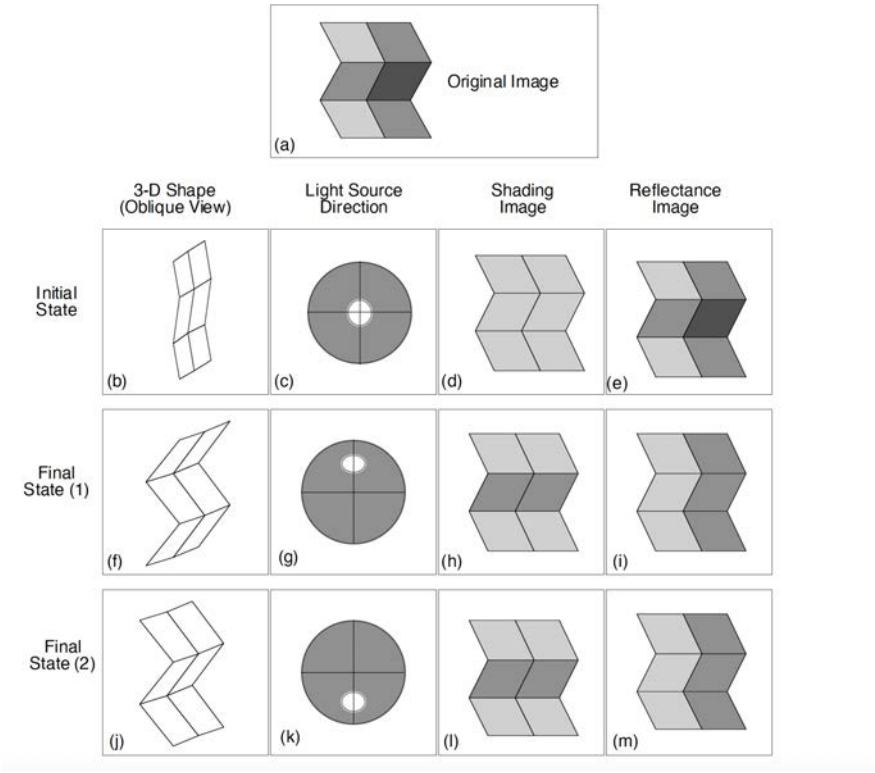


Figure 1.3: Adelson and Pentland's alternating minimization

The modern deep learning approach to intrinsic image factorization dispenses with iterative solutions altogether. A single "feedforward" computation generates images from a set of shared convolutional features (see, e.g. Janner et al. (2017)). These models produce impressive results on artificial data and in fact we adopt elements of this approach for some of our own applications in Chapter 3. However, our feeling is that iterative solutions, i.e. networks that have an element of recurrence, will ultimately be important in factoring real-world scenes.

Dynamic routing networks and neural models of form-and-motion factorization (Olshausen et al., 1993; Cadieu & Olshausen, 2012) are closer to being neurobiologically plausible, but they ultimately also tie themselves to optimization–the dynamics specified by each follow the gradient of a cost function. The second of these two models has an additional goal, the pursuit of sparse solutions, which likely aids in finding a factorization of form and motion. The study of so-called tensor decomposition methods offers a whole family of algorithms for factoring Smolensky's Tensor Product Representations. We cover this topic at some length in Appendix A.4. In Chapter 2, we apply an algorithm from this family to the VSA factorization problem as a benchmark against Resonator Networks, our proposed solution.

Summary

This chapter has argued that multiplicative codes and their attendant factorization problems are fundamental to both perception and cognition. We have additionally argued that what is missing from main-stream connectionist models is a facility for building and manipulating data structures and that Vector Symbolic Architectures offer a way forward. The use of VSAs comes up against a factorization problem, like many other levels of intelligence. On this particular problem (but we would argue more generally throughout intelligence research) neither the symbolic nor connectionist perspectives offers a good solution. A search for such a solution is our topic in Chapter 2.

Previously published work in Chapters 2 and 3

Our work on Resonator Networks has been published previously in Frady et al. (2020) and Kent et al. (2020). Early abstracts and posters on this work were given in Frady, Kent, and Olshausen (2018) and Frady, Kent, Kanerva, et al. (2018). The Vector Symbolic Scene Transformer was proposed in Kent and Olshausen (2017). Our preliminary results on visual scene analogies were presented in Maudgalya et al. (2020). All other analysis and results are previously unpublished.

Chapter 2

Resonator Networks

We limit our analysis of Resonator Networks to a particular definition of the factorization problem, which may seem somewhat abstract, but in fact applies to practical usage of Vector Symbolic Architectures (VSAs). We consider "bipolar" vectors, whose elements are ± 1, used in the popular "Multiply, Add, Permute (MAP)" VSA (Gayler, 1998, 2004). These ideas extend to other VSAs, although we leave a detailed analysis to future work.

The core challenge of factorization is that inferring the factors of a composite object amounts to searching through an enormous space of possible solutions. Resonator Networks do this, in part, by "searching in superposition," a notion that we make precise in Section 2.2. There are in fact many ways to search in superposition, and we introduce a number of them in Section 2.4 as a benchmark for our model and to understand what makes our approach different. A Resonator Network is simply a nonlinear dynamical system designed to solve a particular factorization problem. It is defined by equations (2.7) and (2.8), each representing two separate variants of the network. The system is named for the way in which correct factorizations seemingly 'resonate' out of what is initially an uninformative network state. The size of the factorization problem that can be reliably solved, as well as the speed with which solutions are found, characterizes the performance of all the approaches we introduce–in these terms, Resonator Networks are by far the most effective.

The main results are as follows:

1. We characterize stability at the correct solution, showing that one variant of Resonator Networks is always stable, while the other has stability properties related to classical Hopfield Networks. We show that Resonator Networks are less stable than Hopfield Networks because of a phenomenon we refer to as percolated noise (Section 2.5.1).

2. We define "operational capacity" as a metric of factorization performance and use it to compare Resonator Networks against six benchmark algorithms. We find that Resonator Networks have dramatically higher operational capacity (Section 2.5.2).

3. Through simulation, we determine that operational capacity scales as a quadratic function of vector dimensionality. This quantity is proportional to the number of idealized

neurons in a Resonator Network (also Section 2.5.2).

4. We propose a theory for *why* Resonator Networks perform well on this problem (Section 2.5.6).

2.1 Statement of the problem

We formalize the factorization problem in the following way: $\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_F$ are sets of vectors called 'codebooks'. The fth codebook contains D_f 'codevectors' $\mathbf{x}_1^{(f)}, \mathbf{x}_2^{(f)}, \ldots, \mathbf{x}_{D_f}^{(f)}$

$$\mathbb{X}_f := \{\mathbf{x}_1^{(f)}, \mathbf{x}_2^{(f)}, \ldots, \mathbf{x}_{D_f}^{(f)}\} \quad \forall f = 1, 2, \ldots, F$$

and these vectors all live in $\{-1, 1\}^N$. A composite vector $\mathbf{c}$ is generated by computing the Hadamard product $\odot$ of F vectors, one drawn from each of the codebooks $\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_F$.

$$\mathbf{c} = \mathbf{x}_\star^{(1)} \odot \mathbf{x}_\star^{(2)} \odot \ldots \odot \mathbf{x}_\star^{(F)}$$

$$\mathbf{x}_\star^{(1)} \in \mathbb{X}_1,\ \mathbf{x}_\star^{(2)} \in \mathbb{X}_2,\ \ldots,\ \mathbf{x}_\star^{(F)} \in \mathbb{X}_F$$

The factorization problem we wish to study is

$$\begin{array}{ll} \text{given} & \mathbf{c},\ \mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_F \\ \text{find} & \mathbf{x}_\star^{(1)} \in \mathbb{X}_1,\ \mathbf{x}_\star^{(2)} \in \mathbb{X}_2,\ \ldots\ \mathbf{x}_\star^{(F)} \in \mathbb{X}_F \\ \text{such that} & \mathbf{c} = \mathbf{x}_\star^{(1)} \odot \mathbf{x}_\star^{(2)} \odot \ldots \odot \mathbf{x}_\star^{(F)} \end{array} \tag{2.1}$$

Our assumption in this chapter is that the factorization of $\mathbf{c}$ into F codevectors, one from each codebook, is unique. Then, the total number of composite vectors that can be generated by the codebooks is M:

$$M := \prod_{f=1}^{F} D_f$$

The problem involves searching among M possible factorizations to find the one that generates $\mathbf{c}$. We will refer to M as the search space size, and at some level it captures the difficulty of the problem. The problem size is also influenced by N, the dimensionality of each vector.

Suppose we were to solve (2.1) using a brute force strategy. We might form all possible composite vectors from the sets $\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_F$, one at a time, until we generate the vector $\mathbf{c}$, which would indicate the appropriate factorization. Assuming no additional information is available, the number of trials taken to find the correct factorization is a uniform random variable $K \sim \mathcal{U}\{1, M\}$ and thus $\mathbf{E}[K] = \frac{M+1}{2}$. If instead we could easily store all of the composite vectors ahead of time, we could compare them to any new composite vector via a single matrix-vector inner product, which, given our uniqueness assumption, will yield a value of N for the correct factorization and values strictly less than N for all other factorizations. The matrix containing all possible composite vectors requires MN bits to store.

The core issue is that M scales *very* poorly with the number of factors and number of possible codevectors to be entertained. If $F = 4$ (4 factors) and $D_f = 100\ \forall f$ (100 possible codevectors for each factor), then $M = 100{,}000{,}000$. In the context of Vector Symbolic Architectures, it is common to have $N = 10{,}000$. Therefore, the matrix with all possible composite vectors would require $\approx 125\,\text{GB}$ to store. We aspire to solve problems of this size (and much larger), which are clearly out of reach for brute-force approaches. Fortunately, they are solvable using Resonator Networks.

2.2 Factoring by search in superposition

In our problem formulation (2.1) the factors interact multiplicatively to form $\mathbf{c}$, and this lies at the heart of what makes it hard to solve. One way to attempt a solution is to produce an estimate for each factor in turn, alternating between updates to a single factor on its own, with the others held fixed. In addition, it may make sense to simulatenously entertain all of the vectors in each $\mathbb{X}_f$, in some proportion that reflects our current confidence in each one being part of the correct solution. We call this *searching in superposition* and it is the general approach we take throughout the paper. What we mean by 'superposition' is that the estimate for the fth factor, $\hat{\mathbf{x}}^{(f)}$, is given by $\hat{\mathbf{x}}^{(f)} = g(\mathbf{X}_f \mathbf{a}_f)$ where $\mathbf{X}_f$ is a matrix with each column a vector from $\mathbb{X}_f$. The vector $\mathbf{a}_f$ contains the coefficients that define a linear combination of the elements of $\mathbb{X}_f$, and $g(\cdot)$ is a function from $\mathbb{R}^N$ to $\mathbb{R}^N$, which we will call the activation function. In this work we consider the identity $g : \mathbf{x} \mapsto \mathbf{x}$, the sign function $g : \mathbf{x} \mapsto \text{sgn}(\mathbf{x})$, and nothing else. Other activation functions are appropriate for the other variants of Resonator Networks (for instance where the vectors are complex-valued), but we leave a discussion of this to future work. 'Search' refers to the method by which we adapt $\mathbf{a}_f$ over time. The estimate for each factor leads to an estimate for $\mathbf{c}$ denoted by $\hat{\mathbf{c}}$:

$$\hat{\mathbf{c}} := \hat{\mathbf{x}}^{(1)} \odot \hat{\mathbf{x}}^{(2)} \odot \ldots \odot \hat{\mathbf{x}}^{(F)} = g(\mathbf{X}_1 \mathbf{a}_1) \odot g(\mathbf{X}_2 \mathbf{a}_2) \odot \ldots \odot g(\mathbf{X}_F \mathbf{a}_F) \tag{2.2}$$

Suppose $g(\cdot)$ is the identity. Then $\hat{\mathbf{c}}$ becomes a *multilinear* function of the coefficients $\mathbf{a}_1, \mathbf{a}_2, \ldots \mathbf{a}_F$.

$$\hat{\mathbf{c}} = \hat{\mathbf{x}}^{(1)} \odot \hat{\mathbf{x}}^{(2)} \odot \ldots \odot \hat{\mathbf{x}}^{(F)} = \mathbf{X}_1 \mathbf{a}_1 \odot \mathbf{X}_2 \mathbf{a}_2 \odot \ldots \odot \mathbf{X}_F \mathbf{a}_F \tag{2.3}$$

While this is a 'nice' relationship in the sense that it is linear in each of the coefficients $\mathbf{a}_f$ separately (with the others held fixed), it is unfortunately not convex with respect to the coefficients taken all at once. We can rewrite it as a sum of M different terms, one for each of the possible factorizations of $\mathbf{c}$:

$$\hat{\mathbf{c}} = \sum_{d_1, d_2, \ldots, d_F} \Big((\mathbf{a}_1)_{d_1} (\mathbf{a}_2)_{d_2} \ldots (\mathbf{a}_F)_{d_F} \Big)\ \mathbf{x}^{(1)}_{d_1} \odot \mathbf{x}^{(2)}_{d_2} \odot \ldots \odot \mathbf{x}^{(F)}_{d_F} \tag{2.4}$$

Where d_1 ranges from 1 to D_1, d_2 ranges from 1 to D_2, and so on. The term in parentheses is a scalar that weights each of the possible Hadamard products. Our estimate $\hat{\mathbf{c}}$ is, at any given

time, purely a superposition of *all* the possible factorizations. Moreover, the superposition weights $\Big((\mathbf{a}_1)_{d_1}(\mathbf{a}_2)_{d_2}\dots(\mathbf{a}_F)_{d_F}\Big)$ can be approximately recovered from $\hat{\mathbf{c}}$ alone by computing the cosine similarity between $\hat{\mathbf{c}}$ and the vector $\mathbf{x}^{(1)}_{d_1}\odot\mathbf{x}^{(2)}_{d_2}\odot\dots\odot\mathbf{x}^{(F)}_{d_F}$. The source of 'noise' in this approximation is the fact that $\mathbf{x}^{(1)}_{d_1}\odot\mathbf{x}^{(2)}_{d_2}\odot\dots\odot\mathbf{x}^{(F)}_{d_F}$ will have a nonzero inner product with the other vectors in the sum. When the codevectors are uncorrelated and high-dimensional, this noise is quite small: $\hat{\mathbf{c}}$ transparently reflects the proportion with which it contains each of the possible factorizations. When $g(\cdot)$ is the sign function, *this property is retained.* The vector $\hat{\mathbf{c}}$ is no longer an exact superposition, but the scalar $\Big((\mathbf{a}_1)_{d_1}(\mathbf{a}_2)_{d_2}\dots(\mathbf{a}_F)_{d_F}\Big)$ can still be decoded from $\hat{\mathbf{c}}$ in the same way–the vector $\hat{\mathbf{c}}$ is still an approximate superposition of all the possible factorizations, with the weight for each of these determined by the coefficients $\mathbf{a}_f$. This property, that thresholded superpositions retain relative similarity to each of their superimposed components, is heavily relied on throughout Kanerva's and Gayler's work on Vector Symbolic Architectures (Kanerva, 1996; Gayler, 1998).

One last point of notation before introducing our solution to the factorization problem–we define the vector $\hat{\mathbf{o}}^{(f)}$ to be the product of the estimates for the *other* factors:

$$\hat{\mathbf{o}}^{(f)} := \hat{\mathbf{x}}^{(1)}\odot\dots\odot\hat{\mathbf{x}}^{(f-1)}\odot\hat{\mathbf{x}}^{(f+1)}\odot\dots\odot\hat{\mathbf{x}}^{(F)} \tag{2.5}$$

This will come up in each of the algorithms under consideration and simplify our notation. The notation will often include an explicit dependence on time t like so: $\hat{\mathbf{x}}_f[t] = g(\mathbf{X}_f\mathbf{a}_f[t])$. Each of the algorithms considered in this chapter updates one factor at a time, with the others held fixed so, at a given time t, we will update the factors in order 1 to F, although this is a somewhat arbitrary choice. Including time dependence with $\hat{\mathbf{o}}^{(f)}$, we have

$$\hat{\mathbf{o}}^{(f)}[t] := \hat{\mathbf{x}}^{(1)}[t+1]\odot\dots\odot\hat{\mathbf{x}}^{(f-1)}[t+1]\odot\hat{\mathbf{x}}^{(f+1)}[t]\odot\dots\odot\hat{\mathbf{x}}^{(F)}[t] \tag{2.6}$$

which makes explicit that at the time of updating $\hat{\mathbf{x}}_f$, the factors 1 to $(f-1)$ have already been updated for this 'iteration' t while the factors $(f+1)$ to F have yet to be updated.

2.3 Resonator Networks

A Resonator Network is a nonlinear dynamical system designed to solve the factorization problem (2.1), and it can be interpreted as a neural network in which idealized neurons are connected in a very particular way. We define two separate variants of this system, which differ in terms of this pattern of connectivity. A Resonator Network with outer product (OP) weights is defined by

$$\hat{\mathbf{x}}^{(f)}[t+1] = \operatorname{sgn}\Big(\mathbf{X}_f\mathbf{X}_f^\top\big(\hat{\mathbf{o}}^{(f)}[t]\odot\mathbf{c}\big)\Big) \tag{2.7}$$

Suppose $\hat{\mathbf{x}}^{(f)}[t+1]$ indicates the state of a population of neurons at time $t+1$. Each neuron receives an input $\hat{\mathbf{o}}^{(f)}[t]\odot\mathbf{c}$, modified by synapses modeled as a row of a "weight matrix"

$\mathbf{X}_f\mathbf{X}_f^\top$. This "synaptic current" is passed through the activation function $\text{sgn}(\cdot)$ in order to determine the output, which is either $+1$ or -1. Most readers will be familiar with the weight matrix $\mathbf{X}_f\mathbf{X}_f^\top$ as the so-called "outer product" learning rule of classical Hopfield Networks (Hopfield, 1982). This has the nice interpretation of Hebbian learning (Hebb, 1949) in which the strength of synapses between any two neurons (represented by this weight matrix) depends solely on their pairwise statistics over some dataset, in this case the codevectors.

Prior to thresholding in (2.7), the matrix-vector product $\mathbf{X}^\top\big(\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}\big)$ produces coefficients $\mathbf{a}_f[t]$ which, when premultiplied by $\mathbf{X}_f$, generate a vector in the linear subspace spanned by the codevectors (the columns of $\mathbf{X}_f$). This projection does not minimize the squared distance between $\big(\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}\big)$ and the resultant vector. Instead, the matrix $\big(\mathbf{X}_f^\top\mathbf{X}_f\big)^{-1}\mathbf{X}_f^\top$ produces such a projection, the so-called *Ordinary Least Squares* projection onto $\mathcal{R}(\mathbf{X}_f)$. This motivates the second variant of our model, Resonator Networks with Ordinary Least Squares (OLS) weights:

$$\begin{aligned}\hat{\mathbf{x}}^{(f)}[t+1] &= \text{sgn}\Big(\mathbf{X}_f\big(\mathbf{X}_f^\top\mathbf{X}_f\big)^{-1}\mathbf{X}_f^\top\big(\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}\big)\Big) \\ &:= \text{sgn}\Big(\mathbf{X}_f\mathbf{X}_f^\dagger\big(\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}\big)\Big) \end{aligned} \tag{2.8}$$

where we have used the notation $\mathbf{X}_f^\dagger$ to indicate the Moore-Penrose psuedoinverse of the matrix $\mathbf{X}_f$. Hopfield Networks with this type of synapse were first proposed by Personnaz, Guyon, and Dreyfus (Personnaz et al., 1986), who called this the "projection" rule.

If, contrary to what we have defined in (2.7) and (2.8), the input to each sub-population of neurons was $\hat{\mathbf{x}}^{(f)}[t]$, its own previous state, then one would in fact have a ("Bipolar") Hopfield Network. In our case however, rather than being autoassociative, in which $\hat{\mathbf{x}}^{(f)}[t+1]$ is a direct function of $\hat{\mathbf{x}}^{(f)}[t]$, our dynamics are heteroassociative, basing updates on the states of the *other* factors. This change has a dramatic effect on the network's convergence properties and is also in some sense what makes Resonator Networks useful in solving the factorization problem, a fact that we will elaborate on in the following sections. We imagine F separate subpopulations of neurons which evolve together in time, each one responsible for estimating a different factor of $\mathbf{c}$. For now we have just specified this as a discrete-time network in which updates are made one-at-a-time, but it can be extended as a continuous-valued, continuous-time dynamical system along the same lines as was done for Hopfield Networks (Hopfield, 1984). In that case, we can think about these F subpopulations of neurons evolving in a truly parallel way. In discrete-time, one has the choice of making 'asynchronous' or 'synchronous' updates to the factors, in a sense analogous to Hopfield Networks. Our formulation of $\hat{\mathbf{o}}^{(f)}[t]$ in (2.6) follows the asynchronous convention, which we find to converge faster.

In practice, we will have to choose an initial state $\hat{\mathbf{x}}^{(f)}[0]$ using no knowledge of the correct codevector $\mathbf{x}_\star^{(f)}$ other than the fact it is one of the elements of the codebook $\mathbb{X}_f$. Therefore, we set $\hat{\mathbf{x}}^{(f)}[0] = \text{sgn}\Big(\sum_j \mathbf{x}_j^{(f)}\Big)$, which, as we have said above, has approximately equal cosine similarity to each term in the sum.

2.3.1 Difference between OP weights and OLS weights

The difference between outer product weights and Ordinary Least Squares weights is via $\left(\mathbf{X}_f^\top \mathbf{X}_f\right)^{-1}$, the inverse of the so-called Gram matrix for $\mathbf{X}_f$, which contains inner products between each codevector. If the codevectors are orthogonal, the Gram matrix is $N\,\mathbf{I}$, with $\mathbf{I}$ the identity matrix. When N is large (roughly speaking $>$ 5,000), and the codevectors are chosen randomly i.i.d. from $\{-1, 1\}^N$, then they will be *very nearly* orthogonal, making $N\,\mathbf{I}$ a close approximation. Clearly, in this setting, the two variants of Resonator Networks produce nearly the same dynamics. In section 2.5.2, we define and measure a performance metric called operational capacity in such a way that does not particularly highlight the difference between the dynamics, i.e. it is the setting where codevectors are nearly orthogonal. In general, however, the dynamics are clearly different. In our experience, applications that contain correlations between codevectors may enjoy higher operational capacity under Ordinary Least Squares weights, but it is hard to say whether this applies in every setting.

One application-relevant consideration is that, because $\mathbf{X}_f$ consists of entries that are $+1$ and -1, the outer product variant of a Resonator Network has an integer-valued weight matrix and can be implemented without any floating-point computation–hardware with large binary and integer arithmetic circuits can simulate this model very quickly. Coupled with noise tolerance properties we will establish in Section 2.5.5, this makes Resonator Networks (and more generally, VSAs) a good fit for emerging device nanotechnologies (Rahimi et al., 2017).

2.4 The optimization approach

An alternative strategy for solving the factorization problem is to define a loss function which compares the current estimate $\hat{\mathbf{c}} := \hat{\mathbf{x}}^{(1)} \odot \hat{\mathbf{x}}^{(2)} \odot \ldots \odot \hat{\mathbf{x}}^{(F)}$ with the composite that is to be factored, $\mathbf{c}$, choosing the loss function and a corresponding constraint set so that the global minimizer of this loss over the constraints yields the correct solution to (2.1). One can then design an algorithm that finds the solution by minimizing this loss. This is the approach taken by *optimization* theory. Here we consider algorithms that search in superposition, setting $\hat{\mathbf{x}}^{(f)} = g(\mathbf{X}_f \mathbf{a}_f)$ just as Resonator Networks, but that instead take the optimization approach.

Let the loss function be $\mathcal{L}(\mathbf{c}, \hat{\mathbf{c}})$ and the feasible set for each $\mathbf{a}_f$ be C_f. We write this as a fairly generic optimization problem:

$$\begin{aligned} \underset{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_F}{\text{minimize}} \quad & \mathcal{L}\big(\mathbf{c}, g(\mathbf{X}_1\mathbf{a}_1) \odot g(\mathbf{X}_2\mathbf{a}_2) \odot \ldots \odot g(\mathbf{X}_F\mathbf{a}_F)\big) \\ \text{subject to} \quad & \mathbf{a}_1 \in C_1, \mathbf{a}_2 \in C_2, \ldots, \mathbf{a}_F \in C_F \end{aligned} \tag{2.9}$$

What makes a particular instance of this problem remarkable depends on our choices for $\mathcal{L}(\cdot, \cdot)$, $g(\cdot)$, $C_1, C_2, \ldots, C_F$, and the structure of the vectors in each codebook. Different algorithms may be appropriate for this problem, depending on these details, and we propose *six* candidate algorithms in this chapter, which we refer to as the "benchmarks". It is in

contrast to the benchmark algorithms that we can more fully understand the performance of Resonator Networks–our argument, which we will develop in the Results section, is that Resonator Networks strike a more natural balance between exploring the high-dimensional state space and using local information to move towards the solution. The benchmark algorithms are briefly introduced in Section 2.4.1, but they are each discussed at some length in the Appendix, including Table A.2, which compiles the dynamics specified by each. We provide implementations of each algorithm in the small software library that accompanies this work[1].

2.4.1 Benchmark algorithms

A common thread among the benchmark algorithms is that they take the activation function $g(\cdot)$ to be the identity $g : \mathbf{x} \mapsto \mathbf{x}$, making $\hat{\mathbf{c}}$ a multilinear function of the coefficients, as we discussed in section 2.2. We experimented with other activation functions, but found none for which the optimization approach performed better. We consider two straightforward loss functions for comparing $\mathbf{c}$ and $\hat{\mathbf{c}}$. The first is one half the squared Euclidean norm of the error, $\mathcal{L} : \mathbf{x}, \mathbf{y} \mapsto \frac{1}{2}||\mathbf{x} - \mathbf{y}||_2^2$, which we will call the squared error for short, and the second is the negative inner product $\mathcal{L} : \mathbf{x}, \mathbf{y} \mapsto -\langle \mathbf{x}, \mathbf{y} \rangle$. The squared error is minimized by $\hat{\mathbf{c}} = \mathbf{c}$, which is also true for the negative inner product when $\hat{\mathbf{c}}$ is constrained to $[-1, 1]^N$. Both of these loss functions are convex, meaning that $\mathcal{L}(\mathbf{c}, \hat{\mathbf{c}})$ is a convex function of each $\mathbf{a}_f$ separately[2]. *Some* of the benchmark algorithms constrain $\mathbf{a}_f$ directly, and when that is the case, our focus is on three different convex sets, namely the simplex $\Delta_{D_f} := \{\mathbf{x} \in \mathbb{R}^{D_f} \mid \sum_i x_i = 1, x_i \geq 0 \ \forall i\}$, the unit ℓ_1 ball $\mathcal{B}_{||\cdot||_1}[1] := \{\mathbf{x} \in \mathbb{R}^{D_f} \mid ||\mathbf{x}||_1 \leq 1\}$, and the closed zero-one hypercube $[0, 1]^{D_f}$. Therefore, solving (2.9) with respect to each $\mathbf{a}_f$ *separately* is a convex optimization problem. In the case of the negative inner product loss $\mathcal{L} : \mathbf{x}, \mathbf{y} \mapsto -\langle \mathbf{x}, \mathbf{y} \rangle$ and simplex constraints $C_f = \Delta_{D_f}$, it is a bonafide linear program. The correct factorization is given by $\mathbf{a}_1^\star, \mathbf{a}_2^\star, \ldots, \mathbf{a}_F^\star$ such that $\hat{\mathbf{x}}^{(f)} = \mathbf{X}_f \mathbf{a}_f^\star = \mathbf{x}_\star^{(f)} \ \forall f$, which we know to be vectors with a single entry 1 and the rest 0–these are the standard basis vectors $\mathbf{e}_i$ (where $(\mathbf{e}_i)_j = 1$ if $j = i$ and 0 otherwise). The initial states $\mathbf{a}_1[0], \mathbf{a}_2[0], \ldots \mathbf{a}_F[0]$ must be set with no prior knowledge of the correct factorization so, similar to how we do for Resonator Networks, we set each element of $\mathbf{a}_f[0]$ to the same value (which in general depends on the constraint set).

2.4.1.1 Alternating Least Squares

Alternating Least Squares (ALS) locally minimizes the squared error loss in a fairly straightforward way: for each factor, one at a time, it solves a least squares problem for $\mathbf{a}_f$ and updates the current state of the estimate $\hat{\mathbf{c}}$ to reflect this new value, then moves onto the

[1] https://github.com/spencerkent/resonator-networks

[2] through the composition of an affine function with a convex function

next factor and repeats. Formally, the updates given by Alternating Least Squares are:

$$\begin{aligned}\mathbf{a}_f[t+1] &= \arg\min_{\mathbf{a}_f} \ \tfrac{1}{2}\big|\big|\, \mathbf{c} - \hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{X}_f \mathbf{a}_f[t] \,\big|\big|_2^2 \\ &= \left(\boldsymbol{\xi}^\top \boldsymbol{\xi}\right)^{-1} \boldsymbol{\xi}^\top \mathbf{c} \quad | \quad \boldsymbol{\xi} := \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big)\, \mathbf{X}_f \end{aligned} \tag{2.10}$$

Alternating Least Squares is an algorithm that features prominently in the tensor decomposition literature (Kolda & Bader, 2009), but while ALS has been successful for a particular type of tensor decomposition, there are a few details which make our problem different from what is normally studied (see Appendix A.4). The updates in ALS are quite greedy–they exactly solve each least squares subproblem. It may make sense to more gradually modify the coefficients, a strategy that we turn to next.

2.4.1.2 Gradient-following algorithms

Another natural strategy for solving (2.9) is to make updates that incorporate the gradient of $\mathcal{L}$ with respect to the coefficients–each of the next 5 algorithms does this in a particular way (we write out the gradients for both loss functions in Appendix A.5). The squared error loss is globally minimized by $\hat{\mathbf{c}} = \mathbf{c}$, so one might be tempted to start from some initial values for the coefficients and make gradient updates $\mathbf{a}_f[t+1] = \mathbf{a}_f[t] - \eta \, \nabla_{\mathbf{a}_f} \mathcal{L}$. In Appendix A.5.1 we discuss why this does not work well–the difficulty is in being able to guarantee that the loss function is smooth enough that gradient descent iterates with a fixed stepsize will converge. Instead, the algorithms we apply to the squared error loss utilize a dynamic stepsize.

Iterative Soft Thresholding: The global minimizers of (2.9) are maximally sparse, $||\mathbf{a}_f^\star||_0 = 1$. If one aims to minimize the squared error loss while *loosely* constrained to sparse solutions, it may make sense to solve the problem with Iterative Soft Thesholding (ISTA). The dynamics for ISTA are given by equation (A.3) in Table A.2.

Fast Iterative Soft Thresholding: We also considered Fast Iterative Soft Thesholding (FISTA), an enhancement due to Beck and Teboulle (2009), which utilizes Nesterov's momentum for accelerating first-order methods in order to alleviate the sometimes slow convergence of ISTA (Bredies & Lorenz, 2008). Dynamics for FISTA are given in equation (A.4).

Projected Gradient Descent: Another benchmark algorithm we considered was Projected Gradient Descent on the negative inner product loss, where updates were projected onto either the simplex or unit ℓ_1 ball (A.5). A detailed discussion of this approach can be found in Appendix A.7.

Multiplicative Weights: This is an algorithm that can be applied to either loss function, although we found it worked best on the negative inner product. It very elegantly enforces a simplex constraint on $\mathbf{a}_f$ by maintaining a set of auxilliary variables, the 'weights', which are used to set $\mathbf{a}_f$ at each iteration. See equation (A.6) for the dynamics of Multiplicative Weights, as well as Appendix A.8.

Map Seeking Circuits: The final algorithm that we considered is called Map Seeking Circuits. Map Seeking Circuits are neural networks designed to solve invariant pattern recognition problems using the principle of superposition. Their dynamics are based on the gradient, but are different from what we have introduced so far–see equation (A.7) and Appendix A.9.

2.4.2 Contrasting Resonator Networks with the benchmarks

Convergence of the benchmarks

A remarkable fact about the benchmark algorithms is that *each one converges for all initial conditions*, which we directly prove, or refer to results proving, in the Appendix. That is, given any starting coefficients $\mathbf{a}_f[0]$, their dynamics reach fixed points which are local minimizers of the loss function. In some sense, this property is an immediate consequence of treating factorization as an optimization problem–the algorithms we chose as the benchmarks were *designed* this way. Convergence to a local minimizer is a desirable property, but unfortunately the fundamental non-convexity of the optimization problem implies that this may not guarantee good local minima in practice. In Section 2.5 we establish a standardized setting where we measure how likely it is that these local minima are actually *global* minima. We find that as long as M–the size of the search space–is small enough, each of these algorithms can find the global minimizers reliably. The point at which the problem becomes too large to reliably solve is what we call the operational capacity of the algorithm, and is a main point of comparison with Resonator Networks.

An algorithmic interpretation of Resonator Networks

The benchmark algorithms generate estimates for the factors, $\hat{\mathbf{x}}^{(f)}[t]$, that move through the interior of the $[-1, 1]$ hypercube. Resonator Networks, on the other hand, do not. The $\text{sgn}(\cdot)$ function 'bipolarizes' inputs to the nearest vertex of the hypercube, and this highly nonlinear function, which not only changes the length but also the *angle* of an input vector, is key. We know the solutions $\mathbf{x}_\star^{(f)}$ exist at vertices of the hypercube and these points are very special geometrically in the sense that in high dimensions, most of the mass of $[-1, 1]^N$ is concentrated relatively *far* from the vertices–a fact we will not prove here but that is based on standard results from the study of concentration inequalities (Boucheron et al., 2013). Our motivation for using the $\text{sgn}(\cdot)$ activation function is that moving through the interior of the hypercube while searching for a factorization is unwise, a conjecture for which we will provide some empirical support in the Results section.

One useful interpretation of OLS Resonator Network dynamics is that the network is computing a bipolarized version of Alternating Least Squares. Suppose we were to take the dynamics specified in (2.10) for making ALS updates to $\mathbf{a}_f[t+1]$, but we also bipolarize the vector $\hat{\mathbf{x}}^{(f)}[t+1]$ at the end of each step. When each $\hat{\mathbf{x}}^{(f)}[t+1]$ is bipolar, the vector $\hat{\mathbf{o}}^{(f)}[t]$

is bipolar and we can simplify $\left(\boldsymbol{\xi}^\top\boldsymbol{\xi}\right)^{-1}\boldsymbol{\xi}^\top$:

$$\begin{aligned}\hat{\mathbf{o}}^{(f)}[t] \in \{-1,1\}^N \iff \left(\boldsymbol{\xi}^\top\boldsymbol{\xi}\right)^{-1}\boldsymbol{\xi}^\top &= \left(\mathbf{X}_f^\top \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big)^2 \mathbf{X}_f\right)^{-1}\mathbf{X}_f^\top \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big) \\ &= \left(\mathbf{X}_f^\top\mathbf{X}_f\right)^{-1}\mathbf{X}_f^\top\, \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big) \\ &= \mathbf{X}_f^\dagger\, \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big) \end{aligned} \tag{2.11}$$

Now $\mathbf{a}_f[t+1] = \mathbf{X}_f^\dagger\big(\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}\big)$, which one can see from equation (2.8) is precisely the update used by Resonator Networks with OLS weights. An important word of caution on this observation: it is somewhat of a misnomer to call this algorithm Bipolarized Alternating Least Squares, because at each iteration it is *not* solving a least squares problem, and this conceals a profound difference. To set $\mathbf{a}_f[t+1] = \mathbf{X}_f^\dagger\big(\hat{\mathbf{o}}^{(f)}[t]\odot\mathbf{c}\big)$ is to take the term $g(\mathbf{X}_f\mathbf{a}_f[t])$ present in the loss function and treat the activation function $g(\cdot)$ *as if* it were linear, which it clearly is not. These updates are not computing a Least Squares solution at each step. We actually lose the guarantee of global convergence that comes with Alternating Least Squares, but *this is an exchange well worth making*, as we will show in the Results section.

Unlike Hopfield Networks, which have a Lyapunov function certifying their global asymptotic stability, no such function (that we know of) exists for a Resonator Network. While $\hat{\mathbf{c}} = \mathbf{c}$ is always a fixed point of the OLS dynamics, a network initialized to a random state is not guaranteed to converge. We have observed trajectories that collapse to limit cycles and seemingly-chaotic trajectories that do not converge in any reasonable time. One *a priori* indication that this is the case comes from a simple rewriting of two-factor Resonator Network dynamics that concatenates the states for each factor into a single statespace. To make the transformation exact, we appeal to the continuous-time version of Resonator Networks, which, just like Hopfield networks, define dynamics in terms of time derivatives of the pre-activation state $\dot{\mathbf{u}}^{(f)}(t) = \mathbf{X}_f\mathbf{X}_f^\dagger\big(\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}\big)$, with $\hat{\mathbf{x}}^{(f)}(t) = g(\mathbf{u}^{(f)}(t))$. We write down the continuous-time dynamics à la autoassociative Hopfield Networks:

$$\begin{pmatrix}\dot{\mathbf{u}}^{(1)}(t)\\ \dot{\mathbf{u}}^{(2)}(t)\end{pmatrix} = \left(\begin{array}{c|c} \mathbf{0} & \mathbf{X}_1\mathbf{X}_1^\dagger\,\mathrm{diag}(\mathbf{c}) \\ \hline \mathbf{X}_2\mathbf{X}_2^\dagger\,\mathrm{diag}(\mathbf{c}) & \mathbf{0}\end{array}\right)\begin{pmatrix}\hat{\mathbf{x}}^{(1)}(t)\\ \hat{\mathbf{x}}^{(2)}(t)\end{pmatrix}$$

One can see that the weight matrix is non-symmetric, which has a simple but important consequence: autoassociative networks with non-symmetric weights cannot be guaranteed to converge *in general*. This result, first established by Cohen and Grossberg (Cohen & Grossberg, 1983) and then studied throughout the Hopfield Network literature, is not quite as strong as it may sound, in the sense that symmetry is a sufficient, but not necessary, condition for convergence. One can design a globally-convergent autoassociative network with asymmetric weights (Xu et al., 1996), and moreover, adding a degree of asymmetry has been advocated as a technique to reduce the influence of spurious fixed points (Hertz et al., 1986; Singh et al., 1995; Chengxiang et al., 2000).

Resonator Networks have a large and practical regime of operation, where M (the problem size) is small enough, in which non-converging trajectories are extremely rare. It is simple

to deal with these events, making the model still useful in practice despite the lack of a convergence guarantee. It has also been argued in several places (see Van Vreeswijk and Sompolinsky (1996), for example) that cyclic or chaotic trajectories may be useful to a neural system, including in cases where there are multiple plausible states to entertain. This is just to say that we feel the lack of a convergence guarantee is not a critical weakness of our model, but rather an interesting and potentially useful characteristic. We attempted many different modifications to the model's dynamics which would provably cause it to converge, but these changes always hindered its ability to solve the factorization problem. We emphasize that unlike all of the models in Section 2.4.1, a Resonator Network is *not* descending a loss function. Rather, it makes use of the fact that:

- Each iteration is a bipolarized ALS update–it *approximately* moves the state towards the Least Squares solution for each factor.
- The correct solution is a fixed point (guaranteed for OLS weights, highly likely for OP weights).
- There may be a sizeable 'basin of attraction' around this fixed point, which the iterates help us descend.
- The number of spurious fixed points (which do not give the correct factorization) is relatively small.

This last point is really what distinguishes Resonator Networks from the benchmarks, which we will establish in Section 2.5.6.

2.5 Results

We present a characterization of Resonator Networks along three main directions. The first direction is the stability of the solutions $\mathbf{x}_\star^{(f)}$, which we relate to the stability of classical Hopfield networks. The second is a fundamental measure of factorization capability we call the "operational capacity". The third is the speed with which factorizations are found. We argue that the marked difference in factorization performance between our model and the benchmark algorithms lies in the relative *scarcity of spurious fixed points* enjoyed by Resonator Network dynamics. We summarize the main results **in bold** throughout this section.

In each of the simulations we choose codevectors randomly i.i.d. from the discrete uniform distribution over the vertices of the hypercube–each element of each codevector is a Rademacher random variable (assuming the value -1 with probability 0.5 and $+1$ with probability 0.5). We generate $\mathbf{c}$ by choosing one vector at random from each of the F codebooks and then computing the Hadamard product among these vectors. The reason we choose vectors randomly is because it makes the analysis of performance somewhat easier and more

standardized, and it is the setting in which most of the well-known results on Hopfield Network capacity apply–we will make a few connections to these results. It is also the setting in which we typically use the Multiply, Add, Permute VSA architecture (Gayler, 2004) and therefore these results on random vectors are immediately applicable to a variety of existing works.

2.5.1 Stable-solution capacity with outer product weights

Suppose $\hat{\mathbf{x}}^{(f)}[0] = \mathbf{x}_\star^{(f)}$ for all f (we initialize it to the correct factorization; this will also apply to any t at which the algorithm comes upon $\mathbf{x}_\star^{(f)}$ on its own). What is the probability that the state stays there–i.e. that the correct factorization is a fixed point of the dynamics? This is the basis of what researchers have called the "capacity" of Hopfield Networks, where $\mathbf{x}_\star^{(f)}$ are patterns that the network has been trained to store. We choose to call it the "stable-solution capacity" in order to distinguish it from operational capacity, which we define in Section 2.5.2.

We first note that this analysis is necessary only for Resonator Networks with outer product weights–Ordinary Least Squares weights guarantee that the solutions are stable, and this is one of the variant's desirable properties. If $\hat{\mathbf{x}}^{(f)}[0] = \mathbf{x}_\star^{(f)}$ for all f, then factor 1 in a Resonator Network "sees" an input $\mathbf{x}_\star^{(1)}$ at time $t = 1$. For OLS weights, the vector $\mathbf{X}_1\mathbf{X}_1^\dagger\mathbf{x}_\star^{(1)}$ is exactly $\mathbf{x}_\star^{(1)}$ by the definition of orthogonal projection. True for all subsequent factors, this means that for OLS weights, $\mathbf{x}_\star^{(f)}$ is always a fixed point.

For a Resonator Network with outer product weights, we must examine the vector $\mathbf{\Gamma} := \mathbf{X}_f\mathbf{X}_f^\top\left(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\right)$ at each f, and changing from the psuedoinverse $\mathbf{X}_f^\dagger$ to the transpose $\mathbf{X}_f^\top$ makes the situation significantly more complicated. At issue is the probability that Γ_i has a sign different from $\left(\mathbf{x}_\star^{(f)}\right)_i$, i.e. that there is a bitflip in any particular component of the updated state. In general one may not care whether the state is completely stable–it may be tolerable that the dynamics flip some small fraction of the bits of $\mathbf{x}_\star^{(f)}$ as long as it does not move the state too far away from $\mathbf{x}_\star^{(f)}$. Amit, Gutfreund, and Sompolinsky (Amit et al., 1985, 1987) established that in Hopfield Networks, an avalanche phenomenon occurs where bitflips accumulate and the network becomes essentially useless for values of $D_f > 0.138N$, at which point the approximate bitflip probability is 0.0036. While we don't attempt any of this complicated analysis on Resonator Networks, we do derive an expression for the bitflip probability of any particular factor that accounts for bitflips which "percolate" from factor to factor through the vector $\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}$.

We start by noting that for factor 1, this bitflip probability is the same as a Hopfield network. Readers familiar with the literature on Hopfield Networks will know that with N and D_f reasonably large (approximately $N \geq 1{,}000$ and $D_f \geq 50$) Γ_i can be well-approximated by a Gaussian with mean $\left(\mathbf{x}_\star^{(f)}\right)_i (N + D_f - 1)$ and variance $(N-1)(D_f-1)$;

see appendix A.10 for a simple derivation. This is the *Hopfield bitflip probability* h_f:

$$\begin{aligned} h_f &:= Pr\big[\ \big(\hat{\mathbf{x}}^{(f)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(f)}\big)_i\ \big] \\ &= \Phi\Big(\frac{-N - D_f + 1}{\sqrt{(N-1)(D_f-1)}}\Big) \end{aligned} \tag{2.12}$$

Where Φ is the cumulative density function of the Normal distribution. Hopfield Networks are often specified with the diagonal of $\mathbf{X}_f\mathbf{X}_f^\top$ set to all zeros (having "no self-connections"), in which case the bitflip probability is $\Phi\Big(\frac{-N}{\sqrt{(N-1)(D_f-1)}}\Big)$. For large N and D_f this is often simplified to $\Phi(-\sqrt{N/D_f})$, which may be the expression most familiar to readers. Keeping the diagonal of $\mathbf{X}_f\mathbf{X}_f^\top$ makes the codevectors more stable (see appendix A.10) and while there are some arguments in favor of eliminating it, we have found Resonator Networks to exhibit better performance by keeping these terms.

In Appendix A.10 we derive the bitflip probability for an arbitrary factor in a Resonator Network with outer product weights. This probability depends on whether a component of the state has already been flipped by the previous $f-1$ factors, which is what we call *percolated noise* passed between the factors, and which increases the bitflip probability. The four relevant probabilities are:

$$r_f := Pr\big[\ \big(\hat{\mathbf{x}}^{(f)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(f)}\big)_i\ \big] \tag{2.13}$$

$$n_f := Pr\big[\ \big(\hat{\mathbf{o}}^{(f+1)}[0] \odot \mathbf{c}\big)_i \neq \big(\mathbf{x}_\star^{(f+1)}\big)_i\ \big] \tag{2.14}$$

$$r_{f'} := Pr\big[\ \big(\hat{\mathbf{x}}^{(f)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(f)}\big)_i \ \big|\ \big(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\big)_i = \big(\mathbf{x}_\star^{(f)}\big)_i\ \big] \tag{2.15}$$

$$r_{f''} := Pr\big[\ \big(\hat{\mathbf{x}}^{(f)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(f)}\big)_i \ \big|\ \big(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\big)_i \neq \big(\mathbf{x}_\star^{(f)}\big)_i\ \big] \tag{2.16}$$

Equation (2.13) is the probability of a bitflip compared to the correct value, the *Resonator bitflip probability*. Equation (2.14) gives the probability that the *next* factor will see a net bitflip, a bitflip which has percolated through the previous factors. Equations (2.15) and (2.16) give the probability of a bitflip conditioned on whether or not this factor sees a net bitflip, and they are *different*. It should be obvious that

$$r_f = r_{f'}(1 - n_{f-1}) + r_{f''}n_{f-1} \tag{2.17}$$

and also that

$$n_f = r_{f'}(1 - n_{f-1}) + (1 - r_{f''})n_{f-1} \tag{2.18}$$

We show via straightforward algebra in Appendix A.10 that the conditional probabilities $r_{f'}$ and $r_{f''}$ can be written recursively in terms of n_f:

$$r_{f'} = \Phi\Big(\frac{-N(1-2n_{f-1}) - (D_f - 1)}{\sqrt{(N-1)(D_f-1)}}\Big) \tag{2.19}$$

$$r_{f''} = \Phi\Big(\frac{-N(1-2n_{f-1}) + (D_f - 1)}{\sqrt{(N-1)(D_f-1)}}\Big) \tag{2.20}$$

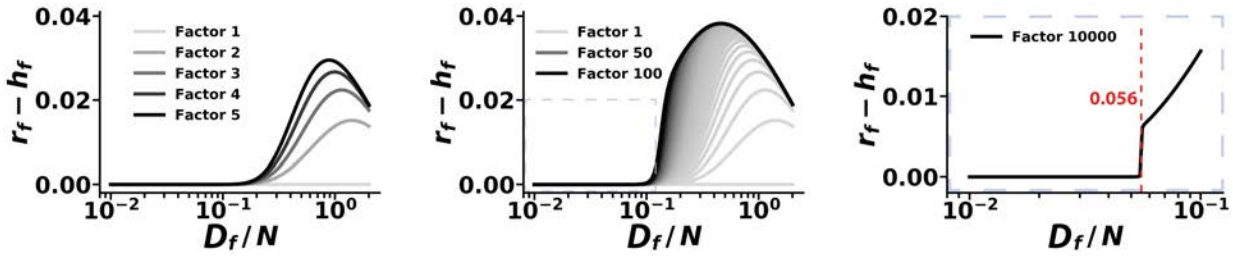
The Resonator bitflip probability r_f has to be computed recursively using these expressions. The base case is $n_0 = 0$ and this is sufficient to compute all the other probabilities–in particular, it implies that $r_1 = h_1 = \Phi\big(\frac{-N-D_1+1}{\sqrt{(N-1)(D_1-1)}}\big)$, which we have previously indicated. We can verify these equations in simulation, and the agreement is very good–see Figure A.5 in the Appendix, which measures r_f.

The main analytical result in this section is the sequence of equations (2.17) - (2.20), which allow one to compute the bitflip probabilities for each factor in an outer product Resonator Network. The fact that r_f in general must be split between the two conditional probabilities and that there is a dependence on n_{f-1} is what makes it different, for all but the first factor, from the bitflip probability for a Hopfield Network (compare eqs. (2.19) and (2.20) against eq. (2.12)). But how much different? We are interested in the quantity $r_f - h_f$.

Here is a simple intuition for what this is capturing: suppose there are F Hopfield Networks all evolving under their own dynamics–they are running simultaneously but not interacting in any way. At time $t = 0$, the bitflip probabilities $h_1, h_2, \ldots, h_F$ for the networks are all the same; there is nothing special about any particular one. A Resonator Network, however, is like a set of F Hopfield networks that have been wired up to receive input $\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}$, which reflects the state of the *other* factors. The networks are no longer independent. In particular, a bitflip in factor f gets passed onto factors $f+1$, $f+2$, and so on. This affects the bitflip probability of these other factors, and the magnitude of this effect, which we call percolated noise, is measured by $r_f - h_f$.

Let us first note that for a Hopfield network *with self connections* the maximum bitflip probability is 0.02275, which occurs at $D_f = N$. The ratio D_f/N is what determines the bitflip probability. Please see Appendix A.10 for an explanation. Percolated noise is measured by the difference $r_f - h_f$, which we plot in Figure 2.1. Part (a) shows just five factors, illustrating that $r_1 = h_1$, but that $r_f \geq h_f$ in general. To see if there is some limiting behavior, we simulated 100 and 10,000 factors; the differences $r_f - h_f$ are also shown in Figure 2.1. In the limit of large F there appears to be a phase change in residual bitflip probability that occurs at $D_f/N = 0.056$. In the Hopfield Network literature this is a very important number. It gives the point at which the codevectors transition away from being global minimizers of the Hopfield Network energy function. When D_f/N falls in between 0.056 and 0.138, the codevectors are only local minimizers, and there exist *spin-glass* states that have lower energy. We do not further explore this phase-change phenomenon, but leave the (in all likelihood, highly technical) analysis to future work.

In conclusion, the second major result of the section is that we have shown, via simulation, that **for $\mathbf{D_f/N \leq 0.056}$, the stability of a Resonator Network with outer product weights is the same as the stability of a Hopfield Network. For $\mathbf{D_f/N > 0.056}$, percolated noise between the factors causes the Resonator Network to be strictly less stable than a Hopfield Network**.



(a) 1 to 5 factors (b) 1 to 100 factors (c) 10,000 factors

Figure 2.1: Extra bitflip probability $r_f - h_f$ due to percolated noise. In the limit of large F, there appears to be a phase change at $D_f/N = 0.056$. Below this value Resonator Networks are just as stable as Hopfield Networks, but above this value they are strictly less stable (by the amount $r_f - h_f$).

2.5.2 Operational capacity

We now define a new notion of capacity that is more appropriate to the factorization problem. This performance measure, called the *operational capacity*, gives an expression for the maximum size of factorization problem that can be solved with high probability. This maximum problem size, which we denote by M_{max}, varies as a function of the number of elements in each vector N and the number of factors F. It gives a very practical characterization of performance, and will form the basis of our comparison between Resonator Networks and the benchmark algorithms we introduced in Section 2.4.1. When the problem size M is below the operational capacity of the algorithm, one can be quite sure that the correct factorization will be efficiently found.

Definition. The $\{p, k\}$ operational capacity of a factorization algorithm that solves (2.1) is the largest search space size M_{max} such that the algorithm, when limited to a maximum number of iterations k, gives a total accuracy $\geq p$.

We now define what we mean by total accuracy. Each algorithm we have introduced attempts to solve the factorization problem (2.1) by initializing the state $\hat{\mathbf{x}}^{(f)}[0]$ and letting the dynamics evolve until some termination criterion is met. It is possible that the final state $\hat{\mathbf{x}}^{(f)}[\infty]$ may not equal the correct factors $\mathbf{x}_\star^{(f)}$ at each and every component, but we can 'decode' each $\hat{\mathbf{x}}^{(f)}[\infty]$ by looking for its nearest neighbor (with respect to Hamming distance or cosine similarity) among the vectors in its respective codebook $\mathbb{X}_f$. This distance computation involves only D_f vectors, rather than M, which was what we encountered in one of the brute-force strategies of Section 2.1. Compared to the other computations involved in finding the correct factorization out of M total possibilities, this last step of decoding has a very small cost, and we always 'clean up' the final state $\hat{\mathbf{x}}^{(f)}[\infty]$ using its nearest neighbor in the codebook. We define the total accuracy to be the sum of accuracies for inferring each factor, which is $1/F$ if the nearest neighbor to $\hat{\mathbf{x}}^{(f)}$ is $\mathbf{x}_\star^{(f)}$ and 0 otherwise. For instance,

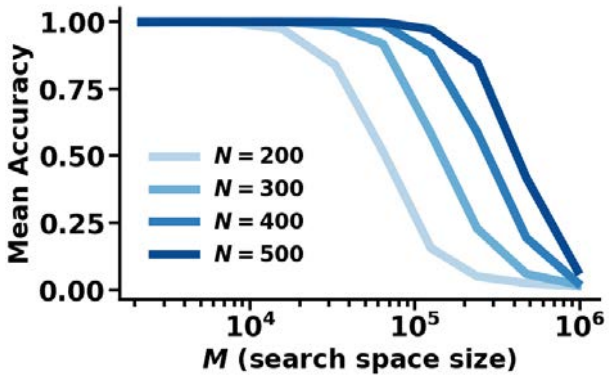


Figure 2.2: Accuracy as a function of M for Resonator Network with outer product weights. Three factors ($F = 3$), average over 5,000 random trials.

correctly inferring one of three total factors gives a total accuracy of 1/3, two of three is 2/3, and three of three is 1.

Analytically deriving the expected total accuracy appears to be quite challenging, especially for a Resonator Network, because it requires that we essentially predict how the nonlinear dynamics will evolve over time. There may be a region around each $\mathbf{x}_\star^{(f)}$ such that states in this region rapidly converge to $\mathbf{x}_\star^{(f)}$, the so-called basin of attraction, but our initial estimate $\hat{\mathbf{x}}_{(f)}[0]$ is likely not in the basin of attraction, and it is hard to predict when, if ever, the dynamics will enter this region. Even for Hopfield Networks, which obey much simpler dynamics than a Resonator Network, it is known that so-called "frozen noise" is built up in the network state, making the shapes of the basins highly anisotropic and difficult to analyze (Amari & Maginu, 1988). Essentially all of the analytical results on Hopfield Networks consider only the stability of $\mathbf{x}_\star^{(f)}$ as a (very poor) proxy for how the model behaves when it is initialized to other states. This less useful notion of capacity, the stable-solution capacity, was what we examined in the previous section.

We can, however, estimate the total accuracy by simulating many factorization problems, recording the fraction of factors that were correctly inferred over many, many trials. We remind the reader that our results in this chapter pertain to factorization of randomly-drawn vectors which bear no particular correlational structure, but that notions of total accuracy and operational capacity would be relevant, and specific, to factorization of non-random vectors. We first note that for fixed vector dimensionality N, the empirical mean of the total accuracy depends strongly on M, the search space size. We can see this clearly in Figure 2.2. We show this phenomenon for a Resonator Network with outer product weights, but this general behavior is true for all of the algorithms under consideration–one can always make the search space large enough that expected total accuracy goes to zero.

Our notion of operational capacity is concerned with the M that causes expected total accuracy to drop below some value p. We see here that there are a range of values M for which the expected total accuracy is 1.0, beyond which this ceases to be the case. For

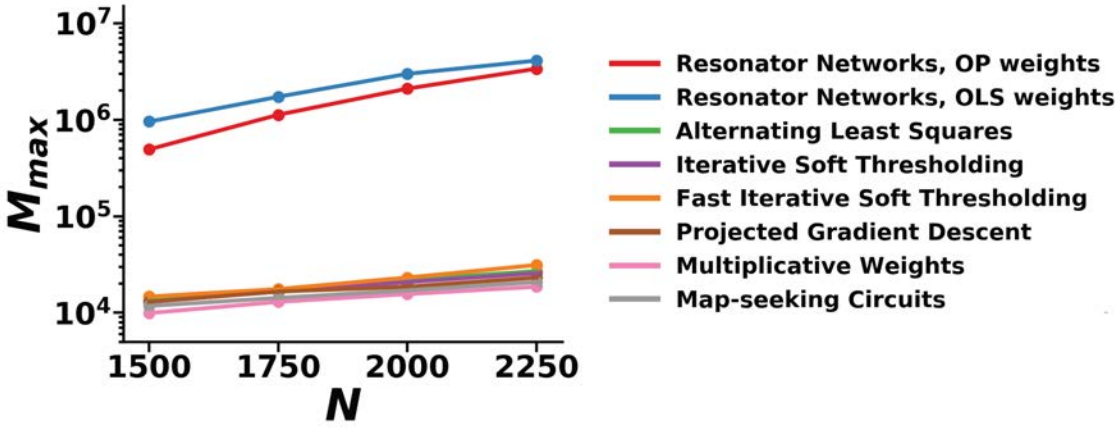


Figure 2.3: Operational capacity is dramatically higher for Resonator Networks (blue and red above) than for any of the benchmark algorithms. These points represent the size of factorization problem that can be solved reliably. Shown is operational capacity for $F = 3$ factors. The gap is similarly large for other F, see plot for $F = 4$ in the Appendix.

all values of M within this range, the algorithm essentially always solves the factorization problem.

In this chapter we estimate operational capacity when $p = 0.99$ ($\geq 99\%$ of factors were inferred correctly) and $k = 0.001M$ (the model can search over at most 1/1,000 of the entire search space). These choices are largely practical: $\geq 99\%$ accuracy makes the model very reliable in practice, and this operating point can be estimated from a reasonable number (3,000 to 5,000) of random trials. Setting $k = 0.001M$ allows the number of iterations to scale with the size of the problem, but restricts the algorithm to only consider a small fraction of the possible factorizations. While a Resonator Network has no guarantee of convergence, it almost always converges in far less than $0.001M$ iterations, so long as we stay in this high-accuracy regime. Operational capacity is in general a function of N and F, which we will discuss shortly.

2.5.2.1 Resonator Networks have superior operational capacity

We estimated the operational capacity of the benchmark algorithms in addition to the two variants Resonator Networks. Figure 2.3 shows the operational capacity estimated on several thousand random trials, where we display M_{max} as a function of N for problems with three factors. One can see that **the operational capacity of Resonator Networks is roughly two orders of magnitude greater than the operational capacity of the other algorithms**. Each of the benchmark algorithms has a slightly different operational capacity (due to the fact that they each have different dynamics and will, in general, find different solutions) but they are all similarly poor compared to the two variants of Resonator Networks. See a similar plot for $F = 4$ in Appendix A.2.

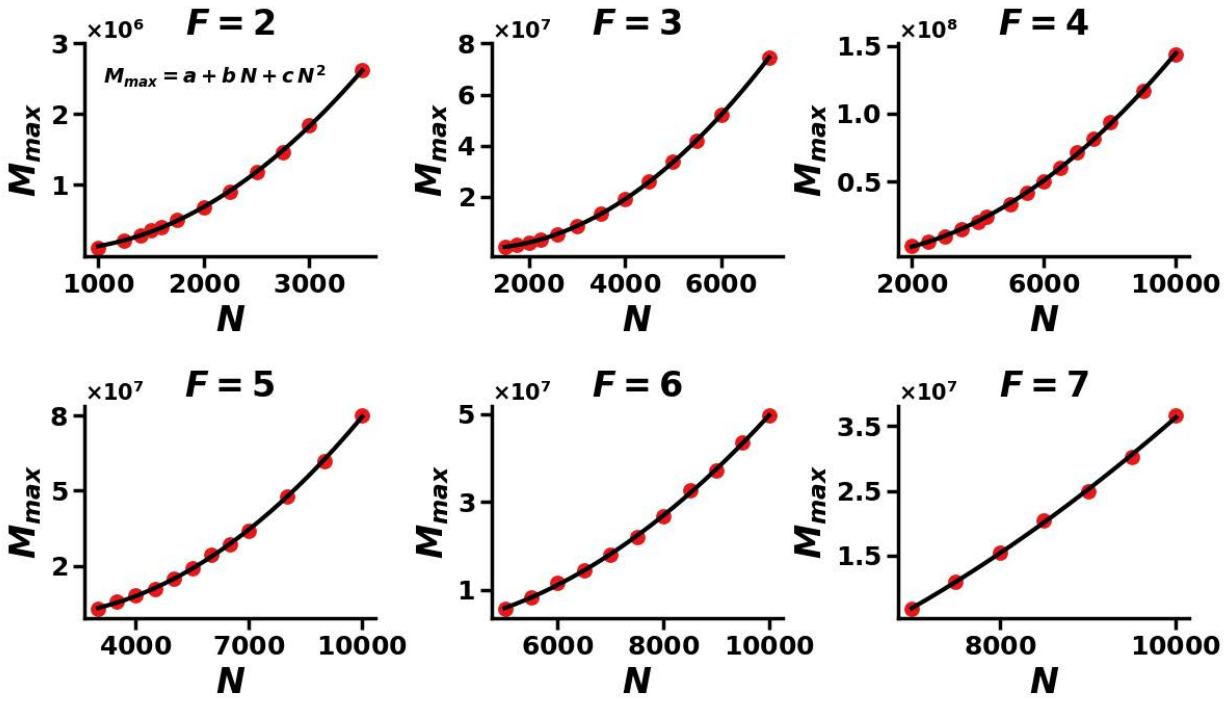
As N increases, the performance difference between the two variants of Resonator Networks starts to disappear, ostensibly due to the fact that $\mathbf{X}_f\mathbf{X}_f^\dagger \approx \mathbf{X}_f\mathbf{X}_f^\top$. The two variants are different in general, but the simulations in this chapter do not particularly highlight the difference between them. Except for Alternating Least Squares, each of the benchmark algorithms has at least one hyperperparameter that must be chosen–we simulated many thousand random trials with a variety of hyperparameter settings for each algorithm and chose the hyperparameter values that performed best on average. We list these values for each of the algorithms in the Appendix. All of the benchmark algorithms converge on their own and the tunable stepsizes make a comparison of the number of iterations non-standardized, so we did not impose a maximum number of iterations on these algorithms–the points shown represent the best the benchmark algorithms can do, even when not restricted to a maximum number of iterations.

2.5.2.2 Operational capacity scales quadratically in N

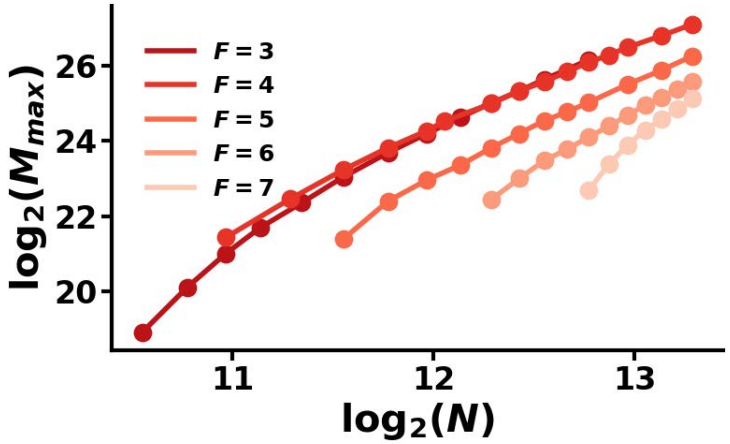
We carefully measured the operational capacity of Resonator Networks in search of a relationship between M_{max} and N. We focused on Resonator Networks with outer product weights–for $N \approx 5000$ and larger, randomly-chosen codevectors are nearly orthogonal and capacity is approximately the same for OLS weights. We reiterate that operational capacity is specific to parameters p and k: p is the threshold for total accuracy and k is the maximum number of iterations the algorithm is allowed to take (refer to Definition 2.5.2). Here we report operational capacity for $p = 0.99$ and $k = 0.001M$ on randomly-sampled codevectors. The operational capacity is specific to these choices, which are practical for Vector Symbolic Architectures.

Our simulations revealed that, empirically, **Resonator Network operational capacity $\mathbf{M_{max}}$ scales as a quadratic function of N**, which we illustrate in Figure 2.4. The points in this figure are estimated from many thousands of random trials, over a range of values for F and N. In part (a) we show operational capacity separately for each F from 2 to 7, with the drawn curves indicating the least-squares quadratic fit to the measured points. In part (b) we put these points on the same plot, following a logarithmic transformation to each axis, in order to illustrate that capacity also varies as a function of F. Appendix A.2 provides some additional commentary on this topic, including some speculation on a scaling law that combines F and N. The parameters of this particular combined scaling are estimated from simulation and not derived analytically–therefore they may deserve additional scrutiny and we do not focus on them here. The main message of this section is that capacity scales quadratically in N, regardless of how many factors are used.

The curves in Figure 2.4 are constructive in the following sense: given a fixed N, they indicate the largest factorization problem that can be solved reliably. Conversely, and this is often the case in VSAs, the problem size M is predetermined, while N is variable–in this case we know how large one must make N. We include in the official software implementation



(a) M_{max} scales quadratically in N. Red points are measured from simulation; black curves are the least-squares quadratic fits. Parameters of these fits included in Appendix A.2.



(b) M_{max} varies as a function of both F and N. Over the measured range for N, capacity is highest for $F = 3$ and $F = 4$. Data for $F = 2$ was omitted to better convey the trend for F=3 and higher, but see Appendix A.2 for the full picture.

Figure 2.4: Operational capacity of Resonator Networks with OP weights.

that accompanies this work[3] a text file with all of the measured operational capacities.

Quadratic scaling means that one can aspire to solve very large factorization problems, so long as he or she can build a Resonator Network with big enough N. We attempted to estimate capacity for even larger values of N than we report in Figure 2.4, but this was beyond the capability of our current computational resources. A useful contribution of follow-on work would be to leverage high-performance computing to measure some of these values. Applications of Vector Symbolic Architectures typically use $N \leq 10{,}000$, but there are other reasons one might attempt to push Resonator Networks further. Early work on Hopfield Networks proposed a technique for storing solutions to the Travelling Salesman Problem as fixed points of the model's dynamics (Hopfield & Tank, 1985), and this became part of a larger approach using nonlinear dynamical systems to solve hard search problems. We do not claim that any particular search problem, other than the factorization we have defined (2.1), can be solved by Resonator Networks. Supposing, however, that some other hard problem can be cast in the form of (2.1), the quadratic scaling of operational capacity makes this a potentially power tool.

Capacity is highest when the codebooks $\mathbb{X}_f$ each have the same number of codevectors ($D_1 = D_2 = \ldots = D_F = \sqrt[F]{M}$), and this was the case for the operational capacity results we have shown so far. We chose this in order to have a simple standard for comparison among the different algorithms, but in general it is possible that the codebooks are unbalanced, so that we have the same $M = \prod_f D_f$ but $D_1 \neq D_2 \neq \ldots \neq D_f$. In this case, capacity is lower than for balanced codebooks. We found that the most meaningful way to measure the degree of balance between codebooks was by the ratio of the smallest codebook to the largest codebook:

$$\xi := \left(\min_f D_f\right) / \left(\max_f D_f\right) \tag{2.21}$$

For $\xi \geq 0.2$ we found that the effect on M_{max} was simply an additive factor which can be absorbed into a (slightly smaller) y-intercept a for the quadratic fit. For extreme values of ξ, where there is one codebook that is for instance 10 or 20 times larger than another, then all three parameters a, b, and c are affected, sometimes significantly. Scaling is still quadratic, but the actual capacity values may be significantly reduced.

Our result–measured operational capacity which indicates an approximately quadratic relationship between M_{max} and N–is an important characterization of Resonator Networks. It suggests that our framework scales to very large factorization problems and serves as a guideline for implementation. Our attempts to analytically derive this result were stymied by the toolbox of nonlinear dynamical systems theory. Operational capacity involves the probability that this system, when initialized to an effectively random state, converges to a particular set of fixed points. No results from the study of nonlinear dynamical systems, that we are aware of, allow us to derive such a strong statement about Resonator Networks. Still, the scaling of Figure 2.4 is fairly suggestive of some underlying law, and we are hopeful that a theoretical explanation exists, waiting to be discovered.

[3] https://github.com/spencerkent/resonator-networks

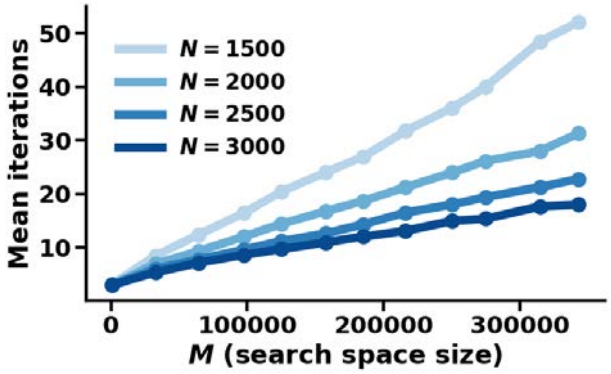


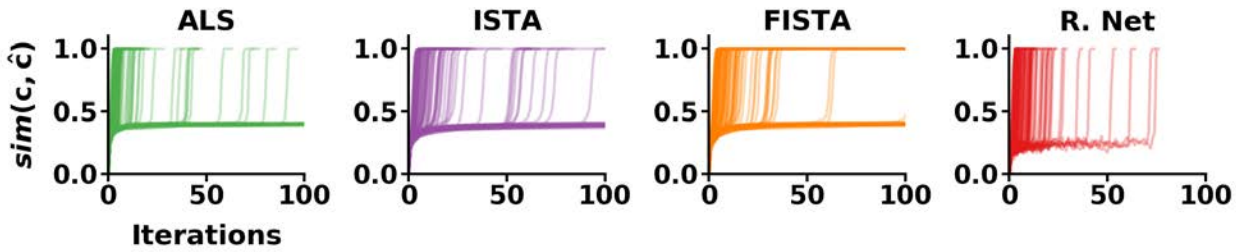
Figure 2.5: Iterations until convergence, Resonator Network with outer product weights and $F = 3$. The number of iterations is a very small compared to the size of the search space

2.5.3 Search speed

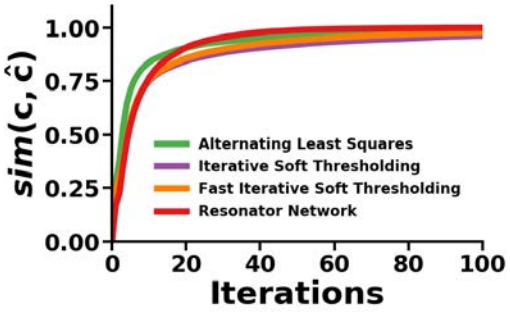
If a Resonator Network is not consistently descending an energy function, is it just aimlessly wandering around the space, trying every possible factorization until it finds the correct one? Figure 2.5 shows that it is not. We plot the mean number of iterations over 5,000 random trials, as a fraction of M, the search space size. This particular plot is based on a Resonator Network with outer product weights and $F = 3$. In the high-performance regime where M is below operation capacity, the number of iterations is far less than the $0.001M$ cutoff we used in the simulations of Section 2.5.2–the algorithm is only ever considering a tiny fraction of the possible factorizations before it finds the solution.

Section 2.5.2.1 compared the operational capacity of different algorithms and showed that compared to the benchmarks, Resonator Networks can solve much larger factorization problems. This is in the sense that the dynamics eventually converge (with high probability) on the correct factorization while, the dynamics of the other algorithms converge on spurious factorizations. This result, however, does not directly demonstrate the relative speed with which factorization are found in terms of either the number of iterations or the amount of time to convergence. We set up a benchmark to determine the relative speed of Resonator Networks and our main finding is depicted in Figure 2.6.

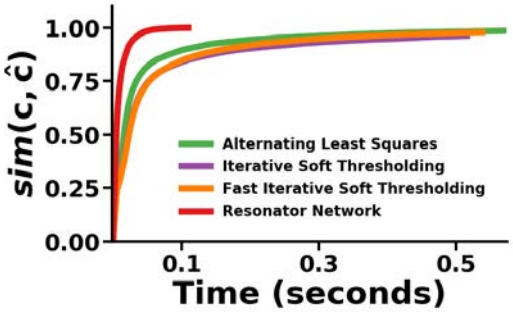
Measured in number of iterations, Resonator Networks are comparable to the benchmark algorithms. We noted that Alternating Least Squares is the most greedy of the benchmarks, and one can see from Figure 2.6 that it is the fastest in this sense. We are considering only trials that ultimately found the correct factorization, which in this simulation was roughly 70% for each of the benchmarks. In contrast, Resonator Networks always eventually found the correct factorization. **Measured in terms of wall-clock time, Resonator Networks are significantly faster than the benchmarks.** This can be attributed to their nearly 5× lower per-iteration cost. Resonator Networks with outer product weights utilize very simple arithmetic operations and this explains the difference between Figures 2.6b and 2.6c.



(a) Convergence traces for 100 randomly-drawn factorization problems–each line is the cosine similarity between $\mathbf{c}$ and $\hat{\mathbf{c}}$ over iterations of the algorithm. Each of the four algorithms is run on *the same* 100 factorization problems. All of the instances are solved by the Resonator Network, whereas a sizeable fraction (around 30%) of the instances are not solved by the benchmark algorithms, at least within 100 iterations.



(b) Avg. cosine similarity vs. iteration number (only trials with accuracy 1.0)



(c) Avg. cosine similarity vs. wall-clock time (only trials with accuracy 1.0)

Figure 2.6: Our benchmark of factorization speed. Implementation in Python with NumPy. Run on machine with Intel Core i7-6850k processor and 32GB RAM. We generated $5,000$ random instantiations of the factorization problem with $N = 1500$, $F = 3$, and $D_f = 40$, running each of the four algorithms in turn. Figure 2.6a gives a snapshot of 100 randomly selected trials. Figures 2.6b and 2.6c show average performance *conditioned on the algorithms finding the correct factorization.*

2.5.4 Dynamics that do not converge

One must be prepared for the possibility that the dynamics of a Resonator Network will not converge. Fortunately, for M below the $p = 0.99$ operational capacity, these will be exceedingly rare. From simulation, we identified three major regimes of different convergence behavior, which are depicted in Figure 2.7:

- For M small enough, almost all trajectories converge, and moreover they converge to a state that yields the correct factorization. Limit cycles are possible but rare, and often still yield the correct factorization. There appear to be few if any spurious

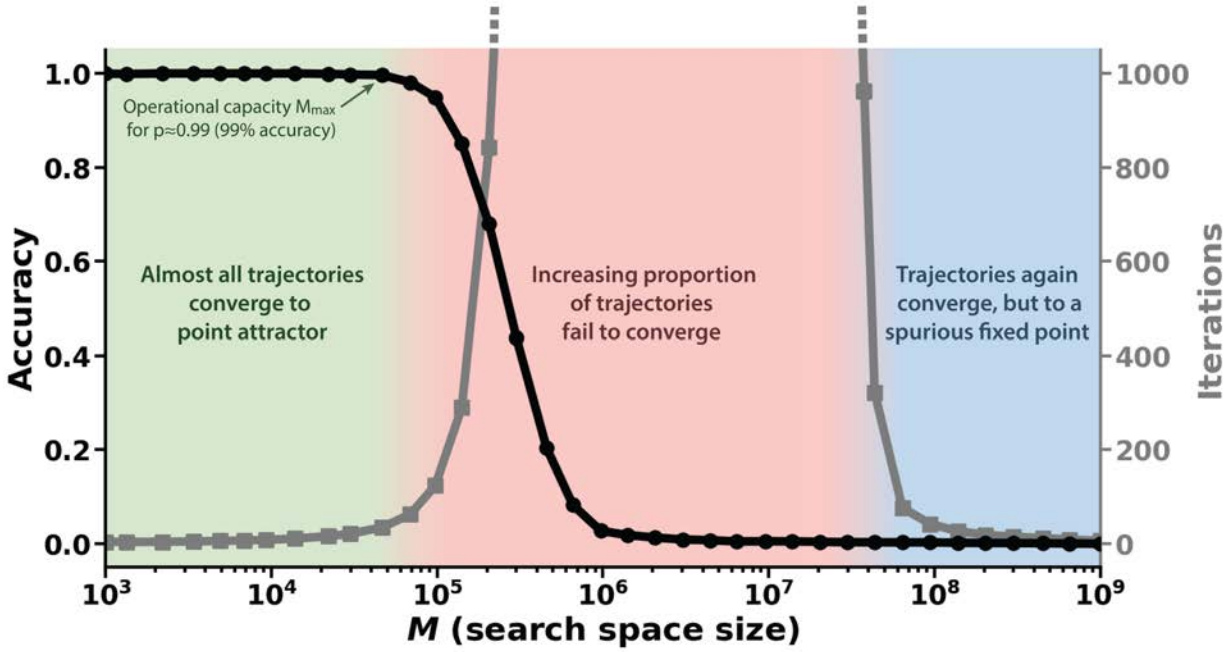


Figure 2.7: Regimes of different convergence behavior. Curves show measurement from simulation of an outer product Resonator Network with 3 factors and $N = 400$. This is also meant as a diagram of convergence behavior for Resonator Networks in general. Shown in black is the average decoding accuracy and shown in gray is the median number of iterations taken by the network. For low enough M, the network always finds a fixed point yielding 100% accuracy. The network will not converge to spurious fixed points in this regime (green). As M is increased, more trajectories wander, not converging in any reasonable time (red). Those that are forcibly terminated yield incorrect factorizations. For large enough M, the network is completely saturated and most states are fixed points, regardless of whether they yield the correct factorization (blue). Resonator Networks with OLS weights are always stable when $D_f = N$, but OP weights give a bitflip probability that is zero only asymptotically in M (see Section 2.5.1 and Appendix A.10).

fixed points (those yielding an incorrect factorization)–if the trajectory converges to a point attractor or limit cycle, one can be confident this state indicates the correct factorization.

- As M increases, non-converging trajectories appear in greater proportion and yield incorrect factorizations. Any trajectories which converge on their own continue to yield the correct factorization, but these become less common.
- Beyond some saturation value M_{sat} (roughly depicted as the transition from red to blue in the figure), both limit cycles and point attractors re-emerge, and they yield the incorrect factorization.

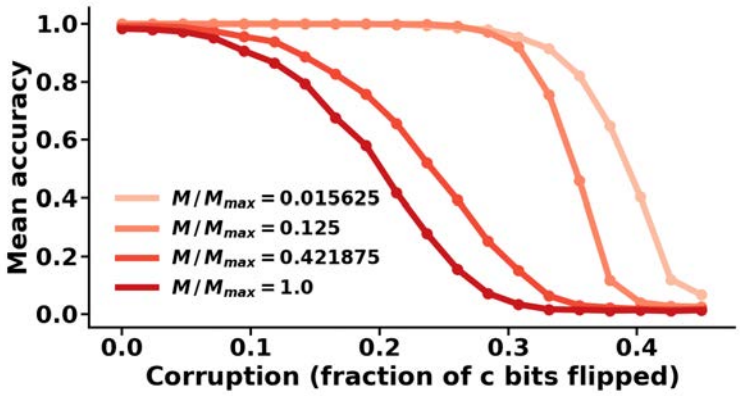


Figure 2.8: Factoring a corrupted $\mathbf{c}$. For M well below capacity (lighter curves above) one can sustain heavy corruption to $\mathbf{c}$ and still find the correct factorization.

In theory, limit cycles of any length may appear, although in practice they tend to be skewed towards small cycle lengths. Networks with two factors are the most likely to find limit cycles, and this likelihood appears to decrease with increasing numbers of factors. Our intuition about what happens in the middle section of Figure 2.7 is that the basins of attraction become very narrow and hard to find for the Resonator Network dynamics. The algorithm will wander, since it has so few spurious fixed points (see Section 2.5.6), but not be able to find any basin of attraction.

2.5.5 Factoring a 'noisy' composite vector

Our assumption has been that one combination of codevectors from our codebooks $\mathbb{X}_f$ generates $\mathbf{c}$ exactly. What if this is not the case? Perhaps the vector we are given for factorization has had some proportion ζ of its components flipped, that is, we are given $\tilde{\mathbf{c}}$ where $\tilde{\mathbf{c}}$ differs from $\mathbf{c}$ in exactly $\lfloor \zeta N \rfloor$ places. The vector $\mathbf{c}$ has a factorization based on our codebooks but $\tilde{\mathbf{c}}$ does not. We should hope that a Resonator Network will return the factors of $\mathbf{c}$ so long as the corruption is not too severe. This is an especially important capability in the context of Vector Symbolic Architectures, where $\tilde{\mathbf{c}}$ will often be the result of some algebraic manipulations that generate noise and corrupt the original $\mathbf{c}$ to some degree. We show in Figure 2.8 that a Resonator Network can still produce the correct factorization even after a significant number of bits have been flipped. This robustness is more pronounced when the number of factorizations is well below operational capacity, at which point the model can often still recover the correct factorization even when 30% of the bits have been flipped.

2.5.6 A theory for differences in operational capacity

The failure mode of each benchmark algorithm is getting stuck at a *spurious fixed point* of the dynamics. This section develops a simple comparison between the spurious fixed

points of Resonator Networks and the benchmarks as an explanation for why Resonator Networks enjoy relatively higher operational capacity. From among the benchmarks we focus on Projected Gradient Descent (applied to the negative inner product with the simplex constraint) to illustrate this point. We will show that the correct factorization is always stable under Projected Gradient Descent (as it is with the OLS variant of Resonator Networks), but that incorrect factorizations are much more likely to be fixed points under Projected Gradient Descent. The definition of Projected Gradient Descent can be found in Table A.2, with some comments in Appendix A.7.

Stability of the correct factorization

The vector of coefficients $\mathbf{a}_f$ is a fixed point of Projected Gradient Descent dynamics when the gradient at this point is exactly $\mathbf{0}$ or when it is in the nullspace of the projection operator. We write

$$\mathcal{N}\big(\mathcal{P}_{C_f}[\mathbf{x}]\big) := \{\mathbf{z} \mid \mathcal{P}_{C_f}\big[\mathbf{x}+\mathbf{z}\big] = \mathcal{P}_{C_f}\big[\mathbf{x}\big]\} \tag{2.22}$$

to denote this set of points. The nullspace of the projection operator is relatively small on the faces and edges of the simplex, but it becomes somewhat large at the vertices. We denote a vertex by $\mathbf{e}_i$ (where $(\mathbf{e}_i)_j = 1$ if $j = i$ and 0 otherwise). The nullspace of the projection operator at a vertex of the simplex is an intersection of halfspaces (each halfspace given by an edge of the simplex). We can compactly represent it with the following expression:

$$\mathcal{N}\big(\mathcal{P}_{\Delta_{D_f}}[\mathbf{e}_i]\big) = \big\{\mathbf{z} \mid \bigcap_{j\neq i} (\mathbf{e}_i - \mathbf{e}_j)^\top \mathbf{z} \geq 1\big\} \tag{2.23}$$

An equivalent way to express the nullspace is

$$\mathcal{N}\big(\mathcal{P}_{\Delta_{D_f}}[\mathbf{e}_i]\big) = \big\{\mathbf{z} \mid z_j \leq z_i - 1 \ \ \forall j \neq i\big\} \tag{2.24}$$

In other words, for a vector to be in the nullspace at $\mathbf{e}_i$, the ith element of the vector must be the largest by a margin of 1 or more. This condition is met for the vector $-\nabla_{\mathbf{a}_f}\mathcal{L}$ at the correct factorization, since $-\nabla_{\mathbf{a}_f}\mathcal{L} = \mathbf{X}_f^\top\big(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\big) = \mathbf{X}_f^\top \mathbf{x}_\star^{(f)}$. This vector has a value N for the component corresponding to $\mathbf{x}_\star^{(f)}$ and values that are $\leq N-1$ for all the other components. Thus, the correct factorization (the solution to (2.1) and global minimizer of (2.9)) is always a fixed point under the dynamics of Projected Gradient Descent (PGD).

This matches the stability of OLS Resonator Networks which are, by construction, always stable at the correct factorization. We showed in Section 2.5.1 that OP weights induce instability and that percolated noise makes the model marginally less stable than Hopfield Networks, but there is still a large range of factorization problem sizes where the network is stable with overwhelming probability. What distinguishes the benchmarks from Resonator Networks is what we cover next, the stability of *incorrect* factorizations.

Stability of incorrect factorizations

Suppose initialization is done with a random combination of codevectors that do not produce $\mathbf{c}$. The vector $\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}$ will be a *completely random bipolar vector*. So long as D_f is significantly smaller than N, which it always is in our applications, $\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}$ will be nearly orthogonal to every vector in $\mathbb{X}_f$ and its projection onto $\mathcal{R}(\mathbf{X}_f)$ will be small, with each component equally likely to be positive or negative. Therefore, under the dynamics of a Resonator Network with OLS weights, each component will flip its sign compared to the initial state with probability 1/2, and the state for this factor will remain unchanged with the minuscule probability $1/2^N$. The total probability of this incorrect factorization being stable, accounting for each factor, is therefore $(1/2^N)^F$. Suboptimal factorizations are *very* unlikely to be a fixed points. The same is true for a Resonator Network with OP weights because each element of the vector $\mathbf{X}_f\mathbf{X}_f^\top\left(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\right)$ is approximately Gaussian with mean zero (see Section 2.5.1 and Appendix A.10).

Contrast this against Projected Gradient Descent. We recall from (2.24) that the requirement for $\mathbf{e}_i$ to be a fixed point is that the ith component of the gradient at this point be largest by a margin of 1 or more. *This is a much looser stability condition than we had for Resonator Networks*–such a scenario will actually occur with probability $1/D_f$ for each factor, and the total probability is $1/M$. While still a relatively small probability, in typical VSA settings $1/M$ is much larger than $(1/2^N)^F$, meaning that compared to Resonator Networks, Projected Gradient Descent is much more stable at incorrect factorizations. Empirically, the failure mode of Projected Gradient Descent involves it settling on one of these spurious fixed points.

Stability in general

The cases of correct and incorrect factorizations drawn from the codebooks are two extremes along a continuum of possible states the algorithm can be in. For Projected Gradient Descent any state will be stable with probability in the interval $[\frac{1}{M}, 1]$, while for Resonator Networks (with OLS weights) the interval is $[\frac{1}{2^{FN}}, 1]$. In practical settings for VSAs, the interval $[\frac{1}{2^{FN}}, 1]$ is, in a relative sense, much larger than $[\frac{1}{M}, 1]$. Vectors drawn uniformly from either $\{-1, -1\}^N$ or $[-1, -1]^N$ concentrate near the lower end of these intervals, suggesting that on average, **Projected Gradient Descent has many more spurious fixed points**.

This statement is not fully complete in the sense that dynamics steer the state along specific trajectories, visiting states in a potentially non-uniform way, but it does suggest that Projected Gradient Descent is much more susceptible to spurious fixed points. The next section shows that these trajectories do in fact converge on spurious fixed points as the factorization problem size grows.

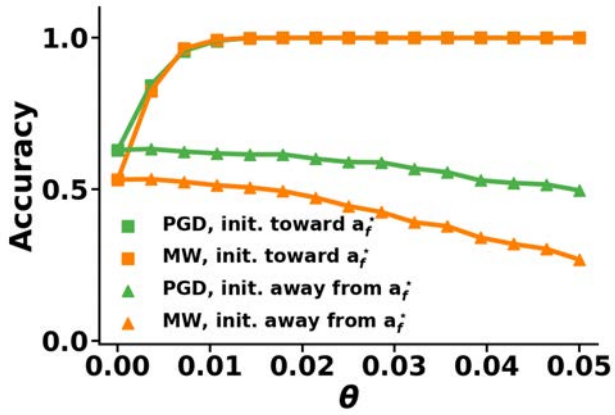


Figure 2.9: States in hypercube interior get pulled into spurious basins of attraction. Projected Gradient Descent is in green and Multiplicative Weights is in orange. Network is initialized at a distance θ from the center of the simplex (see equation (2.25)), and allowed to converge. The y-axis is the accuracy of the factorization implied by the converged state. Triangles indicate initialization slightly away from $\mathbf{a}_f^\star$ toward any of the other simplex vertices, which is most directions in the space. These initial states get quickly pulled into a spurious basin of attraction.

Basins of attraction for benchmark algorithms

It may be that while there are sizable basins of attraction around the correct factorization, moving through the interior of the hypercube causes state trajectories to fall into the basin corresponding to a spurious fixed point. In a normal setting for several of the optimization-based approaches, we initialize $\mathbf{a}_f$ to be at the center of the simplex, indicating that each of the factorizations is equally likely. Suppose we were to initialize $\mathbf{a}_f$ so that it is just slightly nudged toward one of the simplex vertices. We might nudge it toward the correct vertex (the one given by $\mathbf{a}_f^\star$) or we might nudge it toward any of the other vertices, away from $\mathbf{a}_f^\star$. We can parameterize this with a single scalar θ and $\mathbf{e}_i$ chosen uniformly among the possible vertices:

$$\mathbf{a}_f[0] = \theta \mathbf{e}_i + (1-\theta)\frac{1}{D_f}\mathbf{1} \quad | \quad \theta \in [0,1]\,,\ i \sim \mathcal{U}\{1, D_f\} \tag{2.25}$$

We ran a simulation with $N = 1500$ and $D_1 = D_2 = D_3 = 50$, at which Projected Gradient Descent and Multiplicative Weights have a total accuracy of 0.625 and 0.525, respectively. We created 5,000 random factorization problems, initializing the state according to (2.25) and allowing the dynamics to run until convergence. We did this first with a nudge toward the correct factorization $\mathbf{a}_f^\star$ (squares in Figure 2.9) and then with a nudge away from $\mathbf{a}_f^\star$, toward a randomly-chosen spurious factorization (triangles in Figure 2.9).

What Figure 2.9 shows is that by moving just a small distance toward the correct vertex, we very quickly fall into its basin of attraction. However, moving toward any of the other vertices is actually somewhat likely to take us into a spurious basin of attraction (where the

converged state is decoded into an incorrect factorization). The space is *full* of these bad directions. It would be very lucky indeed to start from the center of the simplex and move immediately toward the solution–it is far more likely that initial updates take us somewhere else in the space, toward one of the other vertices, and this plot shows that these trajectories often get pulled towards a spurious fixed point. What we are demonstrating here is that empirically, the interior of the hypercube is somewhat treacherous from an optimization perspective, and this lies at the heart of why the benchmark algorithms fail.

From among the benchmarks, we restricted our analysis of spurious fixed points to Projected Gradient Descent and, in Figure 2.9, Multiplicative Weights. This choice was made for clarity, and similar arguments apply for all of the benchmarks. While the details may differ slightly (for instance, spurious fixed points of ALS appear near the simplex center, not at a vertex), the failure mode of the benchmarks is strikingly consistent. They all become overwhelmed by spurious fixed points, long before this affect is felt by Resonator Networks. We have shown that **in expectation, Projected Gradient Descent has many more spurious fixed points than Resonator Networks**. We have also show that **trajectories moving through the interior of the hypercube are easily pulled into these spurious basins of attraction.**

2.6 Discussion

We studied a vector factorization problem which arises in the use of Vector Symbolic Architectures (as introduced in Chapter 1), showing that Resonator Networks solve this problem remarkably well. Their performance comes from a particular form of nonlinear dynamics, coupled with the idea of searching in superposition. Solutions to the factorization problem lie in a small sliver of $\mathbb{R}^N$–i.e., the corners of the bipolar hypercube $\{-1, 1\}^N$–and the highly nonlinear activation function of Resonator Networks serves to constrain the search to this subspace. We drew connections between Resonator Networks and a number of benchmark algorithms which cast factorization as a problem of *optimization*. This intuitively-satisfying formulation appears to come at a steep cost. None of the benchmarks were competitive with Resonator Networks in terms of key metrics that characterize factorization performance. One explanation for this is that the benchmarks have comparatively many more spurious fixed points of their dynamics, and that the loss function landscape in the interior of the hypercube induces trajectories that approach these spurious fixed points.

Unlike the benchmarks, Resonator Networks do not have a global convergence guarantee, and in some respects we see this as beneficial characteristic of the model. Requiring global convergence appears to unnecessarily constrain the search for factorizations, leading to lower capacity. Besides, operational capacity (defined in this chapter) specifies a regime where the lack of a convergence guarantee can be practically ignored. Resonator Networks almost always converge in this setting, and the fixed points yield the correct solution. The benchmarks are, by steadfastly descending a loss function, in some sense more greedy than Resonator Networks. It appears that Resonator Networks strike a more natural balance

between 1) making updates based on the best-available local information and 2) still exploring the solution space while not getting stuck. Our approach follows a kind of "Goldilocks principle" on this trade off–not too much, not too little, but just right.

We are not the first to consider eschewing convergence guarantees to better solve hard search problems. For instance, randomized search algorithms utilize some explicit form of randomness to find better solutions, typically only converging if this randomness is reduced over time (Spall, 2005). In contrast, our model is completely deterministic, and the searching behavior comes from nonlinear heteroassociative dynamics. Another example is the proposal to add small amounts of random asymmetry to the (symmetric) weight matrix of Hopfield Networks (Hertz et al., 1986). This modification removes the guaranteed absence of cyclic and chaotic trajectories that holds for the traditional Hopfield model. But, at the same time, and without significantly harming the attraction of memory states, adding asymmetry to the weights can improve associative memory recall by shrinking the basins of attraction associated with spurious fixed points (Singh et al., 1995; Chengxiang et al., 2000).

We emphasize that, while Resonator Networks appear to be better than alternatives for the particular vector factorization problem (2.1), this is not a claim they are appropriate for other hard search problems. Rather, Resonator Networks are specifically designed for the vector factorization problem at hand. There exist several prior works involving some aspect of factorization that we would like to mention here, but we emphasize that each one of them deals with a problem or approach that is distinct from what we have introduced in this chapter.

Tensor decomposition is a family of problems that bear some resemblance to the factorization problem we have introduced (2.1). Key differences include the object to be factored, which is a higher-order tensor, not a vector, and constraints on the allowable factors. We explain in Appendix A.4 how our factorization problem is different from traditional tensor decompositions. Our benchmarks actually included the standard tensor decomposition algorithm, Alternating Least Squares, re-expressed for (2.1), and we found that it is not well-matched for this factorization problem. Bidirectional Associative Memory, proposed by Kosko (1988), is an extension of Hopfield Networks that stores *pairs* of factors in a matrix using the outer product learning rule. The composite object is a matrix, rather than a vector, and is much closer to a particular type of tensor decomposition called the 'CP decomposition', which we elaborate on in Appendix A.4. Besides the fact that this model applies only to *two* factor problems, its dynamics are different from ours and its capacity is relatively low (Kobayashi et al., 2002). Subsequent efforts to extend this model to factorizations with 3 or more factors (Huang & Hagiwara, 1999; Kobayashi et al., 2002) have had very limited success and still rely on matrices that connect pairs of factors rather than a single multilinear product, which we have in our model. Bilinear Models of Style and Content (Tenenbaum & Freeman, 2000) was an inspiration for us in deciding to work on factorization problems. This paper applies a different type of tensor decomposition, a 'Tucker decomposition' (again see Appendix A.4), to a variety of different real-valued datasets using what appears to be in one case a closed-form solution based on the Singular Value Decomposition, and in the other case a variant of Alternating Least Squares. In that sense, their method is different

from ours, the factorization problem is itself different, and they consider only pairs of factors. Memisevic and Hinton (2010) revisits the Tucker decomposition problem, but factors the 'core' tensor representing interactions between factors in order to make estimation more tractable. They propose a Boltzmann Machine that computes the factorization and show some results on modeling image transformations. Finally, there is a large body of work on matrix factorization of the form $\mathbf{V} \approx \mathbf{WH}$, the most well-known of which is probably Non-negative Matrix Factorization (Lee & Seung, 2001). The matrix $\mathbf{V}$ can be thought of a sum of outer products, so this is really a type of CP decomposition with an additional constraint on the sign of the factors. Different still is the fact that $\mathbf{W}$ is often interpreted as a basis for the columns of $\mathbf{V}$, with $\mathbf{H}$ containing the coefficients of each column with respect to this basis. In this sense, vectors are being *added* to explain $\mathbf{V}$, rather than combined multiplicatively–Non-negative Matrix Factorization is much closer to Sparse Coding (Hoyer, 2004).

Chapter 3 illustrates how distributed representations of data structures can be built with the algebra of Vector Symbolic Architectures, as well as how Resonator Networks can decompose these data structures. VSAs are a powerful way to think about structured connectionist representations, and Resonator Networks make the framework much more scalable. Extending the examples found in Chapter 3 to more realistic data (for example, complex 3-dimensional visual scenes) could be a useful application of Resonator Networks. This will likely require learning a transform from pixels into the space of high-dimensional symbolic vectors, and this learning should ideally occur in the context of the factorization dynamics–we feel that this is an exciting avenue for future study but something we have not pursued ourselves. Here we have not shown Resonator Circuits for anything other than bipolar vectors. However, a version of the model wherein vector elements are unit-magnitude complex phasors is a natural next extension, and relevant to Holographic Reduced Representations (Plate, 2003). A recent theory of sparse phasor associative memories (Frady & Sommer, 2019) may allow one to perform this factorization with a network of spiking neurons.

Resonator Networks are an abstract neural model of factorization. We believe that as the theory and applications of Resonator Networks are further developed, they may help us understand factorization in the brain, which still remains an important mystery.

Chapter 3

Applications of VSAs and Resonator Networks

So far, we have argued that:

- Multiplicative codes arise throughout perception and cognition, going hand-in-hand with the need to factor these codes efficiently.
- Vector Symbolic Architectures provide a framework for modeling cognitive data structures and they present a clear factorization problem.
- Resonator Networks are an effective way of solving this factorization problem.

We now present applications of this argument. Our main emphasis will be on how to represent and manipulate visual scenes, but first we revisit the tree search problem first introduced in Chapter 1.

3.1 Tree search

Shown in Figure 3.1 is a binary tree that admits VSA representation with the vector $\mathbf{t}$:

$$\begin{aligned}
\mathbf{t} \;=\; & \mathbf{a} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{left}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \\
& \oplus \mathbf{b} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \\
& \oplus \mathbf{c} \otimes \mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \\
& \oplus \mathbf{d} \otimes \mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{right}}) \otimes \rho^3(\mathbf{q}_{\texttt{left}}) \\
& \oplus \mathbf{e} \otimes \mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{right}}) \otimes \rho^3(\mathbf{q}_{\texttt{right}}) \\
& \oplus \mathbf{f} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{right}}) \otimes \rho^3(\mathbf{q}_{\texttt{left}}) \otimes \rho^4(\mathbf{q}_{\texttt{left}}) \\
& \oplus \mathbf{g} \otimes \mathbf{q}_{\texttt{left}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{right}}) \otimes \rho^3(\mathbf{q}_{\texttt{left}}) \otimes \rho^4(\mathbf{q}_{\texttt{right}})
\end{aligned} \tag{3.1}$$

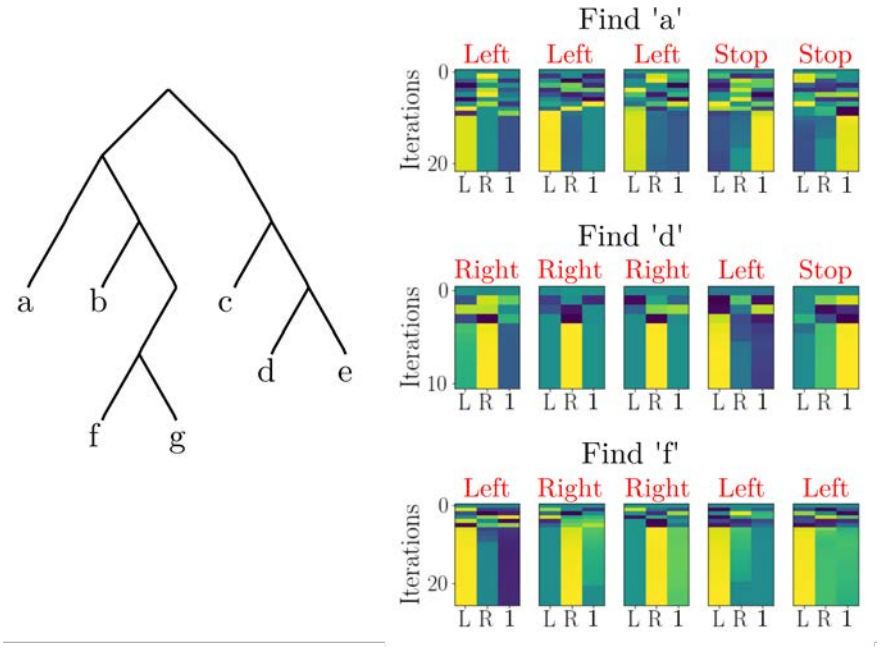


Figure 3.1: Tree search with a Resonator Network. The query of the vector $\mathbf{t}$ produces an encoding of position which the Resonator Network can factor. The colored plots indicate the time evolution of $\hat{\mathbf{x}}^{(0)} \dots \hat{\mathbf{x}}^{(4)}$ (from left to right), showing the cosine similarity of each estimate to each of the three possible vectors $\rho^d(\mathbf{q}_{\texttt{left}}), \rho^d(\mathbf{q}_{\texttt{right}}), \mathbf{1}$. Purple indicates low similarity, and yellow indicates high similarity. Initially the similarity changes significantly, until the three estimators find a coherent factorization and quickly converge. Red letters indicate the converged result for each $\hat{\mathbf{x}}^{(0)} \dots \hat{\mathbf{x}}^{(4)}$.

The factorization problem arises when we try to decode the position of leaf `c` (or any other leaf) in the tree:

$$\mathbf{c}^{-1} \otimes \mathbf{t} = \mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \oplus noise \tag{3.2}$$

We have left our notation in the generic VSA operations $\oplus$, $\otimes$, and $\rho(\cdot)$ to communicate that our approach works for any choice of architecture. Do note, however, that the rigorous results we have shown in Chapter 2 hold specifically for the Multiply, Add, Permute architecture (Gayler, 1998, 2004) and the simulation we show in Figure 3.1 was done with these bipolar vectors.

To set up a Resonator Network for this problem, we first establish a maximum depth to search through–the maximum depth determines the number of factors that need to be estimated. For the tree shown in Figure 3.1, we need five estimators, because this is the depth of the deepest leaves, `f` and `g`.

Each factor estimate will determine whether to go `left`, `right` or to `stop`, for each level down the tree. To indicate `stop` a special vector is used, the multiplicative identity vector, which in the MAP VSA is $\mathbf{1}$ (a vector of all ones). By using the appropriate number of

these identity vectors, each location in the tree can be thought of as a composition with the same depth (the maximum depth), even if the location is only partially down the tree. For instance, if we consider leaf c in the Figure 3.1, then its position $\mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}})$ is also $\mathbf{q}_{\texttt{right}} \otimes \rho(\mathbf{q}_{\texttt{right}}) \otimes \rho^2(\mathbf{q}_{\texttt{left}}) \otimes \mathbf{1} \otimes \mathbf{1}$. This way, we can set up a Resonator Network for five factors and have it decode locations anywhere in the tree.

Following our established convention, we denote each factor estimate in the Resonator Network as $\hat{\mathbf{x}}^{(0)}, \hat{\mathbf{x}}^{(1)}, \hat{\mathbf{x}}^{(2)}, \hat{\mathbf{x}}^{(3)}, \hat{\mathbf{x}}^{(4)}$ and the codebook matrices as $\mathbf{X}_0, \mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \mathbf{X}_4$. Each codebook matrix contains permuted versions of $\mathbf{q}_{\texttt{left}}$ and $\mathbf{q}_{\texttt{right}}$, and $\mathbf{1}$:

$$\mathbf{X}_d = \left[\rho^d(\mathbf{q}_{\texttt{left}}), \rho^d(\mathbf{q}_{\texttt{right}}), \mathbf{1}\right]$$

where d indicates the depth in the tree. The input to the network is $\mathbf{c}^{-1} \otimes \mathbf{t}$, which we expect to be a composite vector plus noise (refer to eq. (3.2) and also see Section 2.5.5)

The process is demonstrated in Figure 3.1. Different leaves in the tree can be found by unbinding the leaf representation from the tree vector and using this result as the input to the Resonator Network. We visualize the network dynamics by displaying the similarity of each factor estimate $\hat{\mathbf{x}}^{(d)}[t]$ to the atoms stored in its corresponding codebook matrix $\mathbf{X}_d$. The evolution of these similarity weights over time is shown as a heat map (Figure 3.1, right). The heat maps show that the system initially jumps around chaotically, with the weighting of each estimate changing drastically each iteration. But then there is a quite sudden transition to a stable equilibrium, where each estimate converges nearly simultaneously, and at this point the output for each factor is essentially the codebook element with highest weight.

This toy example of tree search asks the Resonator Network to search among $3^5 = 243$ possible locations in the tree. Clearly, it does so in a much more intelligent way than brute force. The point is *not* that VSAs and Resonator Networks are the most efficient way to represent or search a tree compared to non-connectionist approaches from computer science. However, if one wanted to do so in a connectionist framework, this would seem a good way to go.

3.2 Visual scene analysis

This section will outline some of our attempts to encode images with VSA representations that capture certain generic structures found in visual scenes. Most of the past work on VSAs has focused on more fundamentally abstract and discrete domains (e.g. language). We are also more concerned with *learning* transformations from raw signals (such as images) into an abstract space of symbolic vectors. We will utilize several different VSAs in this section, so we will try to make our notation specific in each case: $\odot$ is the Hadamard product, $\circledast$ is circular convolution, and $+$ is pointwise addition.

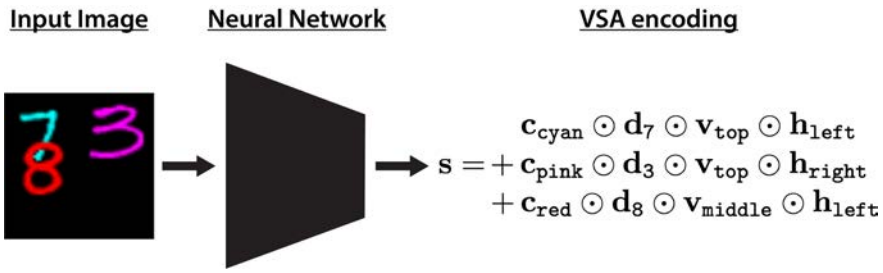


Figure 3.2: Generating a vector symbolic encoding of a visual scene

3.2.1 Black box learning for simple scenes

Consider the scene in Figure 3.2 containing colored MNIST digits (LeCun, 1998) in different positions. Position in the scene is indexed by vertical and horizontal coordinates, each quantized into three possible values, (`top`, `middle`, `bottom`) and (`left`, `center`, `right`), respectively. Each digit can take on one of seven possible colors (`blue`, `green`, `cyan`, `red`, `pink`, `yellow`, `white`). The digits are labelled by their semantic class (`0`, `1`, ..., `9`), but the exact shape will differ, as the stimuli are sampled from the 50,000 exemplars in the MNIST training set.

Any given scene can have between one and three of these objects, which are allowed to partially occlude one another. We generate symbolic vectors $\mathbf{c}_{\texttt{blue}}, \mathbf{c}_{\texttt{green}}, \ldots, \mathbf{c}_{\texttt{white}}$ to encode color, $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_9$ to encode shape, $\mathbf{v}_{\texttt{top}}$, $\mathbf{v}_{\texttt{middle}}$, $\mathbf{v}_{\texttt{bottom}}$ to encode vertical position, and $\mathbf{h}_{\texttt{left}}, \mathbf{h}_{\texttt{center}}, \mathbf{h}_{\texttt{right}}$ to encode horizontal position, which are stored in the codebook matrices $\mathbf{C}$, $\mathbf{D}$, $\mathbf{V}$, and $\mathbf{H}$.

The example scene in Figure 3.2 contains a `cyan 7` at position `top`, `left`, a `pink 3` at position `top`, `right`, and a `red 8` at position `middle`, `left`. While this is a highly simplified type of visual scene, it illustrates the combinatorial challenge of representing and interpreting visual scenes. There are only 23 distinct atomic parameters (10 for digit identity, 7 for color, 3 each for vertical and horizontal position) and yet these combine to describe $10 \times 7 \times 3 \times 3 = 630$ individual objects, and $630 + 630^2 + 630^3 = 250{,}444{,}530$ possible scenes with 1, 2, or 3 objects. This number of combinations still does not include the variability among exemplars for each shape, of which there are $50,000$ in the MNIST training dataset.

To encode this scene, one *could* generate role vectors for color, shape, vertical position, and horizontal position. The idea would then be to form role-filler bindings to encode an object:

$$\{(\texttt{color}, \texttt{red}), (\texttt{shape}, 8), (\texttt{v\_pos}, \texttt{middle}), (\texttt{h\_pos}, \texttt{left})\}$$
$$\sim$$
$$\mathbf{r}_{\texttt{color}} \odot \mathbf{c}_{\texttt{red}} + \mathbf{r}_{\texttt{shape}} \odot \mathbf{d}_8 + \mathbf{r}_{\texttt{v\_pos}} \odot \mathbf{v}_{\texttt{middle}} + \mathbf{r}_{\texttt{h\_pos}} \odot \mathbf{h}_{\texttt{left}}$$

This is an encoding we've come to call a **sum of role-filler pairs**. In certain contexts this encoding might be desirable.

However, we have chosen a slightly different approach. We dispense with role vectors altogether and simply bind the fillers themselves:

$$\{(\texttt{color}, \texttt{red}), (\texttt{shape}, 8), (\texttt{v\_pos}, \texttt{middle}), (\texttt{h\_pos}, \texttt{left})\}$$
$$\sim$$
$$\mathbf{c}_{\texttt{red}} \odot \mathbf{d}_{8} \odot \mathbf{v}_{\texttt{middle}} \odot \mathbf{h}_{\texttt{left}}$$

We have come to call this encoding a **product of fillers**. Assuming we have control over generating the filler vectors, we can make them all quasi-orthogonal and there is no ambiguity between, for instance, shape and color fillers or the position fillers, and therefore no ambiguity about what role they fill in parameterizing the object. Composite vectors encoded this way will all be distinct. Not only does the product of fillers encoding present a more obvious factorization problem, but there are additional reasons to prefer it over a sum of role-filler pairs, which we address in Section 3.5.

The way we encode scenes like the one shown in Figure 3.2 is to superimpose product of filler encodings for each object. Call this the scene vector, $\mathbf{s}$. Such an encoding still provides a flexible data structure such that aspects of the scene can be individually queried. One attractive property of this representation is that its dimensionality does not grow with the number of objects in the scene, nor does it impose any particular ordering on the objects. This in stark contrast to typical approaches in deep learning. These use *localist*, rather than distributed, representations of the object at the output layer and therefore do not support direct superposition. Also, deep learning practitioners use attentional windows to sidestep the issue of processing multiple objects simultaneously. While foveation is a strategy of active perception that benefits both biological and artificial vision systems (Land & Nilsson, 2012; Cheung et al., 2017) , it seems unrealistic to expect that multiple objects will *never* occupy the same window of attention. This type of encoding allows us to represent the whole scene on the same set of units. It solves the "neural binding problem" (Von der Malsburg, 1995).

An issue that remains is how to go from images to symbolic vectors. This being the year 2020, of course one can train a feed-forward neural network to do it. A simple multi-layer peceptron with two fully-connected hidden layers is powerful enough to be trained, with supervision, to output a product of fillers vector like we depict in Figure 3.2. The feed-forward network was trained on a (uniformly) random sample of these scenes, with the MNIST digits chosen from an exclusive training set. Once the feed-forward network has *recoded* the image into a VSA representation, a Resonator Network can go to work parsing it. The vectors $\hat{\mathbf{c}}[t]$, $\hat{\mathbf{d}}[t]$, $\hat{\mathbf{h}}[t]$ and $\hat{\mathbf{v}}[t]$ denote the *guesses* for each factor: color, digit, horizontal- and vertical-location, respectively. The scene can then be decoded by iterating through the

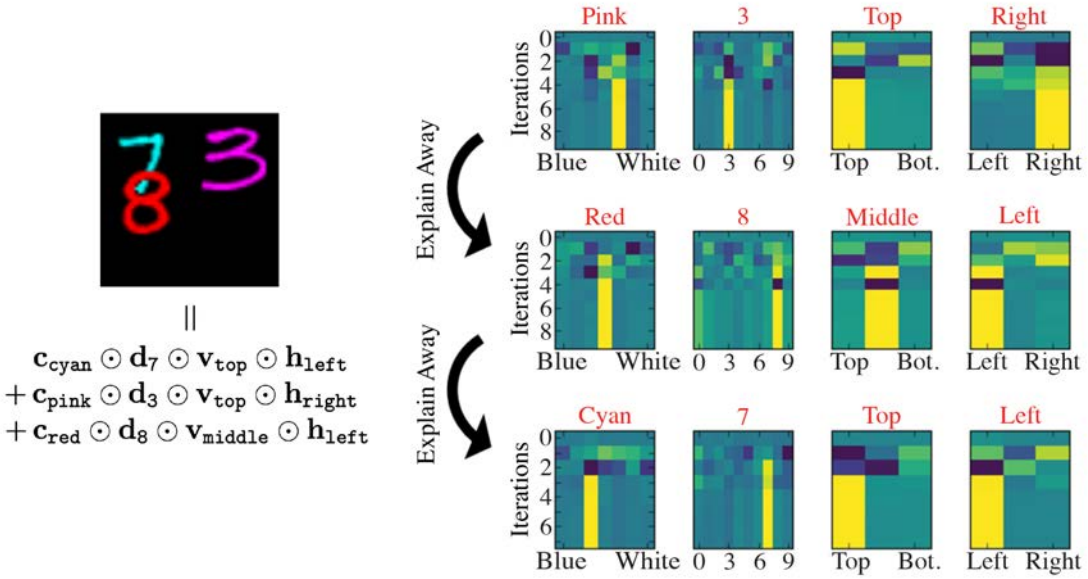


Figure 3.3: Scene vector $\mathbf{s}$ is fed into a Resonator Network which decodes each object in the scene. The model hones in on one object at a time, which is then explained away by subtracting the Resonator Network's converged state from the scene vector. The network is reset and provided with this new input vector. It then converges to another solution, which describes a different object in the scene.

Resonator Network:

$$\begin{aligned}
\hat{\mathbf{c}}[t+1] &= g\Big(\mathbf{C}\mathbf{C}^\top\Big(\mathbf{s} \odot \hat{\mathbf{d}}[t] \odot \hat{\mathbf{v}}[t] \odot \hat{\mathbf{h}}[t]\Big)\Big) \\
\hat{\mathbf{d}}[t+1] &= g\Big(\mathbf{D}\mathbf{D}^\top\Big(\mathbf{s} \odot \hat{\mathbf{c}}[t+1] \odot \hat{\mathbf{v}}[t] \odot \hat{\mathbf{h}}[t]\Big)\Big) \\
\hat{\mathbf{v}}[t+1] &= g\Big(\mathbf{V}\mathbf{V}^\top\Big(\mathbf{s} \odot \hat{\mathbf{c}}[t+1] \odot \hat{\mathbf{d}}[t+1] \odot \hat{\mathbf{h}}[t]\Big)\Big) \\
\hat{\mathbf{h}}[t+1] &= g\Big(\mathbf{H}\mathbf{H}^\top\Big(\mathbf{s} \odot \hat{\mathbf{c}}[t+1] \odot \hat{\mathbf{d}}[t+1] \odot \hat{\mathbf{v}}[t+1]\Big)\Big)
\end{aligned} \tag{3.3}$$

Our encoding of visual scenes superposes a composite vector for each object, each of which individually is a valid solution to the factorization of the scene. When we present the scene vector $\mathbf{s}$ to a Resonator Network, it automatically hones in on a particular one of these composites, finding its factors. For instance, in Figure 3.3 the Resonator Network first identifies the pink three in the top right. Once the factorization has been found, this object is then "explained away" by subtracting it from $\mathbf{s}$. What remains are the other composites, still in superposition. The Resonator Network is then reset (each resonator is reinitialized to the superposition of all possible codevectors) and presented with the new explained-away scene vector. It will then hone in on one of the remaining objects, in this case the red 8. This sequence may be repeated until all the objects have been decoded. This technique is similar

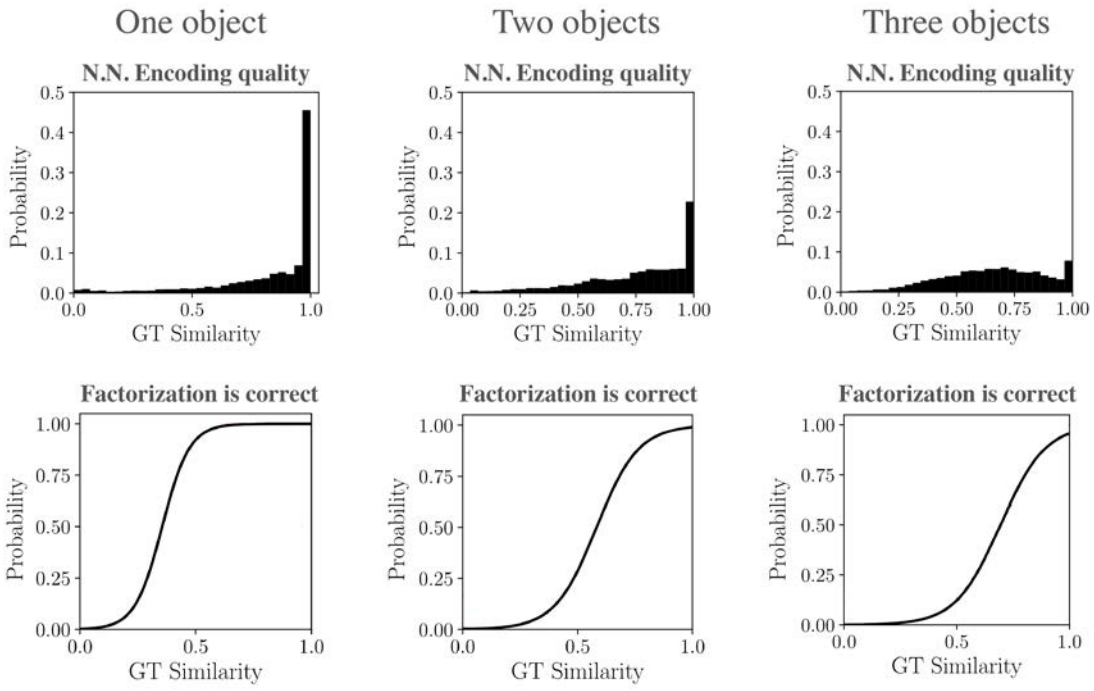


Figure 3.4: Resonator Networks correct encoding errors. Visual scenes with one, two, and three objects are separated into separate columns. Top row gives encoding quality, in terms of cosine similarity between the feed-forward network output and the ground-truth scene vector, across the test-set. We define correct factorization as the case where the Resonator Network correctly infers all the factors of all objects. The bottom row shows the empirical probability of a correct factorization as a function of similarity to the ground truth scene vector. Lines are logistic function fits to the data.

to what is known as "deflation" in the context of tensor decomposition methods (da Silva et al., 2015).

After training on 100,000 images, we used the network to produce symbolic vectors for a held-out test set of 10,000 images. The vector dimensionality N is a free parameter, which we chose to be 500. If the exact ground truth vector is provided to a Resonator Network, it will infer the factors with 100% accuracy provided N is large enough, a fact we established in Chapter 2. For this small visual scene example it turns out $N = 500$ more than suffices for the number of possible factorizations to be searched. Note that $N = 500$ is less than the total number of combinations of all the factors, which is 630.

The encoder network generates VSA scene vectors that are close to the ground-truth encoding, but there is some error. The error gets larger with more digits in the scene, perhaps partially due to occlusion of the digits. Figure 3.4 shows that the Resonator Network can tolerate significant error in the scene vector produced by the feed-forward encoding network, correcting for ambiguity not resolved in the encoding step.

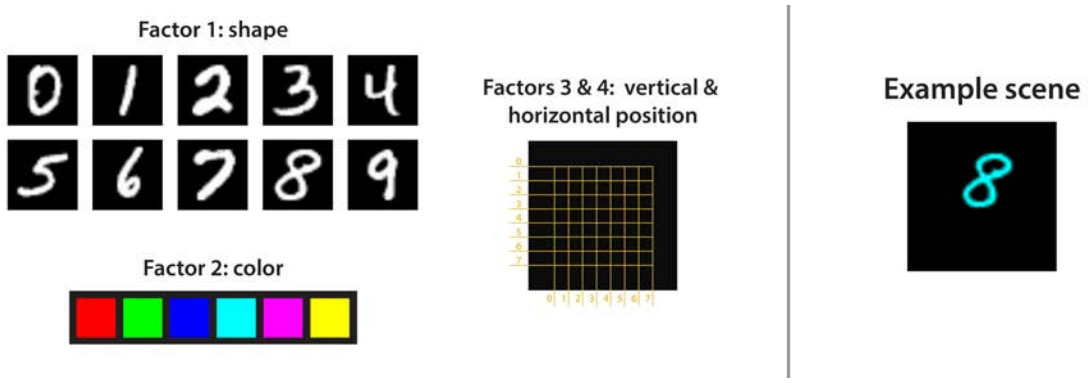


Figure 3.5: Simple 2D scenes with 4 factors

One may be concerned that using a generic neural network in the way we have shown is fairly opaque and brittle. We share this concern. We do see a role for learned neural network encodings as a front-end for perception, but as it stands this particular network is essentially a black box. While perhaps it should be no surprise that a network can be trained for such a simple dataset, it is a starting point for making further improvements, which we develop over the next few sections.

3.2.2 Removing the black box completely

In this section we demonstrate a mapping from images to bipolar vectors which does not involve a neural network. The dataset is simpler, in that it uses only one exemplar for each digit shape, but the method requires *no learning*. A Resonator Network can consume the bipolar vector produced by this mapping, factoring it into each digits' constituent parameters by employing the "explaining away" technique we have previously introduced. What is remarkable about this encoding is that it uses the principle of superposition to go from low level features–individual pixels–to high-level discrete concepts.

In Figure 3.5 we show the parameters for a set of artificial scenes. The parameters of these scenes are similar, but *not the same* as what we used in the previous section. The reason is simply that we conducted these experiments six months apart and had a slightly different variation of the data we had been using in other applications. Each object in this world can be decomposed into 4 factors: shape, color, vertical translation, and horizontal translation. What one measures is a *conjunction* of the 4 factors (an image of the object) and the task is to infer these components. A scene may contain more than one object, in which case the task is to infer the factors for each object separately. There are 10 shapes, 6 colors, 8 discrete values for both vertical and horizontal translation (each separated by 4 pixels), and the images themselves are 56 by 56 pixels.

The Resonator Network we use to compute the factorization has 4 codebooks: $\mathbb{S}$ for shape, $\mathbb{R}$ for color, $\mathbb{T}_v$ for vertical translation, and $\mathbb{T}_h$ for horizontal translation. The following sections detail how these codebooks are set.

Encoding shape templates

There are ten shapes, one for each of the digits 0 to 9, and they are shown in Figure 3.5. We express the pixel values for each shape as $p_i(y, x)$ where i goes from 1 to 10 and y indexes vertical position while x indexes horizontal position. Shapes are taken from the MNIST dataset, but we choose a single *template* shape for each digit–in this demonstration we are not generalizing across different styles in the MNIST dataset.

The pixel values are in the interval $[0, 1]$. We generate a random codebook $\mathbb{V}$ for vertical pixel position as well as a random codebook $\mathbb{H}$ for horizontal pixel position (the shape images are 28×28 pixels):

$$\mathbb{V} := \{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{27}\} \quad \mathbb{H} := \{\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{27}\}$$

The codebook matrices $\mathbf{V}$ and $\mathbf{H}$ stack these vectors into their respective columns, following our usual convention. The bipolar vector $\mathbf{s}_i$, which encodes the ith shape, is generated in the following way:

$$\mathbf{s}_i := \operatorname{sgn}\Big(\sum_{y,x} p_i(y, x)\, \mathbf{v}_y \odot \mathbf{h}_x\Big) \tag{3.4}$$

The properties of this encoding are fairly interesting. For high enough N, it would be virtually impossible to discern the digit to which $\mathbf{s}_i$ corresponds unless one were given the label a priori, or had access to the codebooks $\mathbb{V}$ and $\mathbb{H}$. Yet, $\mathbf{s}_i$ still contains almost all of the information present in the image in the sense that we can approximately generate the image from $\mathbf{s}_i$. The cosine similarity between the vector $\mathbf{v}_y \odot \mathbf{h}_x$ and $\mathbf{s}_i$ can be used to approximately recover pixel value $p_i(y, x)$:

$$\operatorname{sim}(\mathbf{v}_y \odot \mathbf{h}_x,\, \mathbf{s}_i) := \frac{1}{||\mathbf{v}_y \odot \mathbf{h}_x||_2\ ||\mathbf{s}_i||_2}(\mathbf{v}_y \odot \mathbf{h}_x)^T \mathbf{s}_i = \frac{1}{N}\sum_{j=1}^{N}\big(\mathbf{v}_y \odot \mathbf{h}_x \odot \mathbf{s}_i\big)_j = \alpha p_i(y, x) + \gamma$$

where α represents a scaling indeterminancy due to the bipolarizing function $g(\cdot)$ and γ is a noise term due to the (hopefully small) overlaps between $\mathbf{v}_y \odot \mathbf{h}_x$ and the other terms in the sum defining $\mathbf{s}_i$. If we know pixels values are bounded (we do in this case, they are in the interval $[0, 1]$), then we can estimate α from $\operatorname{sim}(\mathbf{v}_y \odot \mathbf{h}_x, \mathbf{s}_i)$. Our estimate for pixel $p_i(y, x)$ we denote by $\hat{p}_i(y, x) = \frac{1}{\alpha}\operatorname{sim}(\mathbf{v}_y \odot \mathbf{h}_x, \mathbf{s}_i)$. The variance of the noise γ is reduced by increasing N–this fact is illustrated in Figure 3.6. The information in the image is contained in $\mathbf{s}_i$, but is in some sense *encrypted* by the codebooks, and the invertibility of this encoding improves with increasing N.

Another interesting property of our mapping is that $\mathbf{s}_i$ is *distributed* in the sense that information is spread evenly across the elements of the vector. This is in contrast to the *localist* representations of many modern deep learning approaches, in which the burden of representing a scene parameter may fall on an individual component of the (usually low-dimensional) vector representation at the final layer of these networks. The benefits of a distributed representation have been argued in Hinton et al. (1986), Kanerva (2009), and

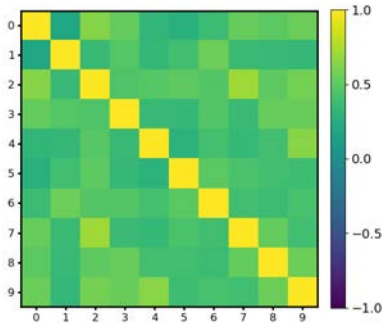
(a) Original images, $p_i(y, x)$

(b) $\hat{p}_i(y, x)$ for $N = 1{,}000$

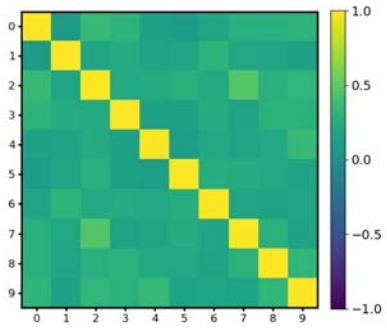
(c) $\hat{p}_i(y, x)$ for $N = 10{,}000$

(d) $\hat{p}_i(y, x)$ for $N = 100{,}000$

Figure 3.6: Bipolar vector $\mathbf{s}_i$ noisily encodes the original image



(a) Cosine similarity matrix, images



(b) Cosine similarity matrix, $\mathbf{s}_i$

Figure 3.7: Bipolar encoding makes shapes more orthogonal

many other places. They include the fact that distributed representations are highly robust to pertubations and give a natural way to superpose the representation of multiple objects without having to add any additional units. Besides these properties, encoding shapes according to (3.4) also makes them slightly more orthogonal. We plot in Figure 3.7a the cosine similarity matrix between all of the shape images and in Figure 3.7b the cosine similarity matrix between all of the vectors $\mathbf{s}_i$. There is an increase in orthogonality between different shapes when we encode them in this way, which may aid in the determination of which shape is present in an image. We stack each $\mathbf{s}_i$ into the columns of matrix $\mathbf{S}$, which we call the shape codebook matrix.

Encoding color

Color can be encoded the same way we encoded shape, where the sum in (3.4) is instead over color channels, but the issue with this is that the encodings for some colors overlap significantly–cyan is blue *plus* green (the encoding for cyan will be similar to both blue and green), pink is red *plus* blue, and yellow is red *plus* green. We want the bipolar vector representation for each color to be as orthogonal to the others as possible, so we just draw a random codebook $\mathbb{R}$ with 6 vectors, one for each color:

$$\mathbb{R} := \{\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_5\}$$

with $\mathbf{R}$ the associated codebook matrix. When we encounter a color image, we can decode the RGB pixel color, represented by the 3-element vector $\mathbf{p}$, with a simple winner-take-all perceptron

$$\sigma(\mathbf{p}) := WTA(\mathbf{W}\mathbf{p} + \mathbf{b})$$

where we just set $\mathbf{W}$ and $\mathbf{b}$ manually so that this outputs the correct index for each of the 6 colors.

Encoding translation

The two remaining factors in the dataset (Figure 3.5) are the 8 vertical and horizontal positions of the shape within a larger scene. We'll refer to these as translations of the template so as to distinguish them from position *within* a shape template. Translation of the template is encoded by the random codebooks $\mathbb{T}_v$ and $\mathbb{T}_h$:

$$\mathbb{T}_v := \{\mathbf{t}_{v_0}, \mathbf{t}_{v_1}, \ldots, \mathbf{t}_{v_7}\} \quad \mathbb{T}_h := \{\mathbf{t}_{h_0}, \mathbf{t}_{h_1}, \ldots, \mathbf{t}_{h_7}\}$$

and again following our convention, $\mathbf{T}_v$ and $\mathbf{T}_h$ are the associated codebook matrices. The last thing we must address is how to encode pixel position in the larger scene (which potentially contains several different shapes with initially unknown factors). We have already defined $\mathbb{V}$ and $\mathbb{H}$ for pixel position *within a shape template* and $\mathbb{T}_v$ and $\mathbb{T}_h$ for translation of this shape template–we combine them in order to encode pixel positions in the full scene. Suppose that in some scene we have a nonzero pixel at location $(8, 9)$. This could come from the $(8, 9)$ pixel of a shape template that is not at all translated, or it could come from the $(4, 3)$ pixel of a shape template that has been been translated by 4 pixels vertically and 6 pixels horizontally (or many other possible combinations of template pixel position and template translation). One can see that there is an ambiguity here due to the need to simultaneously estimate shape and translation from only the pixel information. Our strategy is to superimpose all the relevant combinations of template pixel position and template translation. We define f as the encoding of a position within the full scene in terms of codebooks for the template pixel position and template translation:

$$f(y\ ;\ \mathbb{V}, \mathbb{T}_v\ ) := \sum_{y',\, y'' \in \mathbb{Q}_y} \mathbf{v}_{y'} \odot \mathbf{t}_{v_{y''}} \tag{3.5}$$

$$f(x\ ;\ \mathbb{H}, \mathbb{T}_h\ ) := \sum_{x',\ x'' \in \mathbb{Q}_x} \mathbf{h}_{x'} \odot \mathbf{t}_{h_{x''}} \tag{3.6}$$

The sets $\mathbb{Q}_y$ and $\mathbb{Q}_x$ contain all of the combinations for pixel position within a template and template translation that would generate y and x, respectively. The reason this approach makes some sense is that given the correct translation, a full shape vector $\mathbf{s}_i$ should pop out of an image encoded using f, albeit with some "noise" due to the extra terms in the sum. The *coincidence* of the same translation applied to every pixel in a shape template is one of the ideas underpinning Hinton's Capsules theory (Hinton, 1981b; Sabour et al., 2017) and Olshausen's theory of Dynamic Routing in biological vision (Olshausen et al., 1993). If our definition of f seems to be an inelegant way to deal with the translation ambiguity, it is because bipolar vectors are not the natural choice for encoding translation via multiplication–complex-valued vectors are much more appropriate, which we will cover in the next section.

Performance

Putting all the pieces together, we set up a Resonator Network with the codebook matrices $\mathbf{S}$ (shape), $\mathbf{R}$ (color), $\mathbf{T}_v$ (vertical translation), and $\mathbf{T}_h$ (horizontal translation). In our RGB scenes, pixels are $\mathbf{p}(y, x)$ and the greyscale pixel luminance we write $p(y, x)$ (taken as the average of $\mathbf{p}(y, x)$). The encoding for each image is:

$$\mathbf{s} = \text{sgn}\Big(\sum_{y,x} p(y,x)\, f(y\ ;\ \mathbb{V}, \mathbb{T}_v\ ) \odot f(x\ ;\ \mathbb{H}, \mathbb{T}_h\ ) \odot \mathbf{r}_{\sigma(\mathbf{p}(y,x))} \Big) \tag{3.7}$$

This generates a single bipolar vector of size N which represents everything in the scene, and this was done purely through superposition over a simple encoding of each pixel, not with a neural network.

Our results show that this vector contains all the information necessary to factor the scene into its components. If a scene contains more than one digit, we can use the explaining away procedure to factor each digit one at a time. When generating the images, we layer the digits on top of one another, so there can be significant occlusion due to those that are in the foreground. We place the digits in the scene in the order 3, 2, 1, so *Object 1* is always guaranteed to be in the foreground, unobscured.

The results we report in Table 3.1 were generated with bipolar vectors of size $N = 10{,}000$ and a Resonator Network with OLS weights. There are 3840 possible scenes with a single digit and we applied the Resonator Network to factoring each one of them–the accuracy was 0.932 in our experiment. There are 3840^2 possible two-object scenes, so we randomly subsample 5000 of them. We also randomly subsample 5000 of the three-digit scenes. The object in the foreground is always easiest to factor, and this is often the one chosen first by the Resonator Network. We ran our model on the three scenes shown in Figure 3.8; the reported factorizations are shown Table 3.2.

Notice that the object in the foreground is the most salient–it has a proportionally larger representation in $\mathbf{s}$ than do the other digits, so it makes sense that it would be factored first.

	Obj 1	Obj 2	Obj 3
One object	0.932		
Two objects	0.864	0.824	
Three objects	0.826	0.762	0.688

Table 3.1: Factorization accuracies for multi-factor digit scenes

(a) One object (b) Two objects (c) Three objects

Figure 3.8: Example multi-factor digit scenes

Scene	Reported factorization	Accuracy
3.8a	{ shape: 8, color: cyan, vertical trans: 2, horizontal trans: 4 }	1.0
3.8b	{ shape: 2, color: pink, vertical trans: 5, horizontal trans: 3 }	1.0
	{ shape: 2, color: yellow, vertical trans: 7, horizontal trans: 3 }	1.0
3.8c	{ shape: 6, color: blue, vertical trans: 0, horizontal trans: 2 }	1.0
	{ shape: 5, color: yellow, vertical trans: 0, horizontal trans: 0 }	1.0
	{ shape: 5, color: red, vertical trans: 1, horizontal trans: 0 }	1.0

Table 3.2: Reported factorizations for scenes in Figure 3.8

Once the Resonator Network factors this object, it can be subtracted from the scene and what is left are fragments of the digit(s) that had been occluded–this information is enough, at least in these examples, to factor the remaining digits as well.

3.2.3 A continuous, complex vector encoding

The representation of position in the previous section left several things to be desired. Position in one dimension is a continuously-varying scalar variable, yet we were encoding in the non-continuous domain of bipolar vectors. Worse still, the assignment of vectors to discrete positions was random, not reflecting the underlying geometry of the real number line. One partial solution to this problem is a so-called "thermometer code" which flips bits sequentially to encode nearby positions, but this still enforces a fundamental discreteness (and therefore a minimum resolution) on the encoding of what is ultimately a continuous quantity.

We prose a remedy that relies on complex-valued vectors. This is an encoding that several of us at the Redwood Center for Theoretical Neuroscience have been using over the last 4 years to encode continuous quantities whenever they arise in VSA applications. I have come to call it "phase-scaling."

We use complex-valued vectors, particularly the subset Plate calls "unitary" vectors, which are those in which each element has complex magnitude 1 (Plate, 1994). These vectors, which are the most direct generalization of bipolar vectors, have many desirable properties like the exactness and numerical stability of inversion as well as exact similarity-preservation under the binding operator (the Hadamard product).

The phase-scaling mapping is not completely deterministic in that randomness is generated in an initial setup phase. Ultimately the mapping moves along a one-dimensional subspace of the N-dimension space of unitary vectors–this initial randomness determines which one-dimensional subspace gets traversed. We start by drawing a *base vector* of phases:

$$\boldsymbol{\theta} := \begin{bmatrix} \theta_1 \\ \theta_1 \\ \vdots \\ \theta_N \end{bmatrix}$$

Where each θ_i is drawn i.i.d. from the uniform distribution on the interval $[-\pi, \pi)$. There is a parameter of the phase-scaling encoding that I call the "bandwidth," denoted in this section by b.

Then, the phase-scaling encoding of x with a bandwidth of b and base phases $\boldsymbol{\theta}$ is:

$$V(x;\, \boldsymbol{\theta}, b) := \begin{bmatrix} e^{j\frac{\theta_1}{b}x} \\ e^{j\frac{\theta_2}{b}x} \\ \vdots \\ e^{j\frac{\theta_N}{b}x} \end{bmatrix} \tag{3.8}$$

Hopefully it is clear why I am calling this a phase-scaling. We take a vector of assorted complex phases and scale these phases by x, wrapping them around the unit circle at a rate determined by θ_i/b, which one can think of more like a frequency.

The first thing to notice is that this mapping is continuous. The *smoothness* of this continuous mapping is something we will come to next. Assuming N is of the size normally used in VSAs (≥ 1000), it is bijective over a large range of x. The mapping does eventually go through a cycle, whose length is related to the least common multiple of $\{\theta_1/b, \theta_2/b, \ldots, \theta_N/b\}$, modulo π. I have not made this statement completely concrete, but hopefully the gist is clear. This mapping is briefly mentioned in Plate's thesis (in Section 3.6.5), where he refers to it as the "convolutive power" and he uses this encoding in trajectory-association experiments in Section 5 (Plate, 1994). He mainly uses it, however, with *integer* "powers," which effectively generate random trajectory-association labels, and does not particularly stress the continuity or bandwidth of this encoding.

It has not been lost on us that this encoding bears more than a passing resemblance to a grid-cell code (Hafting et al., 2005; Fiete et al., 2008). While we have not pursued this in depth, it may be that the next section suggests a nice interpretation of high-dimensional grid-cell codes.

3.2.3.1 Properties of the cosine similarity

Suppose we have two complex-valued vectors $\mathbf{v}_1$ and $\mathbf{v}_2$:

$$\mathbf{v}_1 := \begin{bmatrix} m_1 e^{j\phi_1} \\ m_2 e^{j\phi_2} \\ \vdots \\ m_N e^{j\phi_N} \end{bmatrix} \quad \mathbf{v}_2 := \begin{bmatrix} p_1 e^{j\gamma_1} \\ p_2 e^{j\gamma_2} \\ \vdots \\ p_N e^{j\gamma_N} \end{bmatrix}$$

The cosine similarity is defined as

$$\begin{aligned} sim(\mathbf{v}_1, \mathbf{v}_2) &:= \Re\Big[\frac{\mathbf{v}_1^H \mathbf{v}_2}{||\mathbf{v}_1||\,||\mathbf{v}_2||}\Big] \\ &= \frac{\Re\big[\sum_i^N m_i p_i e^{j(\gamma_i - \phi_i)}\big]}{||\mathbf{v}_1||\,||\mathbf{v}_2||} \\ &= \frac{\sum_i^N m_i p_i \cos(\gamma_i - \phi_i)}{||\mathbf{v}_1||\,||\mathbf{v}_2||} \end{aligned}$$

where $\mathbf{v}_1^H$ is the conjugate-transpose of $\mathbf{v}_1$. When the vectors are "unitary," their Euclidean norms ($||\cdot||$) are $\sqrt{N}$, and this expression simplifies to

$$\frac{1}{N}\sum_i^N \cos(\gamma_i - \phi_i) \tag{3.9}$$

Let us leave this expressed as a complex number and simply examine the properties of cosine similarity between the encoding of two distinct values x_1 and x_2:

$$sim(V(x_1;\,\boldsymbol{\theta}, b), V(x_2;\,\boldsymbol{\theta}, b)) = \frac{1}{N}\Re\big[\sum_i^N e^{j\frac{\theta_i}{b}(x_2 - x_1)}\big] := \frac{1}{N}\Re\big[\sum_i^N e^{j\frac{\theta_i}{b}\Delta x}\big] \tag{3.10}$$

This expression is obviously shift-invariant and maximized at $\Delta x = 0$. Recall that the random variables θ_i are uniformly distributed on the interval $[-\pi, \pi)$, so the term $\frac{\theta_i}{b}$ is uniform on $[-\frac{\pi}{b}, \frac{\pi}{b})$. What happens in the limit of large N is quite nice:

$$\lim_{N\to\infty} \frac{1}{N}\Re\big[\sum_i^N e^{j\frac{\theta_i}{b}\Delta x}\big] = \Re\big[\int_{-\infty}^{\infty} \mathrm{rect}(bf)\, e^{j2\pi f \Delta x}\, df\big] \tag{3.11}$$

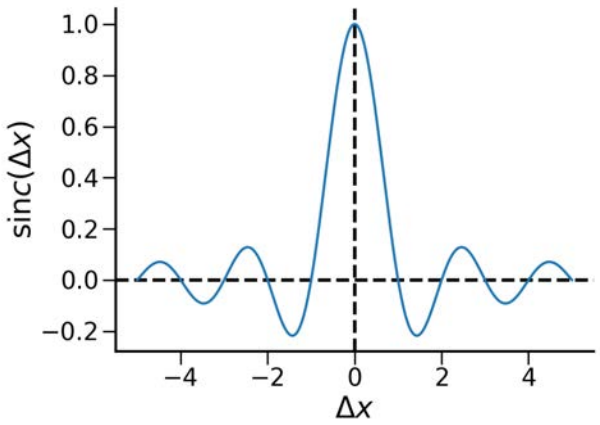


Figure 3.9: Normalized sinc function

Where $\text{rect}(t)$ is the rectangle function:

$$\text{rect}(t) = \begin{cases} 1 & -\frac{1}{2} < t < \frac{1}{2} \\ \frac{1}{2} & |t| = \frac{1}{2} \\ 0 & otherwise \end{cases} \tag{3.12}$$

Let us just assume b is 1 for a moment. What we have in (3.11) is the real part of *the inverse Fourier transform of the rectangle function.* Therefore, it is equal to the normalized sinc function:

$$\text{sinc}(\Delta x) := \frac{\sin(\pi \Delta x)}{\pi \Delta x} \tag{3.13}$$

which I plot in Figure 3.9. Note that the zero-crossings of the normalized sinc are on the integers. Also note that it is real-valued. For a general bandwidth b, the signal $\text{rect}(bf)$ is non-zero on the interval $(-\frac{1}{2b}, \frac{1}{2b})$ and the inverse Fourier transform of this signal has zero-crossings at integer multiples of b.

What the combination of (3.10) and (3.11) shows is that, when we encode x_1 and x_2 with the phase-scaling encoding, the cosine similarity between the resulting vectors is a sinc function applied to their difference Δx, and that we can tune the bandwidth of this sinc function with the parameter b.

We use Figures 3.10-3.12 to illustrate that the preceeding analysis is borne out in simulation, and how this type of encoding can be used. First, in Figure 3.10 we tile the interval $[-10, 10]$ with vectors representing the values $\{-10, -5, 0, 5, 10\}$. The bandwidth b can be adjusted to tune the degree of specifity for the encoding. This is how we might imagine controlling the amount of *interpolation* that comes from taking simple linear combinations of adjacent vectors. Figure 3.11 fixes x_1 to 0 and demonstrates how the bandwidth is exactly b in expectation. For small N, deviation from a sinc function occurs on a trial-by-trial basis (for different random samples of the base vector $\boldsymbol{\theta}$), but this "noise" is mean-zero and its variance is small. This is shown by Figure 3.12.

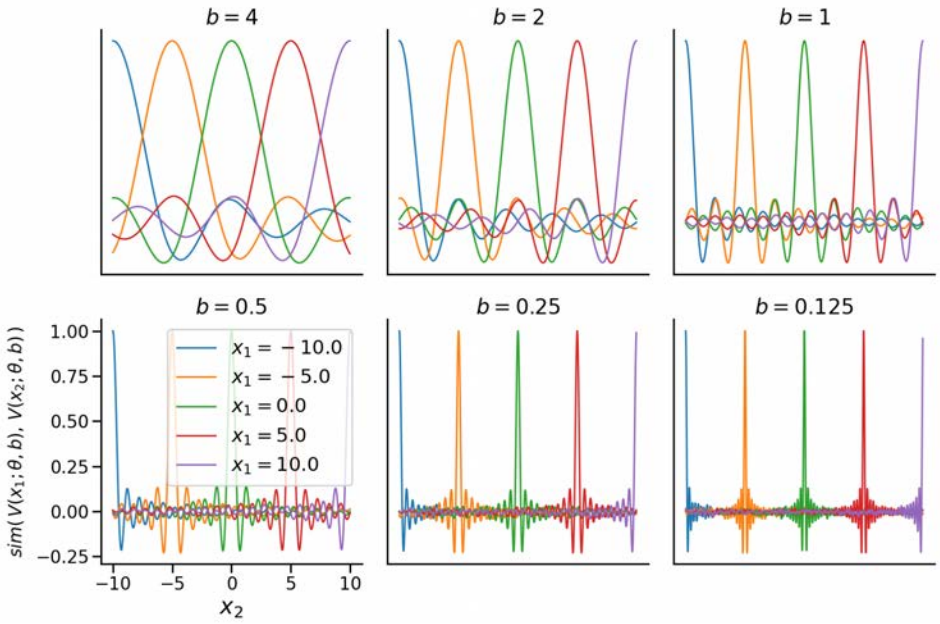


Figure 3.10: Phase-scaling encoding can be used to tile a domain with varying degrees of specificity. In these simulations $N = 1000$.

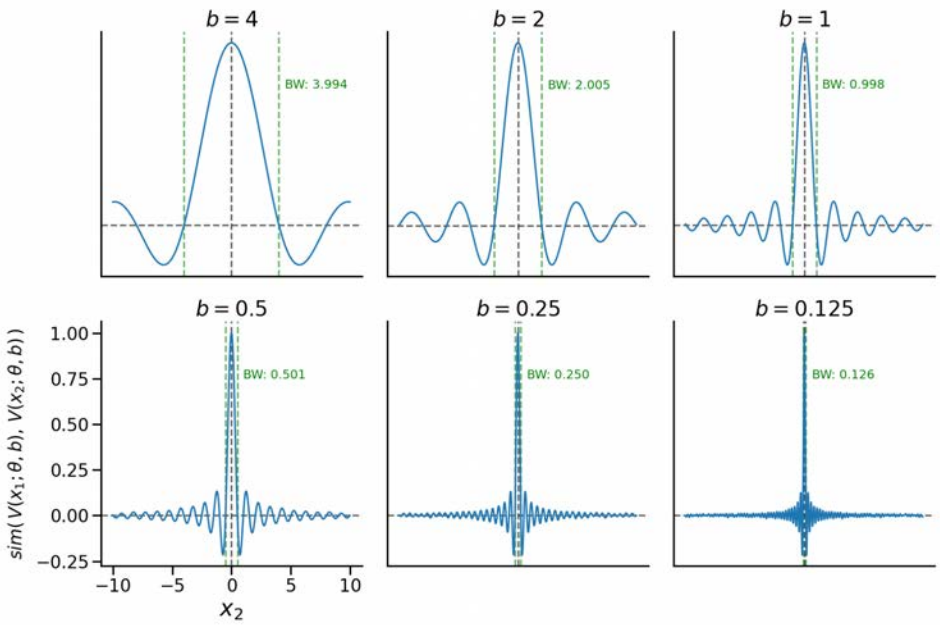


Figure 3.11: Expected bandwidth of cosine similarity is exactly b. Plot shows empirical mean of cosine similarity estimated from 100 random trials, and exact bandwidth calculated from linear interpolation between measured points.

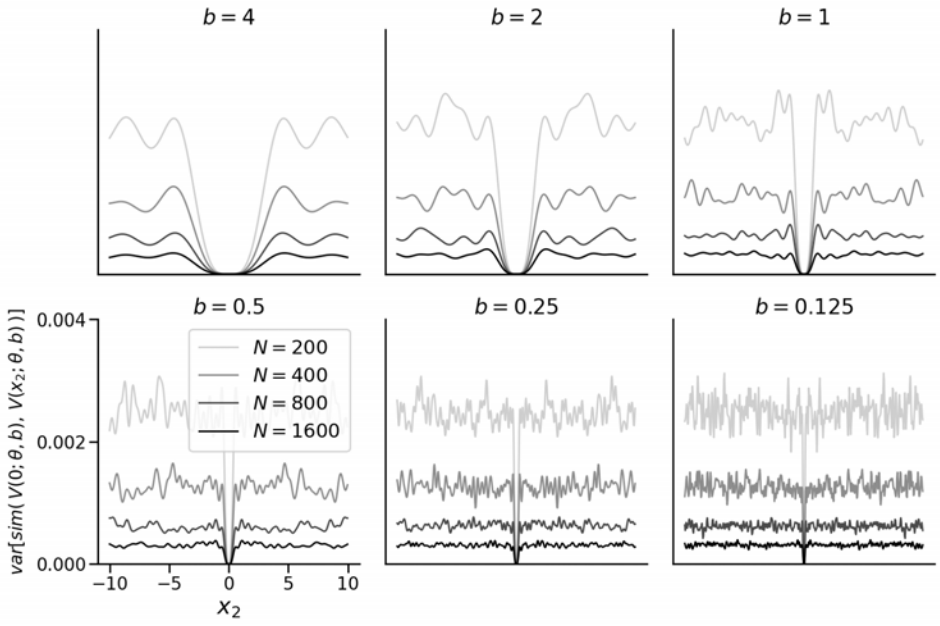


Figure 3.12: Variance of the cosine similarity is small and depends on N

Notice also that the phase-scaling encoding induces an isomorphism between addition with real numbers and the Hadamard product applied to these unitary complex vectors. Due to the 2π-symmetry of phase variables, this mapping is not technically invertible over the entire real number line (there will eventually be a cycle that makes the original x ambiguous), but it will be invertible over a very large interval. As a consequence of the fact that multiplication in the vector space produces translation in the input space, translation of multiple features stored in superposition becomes quite elegant. We can reimplement the scene encoding scheme covered in Section 3.2.2, this time with complex vectors, and simplify the encoding of position dramatically. We won't devote more space to a discussion here, as this has been previously published in Frady, Kent, Kanerva, et al. (2018).

3.3 Vector symbolic scene transformation

This section is about some of our early attempts to generate transformations of visual scenes by using VSA algebra to manipulate representations of those scenes. This involves an encoding of the image, followed by a transformation, and then a rendering of the transformed scene (depicted in Figure 3.14 and explained below).

Physically-realistic transformations of visual scenes provide strong cues for underlying structure. In particular, naturalistic movements of objects, smooth changes in lighting, or changes in camera viewpoint can help a vision system better model the three-dimensional configuration of things in the scene. The idea of modeling transformations in a latent space (i.e. not the space of images) is depicted in Figure 3.13. Transformations of the underlying

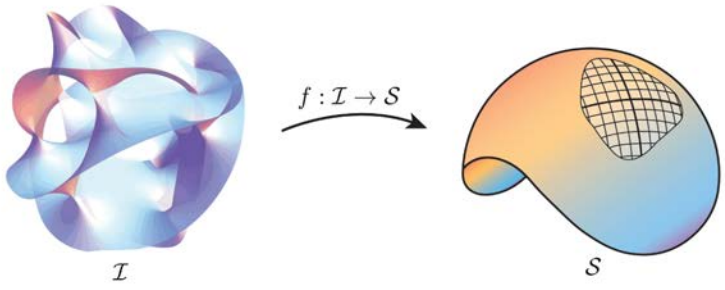


Figure 3.13: The objective of modeling transformation in a new space $\mathcal{S}$

three-dimensional scene produce transfomations in the image space $\mathcal{I}$ which are convoluted and difficult to model. One should seek a mapping from $\mathcal{I}$ into a new latent space $\mathcal{S}$ that simplifies the geometry of these transformations.

At some level this always simplifies to 1) finding a mapping $f(\cdot) : \mathcal{I} \to \mathcal{S}$ and 2) finding operators on $\mathcal{S}$ that capture the desired transformations. In some prior work, the mapping is learned based on a task largely unrelated to transformation, but then it is observed after the fact that simple operations, including vector sums and differences, approximately capture meaningful transformations (Cheung et al., 2015; Radford et al., 2015). The more interesting results have been where transformation is explicitly incorporated into the learning objective. In some cases the mappings are selected from a large and hard to interpret family of neural network mappings (Reed et al., 2014; Reed et al., 2015; Tulsiani et al., 2018), while in others, simpler and more easily analyzed mappings have been learned (Culpepper & Olshausen, 2009; Sohl-Dickstein et al., 2010; Cadieu & Olshausen, 2012). It is often an objective, either implicit or explicit, that the mapping *linearize* the manifold of scene representations (Chen et al., 2018).

We take a slightly different approach here. On the one hand, we allow the mapping to be a neural network (specifically a multi-layer perceptron with two hidden layers). However we aim to *impose* structure on the learned mapping by applying VSA transformations to the output of the encoder. Moreover, instead of being based on simple scalings and translations of vectors, our transformations are based on binding.

Suppose the encoded representation of our scene is vector $\mathbf{s}$. We notate a generic transformation of the scene as

$$\mathbf{s}^{\texttt{new}} = T_\alpha(\mathbf{s}, \{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_k\}_\alpha)$$

where $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_k\}_\alpha$ are some VSA vectors that are used to encode a particular transformation labeled α, and $\mathbf{s}_{\texttt{new}}$ is the VSA representation of the transformed scene. We can set up an autoencoder-like architecture where error is calculated on a rendered image of $\mathbf{s}_{\texttt{new}}$, and this is backpropagated through the entire model. The actual vectors $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_k\}_\alpha$ may be selected based on a provided label for the transformation, or in some cases it may be possible to infer them. This makes model training unsupervised, in a loose sense. See Figure 3.14 for a diagram of the model. We call it a Vector Symbolic Scene Transformer

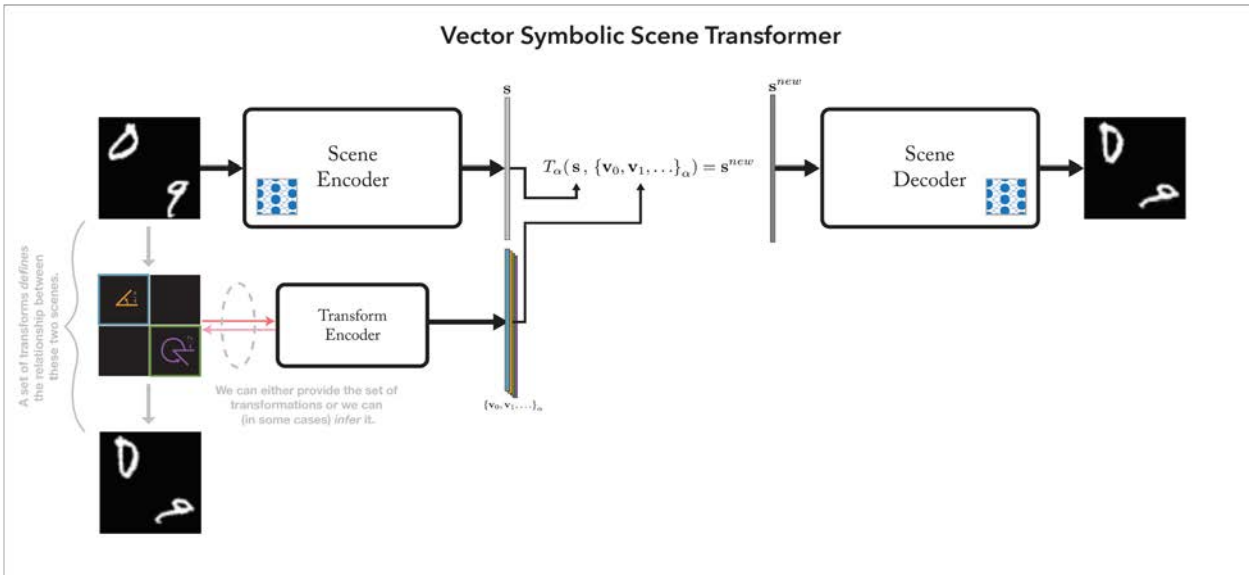


Figure 3.14: Vector Symbolic Scene Transformer

(not to be confused with transformers in the sense of Vaswani et al. (2017) and Devlin et al. (2018), which became popular after we introduced this (Kent & Olshausen, 2017)).

One objective of this work was to show that one can coerce $\mathbf{s}$ to have certain VSA structures simply by the operations applied to it. Figure 3.14 shows data that has three factors–shape, position, and rotation. There are two digits in a scene, each occupying one of four quandrants for position. The task is to selectively rotate individual objects–induce rotations which are *local* and not applied over the entire image (despite the fact that $\mathbf{s}$ encodes the entire image). This demonstrates an encoding which captures the *conjunction* of position with rotation. Suppose position is encoded with vectors $\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$. Rotation is encoded with $\mathbf{r}_\theta$, where this takes the continuous phase-scaling encoding we introduced in the previous section.

The transformation shown in Figure 3.14 is a $\pi/4$ rotation applied to the digit in quadrant 1 and a $7\pi/4$ rotation applied to the digit in quadrant 3. We capture such a transformation in the following way:

$$\mathbf{s}^{\texttt{new}} = \rho(\mathbf{q}_1) \odot \mathbf{r}_{\pi/4} \odot \mathbf{q}_1^{-1} \odot \mathbf{s} + \rho(\mathbf{q}_3) \odot \mathbf{r}_{7\pi/4} \odot \mathbf{q}_3^{-1} \odot \mathbf{s} \tag{3.14}$$

The motivation for this is that *if* $\mathbf{s}$ has a product of fillers representation for each digit (like we have used in previous sections, see 3.2.1, for example), then such a transformation will result in precisely what we are after:

$$\begin{aligned} \mathbf{s} &= \mathbf{d}_0 \odot \mathbf{r}_{\theta'} \odot \mathbf{q}_1 + \mathbf{d}_9 \odot \mathbf{r}_{\theta''} \odot \mathbf{q}_3 \\ \implies \mathbf{s}^{\texttt{new}} &\approx \rho(\mathbf{q}_1) \odot \mathbf{r}_{\pi/4} \odot \mathbf{d}_0 \odot \mathbf{r}_{\theta'} + \rho(\mathbf{q}_3) \odot \mathbf{r}_{7\pi/4} \odot \mathbf{d}_9 \odot \mathbf{r}_{\theta''} \\ &= \mathbf{d}_0 \odot (\mathbf{r}_{\pi/4} \odot \mathbf{r}_{\theta'}) \odot \rho(\mathbf{q}_1) + \mathbf{d}_9 \odot (\mathbf{r}_{7\pi/4} \odot \mathbf{r}_{\theta''}) \odot \rho(\mathbf{q}_3) \end{aligned} \tag{3.15}$$

What we are left with at the end of 3.15 is an encoding of the same shapes in the same positions, but modified by a relative rotation of $\pi/4$ and $7\pi/4$, respectively. We have added parentheses in (3.15) simply to make it easier to visually parse. The reason we must use $\rho(\mathbf{q}_1)$ in addition to $\mathbf{q}_1$ to specify position is because otherwise the commutativity of $\odot$ would cause $\mathbf{q}_1$ to nullify the effect of $\mathbf{q}_1^{-1}$. Also note that we never explicitly encode digit shape; it is assumed that this information is present in the form of some vector $\mathbf{d}$ produced by the encoder, but it is not explicitly part of the transformation imposed in 3.14.

This idea can be scaled up significantly and generalized to other types of physical properties. We show in Figure 3.15 scenes that have 6 digits in one of nine different grid positions, each rotated independently from one another. In Figure 3.16 we challenge the model with scenes from a dataset of rendered 3D scenes that we developed specifically for this project. A new model is trained for each of the different datasets and transformation tasks. The quality of the rendered images could be improved by any number of changes to our very simple decoder (a multi-layer perceptron), but hopefully it is clear that the scene encodings are at some level capturing the conjunction of position with other object features.

Figure 3.15: Results on local rotation for a held-out test set containing MNIST digits

We had difficulty training the model in an unsupervised way on the more complex 3D scenes shown in Figure 3.16. This is the setup that requires making alternating updates to

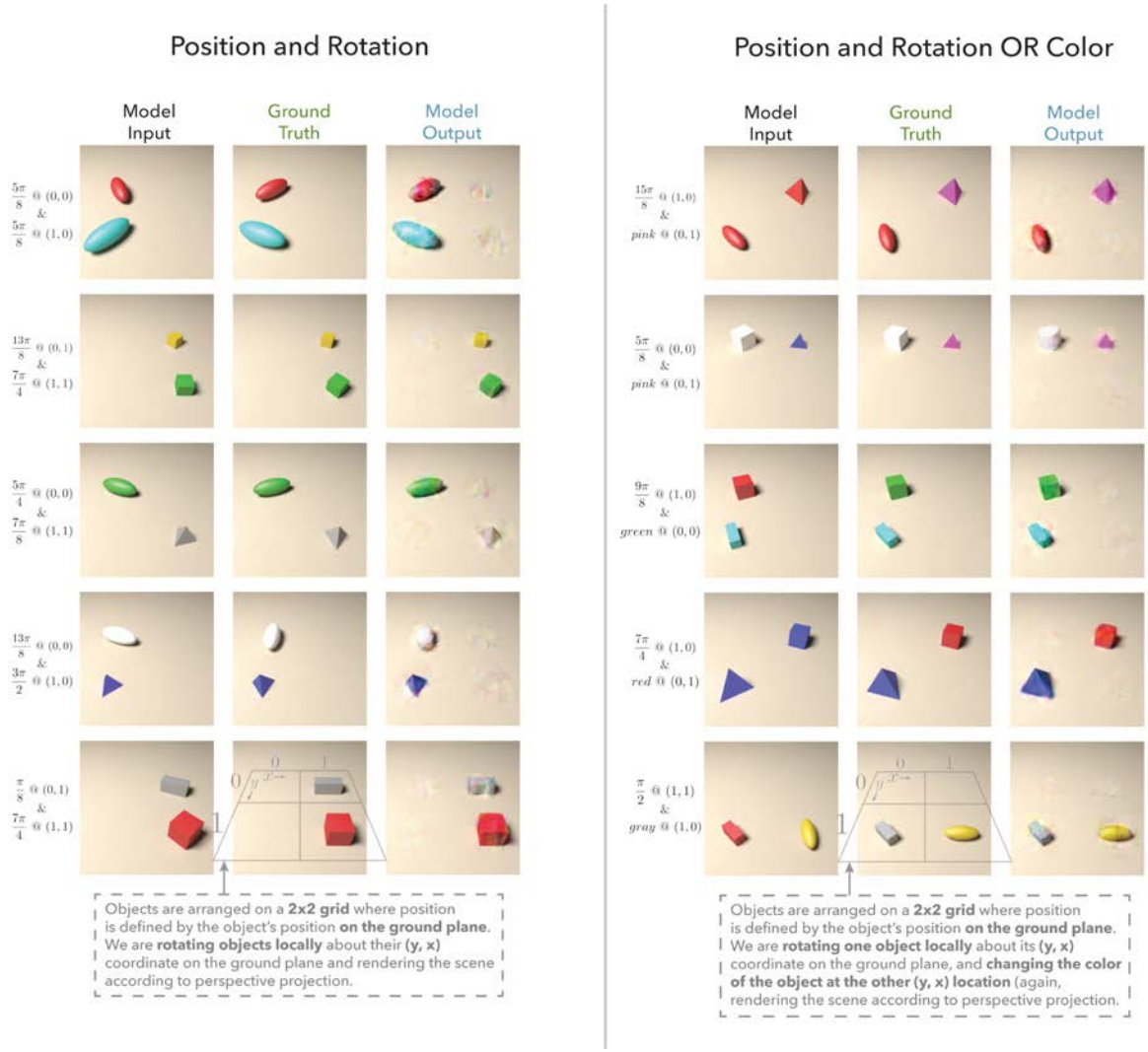


Figure 3.16: Random sample of outputs from VSST model applied to 3D rendered scenes

the model parameters ("learning"), and regressing the rotations and positions ("inference"). The results shown are for when the model is given what the transformations are as a label. It encodes these into VSA vectors and then applies VSA arithmetic as we have shown above in (3.14). We are not sure specifically why unsupervised learning was difficult here, but it might have to do simply with collapsing many degrees of freedom (the rotation vectors are N-dimensional) onto a single scalar, and the general difficulty of bad local minima in this 1D subspace.

3.4 Sub-symbolic superposition

The best way to map from images to a rich symbolic vector space is unknown, but in the previous sections we have outlined several different approaches. On the one hand, the mapping can be learned with essentially a "black box" encoder, provided strict enough supervision of the vector outputs is used. This can either be with the desired ground truth representations (Section 3.2.1) or by transforming the vector outputs using VSA algebra while supervising the reconstructions in the image domain (Section 3.3). On the other hand, one can explicitly generate a VSA encoding of every single pixel and directly superimpose these (Section 3.2.2). This has some interesting properties–it effectively asks each pixel to "vote" for higher-level object features, and in a high dimensional space this superposition collects these votes into what tend to act like discrete categories. One very nice consequence of allowing each pixel a vote is that it deals automatically with multiple objects in a scene. The superposition ends up congealing into k clusters of votes–one for each of the k objects. Therefore the output encodes a superposition of objects without having to do any extra work.

The next logical evolution of this idea applies superposition to higher-level feature conjunctions. I call it an sub-symbolic superposition encoder. A diagram of a sub-symbolic superposition encoder is given in Figure 3.17, showing that it has 4 simple stages: feature extraction, factor inference, factor combination, and superposition.

Feature extraction

The feature extraction stage could be any variety of differentiable layers–we have used standard convolutions with rectified linear unit activations. In most convolutional networks, significant spatial downsampling of the image would occur here, but one should consider whether this is appropriate for tasks that require positional encoding and/or fine spatial resolution. In our experiments we have discretely sampled position in the dataset so as to allow a 4x downsampling (of an originally 160x160 pixels image) simply to reduce the number of trainable features and the size of the activation layers.

Factor inference

Factor inference branches into F factors. In each of these branches it selects, at every position (y, x) in the feature map, a linear combination of the codevectors for this factor. The factors can be provided ahead of time or they can be learned online. In most of our experiments we provide the vectors, which are often drawn randomly. Most of our experience training this model so far has been with unitary complex vectors (Plate, 2003). More concretely, the computation of factor inference is:

$$\mathbf{X}_f \mathbf{a}_f[y, x] = \hat{\mathbf{x}}_f[y, x] \quad \forall f = 1, 2, \ldots, F \tag{3.16}$$

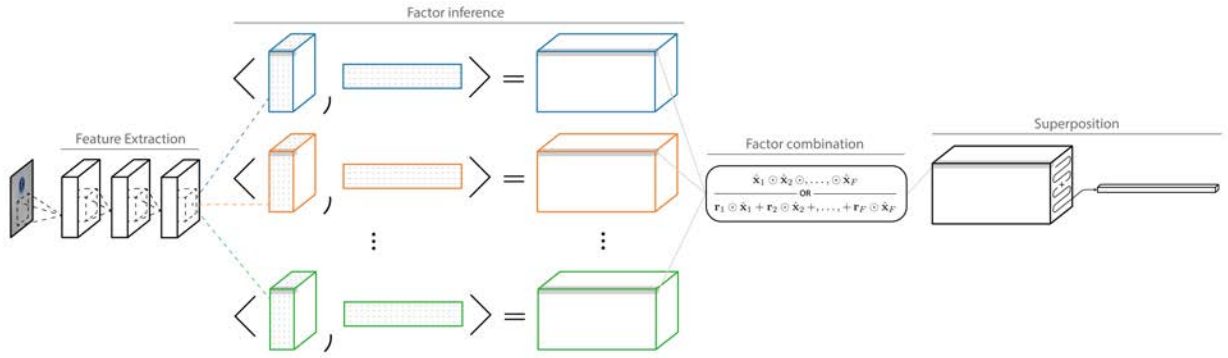


Figure 3.17: Sub-symbolic superposition encoder

Factor combination

Factor combination is either a product of fillers or a sum of role-filler pairs encoding at each position (y, x) position:

$$\hat{\mathbf{c}}[y,x] = \hat{\mathbf{x}}_1[y,x] \odot \hat{\mathbf{x}}_2[y,x] \odot, \ldots, \odot \hat{\mathbf{x}}_F[y,x] \tag{3.17}$$

$$\hat{\mathbf{c}}[y,x] = \mathbf{r}_1 \odot \hat{\mathbf{x}}_1[y,x] + \mathbf{r}_2 \odot \hat{\mathbf{x}}_2[y,x] +, \ldots, + \mathbf{r}_F \odot \hat{\mathbf{x}}_F[y,x] \tag{3.18}$$

Superposition

Superposition is the sum over each position of the (ostensibly sub-symbolic) prod_f or sum_rfp vectors:

$$\hat{\mathbf{s}} = \sum_{y,x} \hat{\mathbf{c}}[y,x] \tag{3.19}$$

This encoder is fairly easy to train, although the memory requirements of representing the tensors in the factor inference step can be a challenge. We have trained this mostly on a custom variant of the CLEVR images dataset (Johnson et al., 2017). The main point of emphasis we would like to make about this encoder is that it naturally handles multiple objects in superposition. We trained the model on images containing one object, while keeping track of its performance on images with two objects. Compared to a baseline we refer to as "black-box" (it generates HD vector encodings via an MLP mapping from the convolutional features down to a vector output), our encoder shows much better generalization. We show this in Figure 3.18.

3.5 Analogical Reasoning

Psychologists have studied analogy for decades because, at some level, it captures what is most powerful about human cognition. From among the many definitions given for analogy,

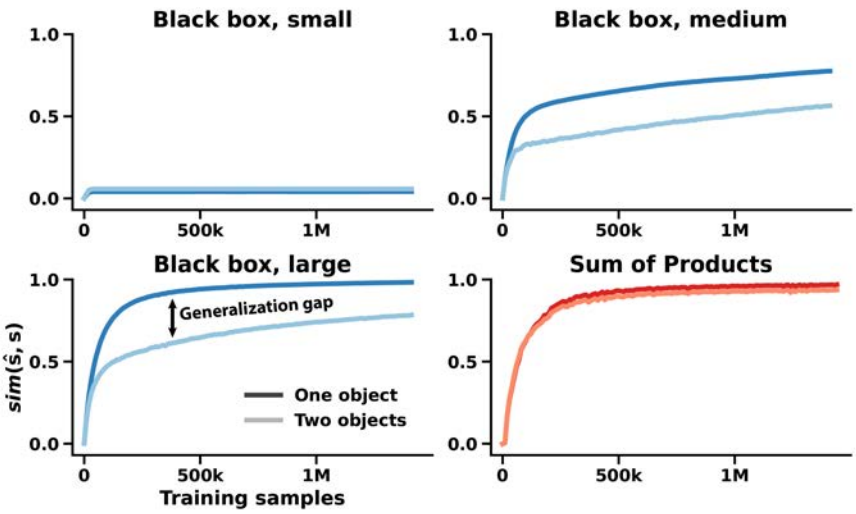


Figure 3.18: Generalization to multiple objects comes from the architecture of sub-symbolic superposition. Black box variants vary by the number of trainable parameters.

let us sythesize our own: *analogical reasoning* is a cognitive process that infers similarity and correspondence between different cognitive concepts. An *analogy* is whatever correspondence can be found. Clearly, certain concepts can be more or less analogous, and while we know some things about the types of analogies that humans prefer (Ross, 1989; Gentner et al., 1993), we are still only guessing about the computational and neural basis of human analogical reasoning.

What makes analogy so powerful is that it gives one a framework for repurposing learned knowledge in situtations that have never been encountered. Sometimes these novel cognitive concepts come from completely different domains of knowledge than what has been previously learned, making them appear, based only on their surface features, novel. After further inspection, however, analogies exist precisely where one can draw structural correspondences between elements of the learned and novel concepts. In some sense, and this is a term often employed by psychologists in this context, an analogy involves an *isomorphism* between concepts.

That the human facility for analogical reasoning helps with learning new information and categorizing it for later recall should be fairly obvious. Think about how one typically learns a second language. It is by analogy to concepts and words in one's own fluent language that novel concepts and words are introduced. Physics and mathematics textbooks are brimming with analogy. The arrangement of subatomic particles in an atom is often explained in analogy to the orbits of planets in our solar system. Consider the frequent appeal to geometric intuition that permeates much of mathematics. Perhaps it is not too bold to say (as has been argued by some cognitive scientists, see, e.g., Hofstadter and Sander (2013)) that cognition *is* analogical reasoning.

It would appear that all but the most basic analogies require a facility for representing

$\texttt{Name} \sim \mathbf{r}_{\texttt{Name}}$	$\texttt{Language} \sim \mathbf{r}_{\texttt{Language}}$	$\texttt{Currency} \sim \mathbf{r}_{\texttt{Currency}}$
$\texttt{USA} \sim \mathbf{n}_{\texttt{USA}}$	$\texttt{Mex} \sim \mathbf{n}_{\texttt{Mex}}$	
$\texttt{English} \sim \mathbf{a}_{\texttt{English}}$	$\texttt{Spanish} \sim \mathbf{a}_{\texttt{Spanish}}$	
$\texttt{Dollar} \sim \mathbf{c}_{\texttt{Dollar}}$	$\texttt{Pound} \sim \mathbf{c}_{\texttt{Pound}}$	$\texttt{Peso} \sim \mathbf{c}_{\texttt{Peso}}$

Table 3.3: Vector encodings of atomic concepts

and manipulating data structures, as defined in a general sense. The most basic structural requirement of analogy is an association of *variables* with instance-specific *values*, otherwise known as role-filler bindings. While various authors have suggested that these associations be made in other ways (Thagard et al., 1990; Forbus et al., 1995), we suggest that the VSA binding operation is an elegant way to do it, and in the preceeding chapters we have sketched out a more general framework for building *VSA data structures*.

Analogical reasoning therefore appears in this thesis as an application of the multiplicative distributed codes found in VSA data structures. We show how one can encode concepts in an analogical reasoning problem. We will also show preliminary results on training a sub-symbolic superposition encoder on analogy problems.

3.5.1 Encoding considerations

We will start with an example of analogical reasoning that is simple, abstract, and may seem slightly afield from our prior experiments with visual scenes, but is relevant to VSAs in a more general sense. Suppose one describes a country by its language, name, and currency. Clearly much more than these properties goes into the cognitive concept of a country but allow us to simplify things for the sake of illustration. We consider 3 different countries:

$$\texttt{United States} = \{(\texttt{Name}, \texttt{USA}), (\texttt{Language}, \texttt{English}), (\texttt{Currency}, \texttt{Dollar})\} \tag{3.20}$$

$$\texttt{Thirteen Colonies} = \{(\texttt{Name}, \texttt{USA}), (\texttt{Language}, \texttt{English}), (\texttt{Currency}, \texttt{Pound})\} \tag{3.21}$$

$$\texttt{Mexico} = \{(\texttt{Name}, \texttt{Mex}), (\texttt{Language}, \texttt{Spanish}), (\texttt{Currency}, \texttt{Peso})\} \tag{3.22}$$

Here we have three compound cognitive concepts, `United States`, `Thirteen Colonies`, and `Mexico`, built from other cognitive concepts for languange, currency, etc., all of which we assume to be atomic. Let us encode each atomic concept into a random high-dimensional vector as shown in Table 3.3, where we mean by $\texttt{English} \sim \mathbf{a}_{\texttt{English}}$ that the concept `English` is encoded into a vector $\mathbf{a}_{\texttt{English}}$.

The key question in this section is how to combine these atomic vectors to produce a representation for each country. We start with what has been most commonly used in the history of VSAs–a superposition of role-filler pairs.

3.5.2 Superposition of role-filler pairs (sum_rfp)

A superposition of role-filler pairs represents `United States` in the following way:

$$\texttt{United States} \sim \mathbf{d}_{\texttt{US}} = \mathbf{r}_{\texttt{Name}} \otimes \mathbf{n}_{\texttt{USA}} \oplus \mathbf{r}_{\texttt{Language}} \otimes \mathbf{a}_{\texttt{English}} \oplus \mathbf{r}_{\texttt{Currency}} \otimes \mathbf{c}_{\texttt{Dollar}}$$

`Mexico` is encoded by

$$\texttt{Mexico} \sim \mathbf{d}_{\texttt{Mexico}} = \mathbf{r}_{\texttt{Name}} \otimes \mathbf{n}_{\texttt{Mex}} \oplus \mathbf{r}_{\texttt{Language}} \otimes \mathbf{a}_{\texttt{Spanish}} \oplus \mathbf{r}_{\texttt{Currency}} \otimes \mathbf{c}_{\texttt{Peso}}$$

One can form $\mathbf{d}_{\texttt{US}}^{-1} \otimes \mathbf{d}_{\texttt{Mexico}}$ to find all the relevant *correspondences* between these two concepts:

$$\mathbf{d}_{\texttt{US}}^{-1} \otimes \mathbf{d}_{\texttt{Mexico}} = \mathbf{n}_{\texttt{USA}}^{-1} \otimes \mathbf{n}_{\texttt{Mex}} \oplus \mathbf{a}_{\texttt{English}}^{-1} \otimes \mathbf{a}_{\texttt{Spanish}} \oplus \mathbf{c}_{\texttt{Dollar}}^{-1} \otimes \mathbf{c}_{\texttt{Peso}} \oplus \boldsymbol{\eta}_1 \tag{3.23}$$

where $\boldsymbol{\eta}_1$ collects the 6 other cross-terms, each of which is a product of 4 atomic vectors. This implements Genter's structure mapping idea, because it pairs up each of the concepts that fill the same role (Gentner, 1983). One can query this with the representation of `Dollar` and get back (a noisy version of) the representation of `Peso`:

$$\mathbf{c}_{\texttt{Dollar}} \otimes \mathbf{d}_{\texttt{US}}^{-1} \otimes \mathbf{d}_{\texttt{Mexico}} = \mathbf{c}_{\texttt{Peso}} \oplus \boldsymbol{\eta}_2 \tag{3.24}$$

where $\boldsymbol{\eta}_2$ is analogous, but different from $\boldsymbol{\eta}_1$ (and generally I will take $\boldsymbol{\eta}_x$ to be different noise terms). This has been the method of VSA analogy works in the past (Kanerva, 2010).

There is an issue with using the multiplicative inverse on sum_rfp encodings. When the concepts to be compared share one or more role-filler pairs, this leaves one or more multiplicative identity vectors in superposition. We illustrate by encoding `Thirteen Colonies`:

$$\texttt{Thirteen Colonies} \sim \mathbf{d}_{\texttt{13c}} = \mathbf{r}_{\texttt{Name}} \otimes \mathbf{n}_{\texttt{USA}} \oplus \mathbf{r}_{\texttt{Language}} \otimes \mathbf{a}_{\texttt{English}} \oplus \mathbf{r}_{\texttt{Currency}} \otimes \mathbf{c}_{\texttt{Pound}}$$

The result of $\mathbf{d}_{\texttt{US}}^{-1} \otimes \mathbf{d}_{\texttt{13c}}$ is

$$\mathbf{d}_{\texttt{US}}^{-1} \otimes \mathbf{d}_{\texttt{13c}} = \mathbf{c}_{\texttt{Dollar}}^{-1} \otimes \mathbf{c}_{\texttt{Pound}} \oplus \mathbf{I} \oplus \mathbf{I} \oplus \boldsymbol{\eta}_3 \tag{3.25}$$

These multiplicative identity vectors turn out to be problematic because forming the same type of query before, $\mathbf{c}_{\texttt{Dollar}} \otimes \mathbf{d}_{\texttt{US}}^{-1} \otimes \mathbf{d}_{\texttt{13c}}$ does not yield $\mathbf{c}_{\texttt{Pound}}$, as we desire. It might be possible to infer from $\mathbf{d}_{\texttt{US}}$ and $\mathbf{d}_{\texttt{13c}}$ on how many role-filler pairs they match, and then subtract this many copies of $\mathbf{I}$ from $\mathbf{d}_{\texttt{US}}^{-1} \otimes \mathbf{d}_{\texttt{13c}}$ before multiplying by $\mathbf{c}_{\texttt{Dollar}}$. However, this is not entirely straightforward and a better solution exists.

Rather than using the binding operation on sum_rfp encodings, one should use vector addition and subtraction. This still captures all the correspondences:

$$\begin{aligned} \mathbf{d}_{\texttt{Mexico}} \ominus \mathbf{d}_{\texttt{US}} \;=\; & \mathbf{r}_{\texttt{Name}} \otimes (\mathbf{n}_{\texttt{Mex}} \ominus \mathbf{n}_{\texttt{USA}}) \\ & \oplus \mathbf{r}_{\texttt{Language}} \otimes (\mathbf{a}_{\texttt{Spanish}} \ominus \mathbf{a}_{\texttt{English}}) \\ & \oplus \mathbf{r}_{\texttt{Currency}} \otimes (\mathbf{c}_{\texttt{Peso}} \ominus \mathbf{c}_{\texttt{Dollar}}) \end{aligned} \tag{3.26}$$

$$\mathbf{d}_{\texttt{13c}} \ominus \mathbf{d}_{\texttt{US}} \;=\; \mathbf{r}_{\texttt{Currency}} \otimes (\mathbf{c}_{\texttt{Pound}} \ominus \mathbf{c}_{\texttt{Dollar}}) \tag{3.27}$$

Where $\ominus$ is just equivalent to adding by the *additive* inverse in the VSA algebra. From above we can see that correspondences between the different fillers are still maintained, thanks to the role vectors. This vector can be queried like so:

$$\mathbf{c}_{\texttt{Dollar}} \oplus \mathbf{r}^{-1}_{\texttt{Currency}} \otimes \left(\mathbf{d}_{\texttt{Mexico}} \ominus \mathbf{d}_{\texttt{US}}\right) = \mathbf{c}_{\texttt{Peso}} + \boldsymbol{\eta}_4 \tag{3.28}$$

This works for any country, regardless of how many role-filler pairs they share in common. The idea here is that, because the method of combination for role-fill pairs was summation, the operation that captures relations should be summation/subtraction. Otherwise unnnecessary cross-terms are introduced that affect the recovery of atomic concepts. In previous deep learning works that have computed vector sums and differences to capture analogy (Mikolov et al., 2013; Pennington et al., 2014), it has not been convincingly argued *why* sums and differences are the appropriate operations for analogy. For the sum_rfp encoding this is significantly more clear.

3.5.3 Product of fillers (prod_f)

The product of fillers encoding is:

$$\texttt{United States} \sim \mathbf{d}_{\texttt{US}} = \mathbf{n}_{\texttt{USA}} \otimes \mathbf{a}_{\texttt{English}} \otimes \mathbf{c}_{\texttt{Dollar}}$$

$$\texttt{Thirteen Colonies} \sim \mathbf{d}_{\texttt{13c}} = \mathbf{n}_{\texttt{USA}} \otimes \mathbf{a}_{\texttt{English}} \otimes \mathbf{c}_{\texttt{Pound}}$$

$$\texttt{Mexico} \sim \mathbf{d}_{\texttt{Mexico}} = \mathbf{n}_{\texttt{Mex}} \otimes \mathbf{a}_{\texttt{Spanish}} \otimes \mathbf{c}_{\texttt{Peso}}$$

We have dispensed with role vectors entirely. This makes decoding a country's VSA representation harder than in sum_rfp, unless of course one has a Resonator Network. The benefit of doing it this way is the following:

$$\mathbf{d}^{-1}_{\texttt{US}} \otimes \mathbf{d}_{\texttt{13c}} = \mathbf{c}^{-1}_{\texttt{Dollar}} \otimes \mathbf{c}_{\texttt{Pound}} \tag{3.29}$$

There are no cross-terms here. If the binding operation used is not exactly invertible (e.g. circular convolution in real-valued HRRs) then there will be a small amount of noise still, but it is not of the magnitude represented by $\boldsymbol{\eta}_1$ to $\boldsymbol{\eta}_4$ above (Plate, 1994). This might be very helpful depending on the application and how many atomic vectors need to be decoded. Plate (1994) established (and Frady, Kleyko, et al. (2018) confirmed) that the cross-terms contained in $\boldsymbol{\eta}_1$ to $\boldsymbol{\eta}_4$ are actually fairly problematic for VSA decoding capacity. In particular, the number of neurons N has to grow at a rate linear in the number of cross terms in order to keep up with this noise, which is not the type of scaling (i.e. logarithmic) we usually benefit from in high dimensions. The prod_f encoding tries to avoid putting things in superposition whenever possible.

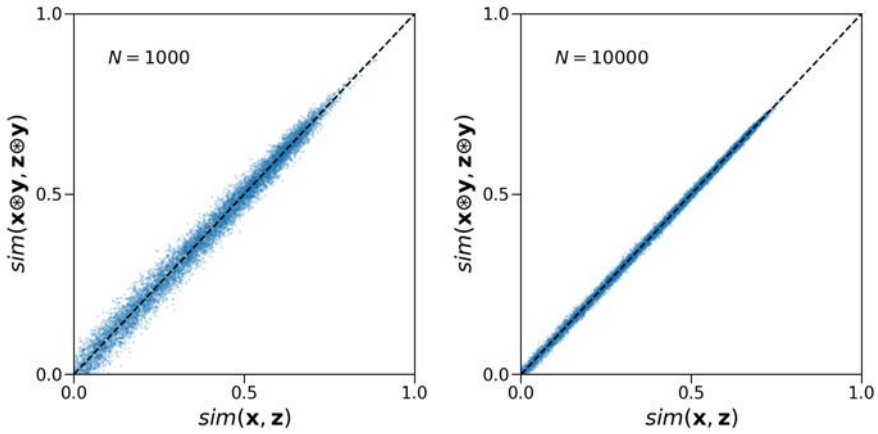


Figure 3.19: Similarity-preservation of binding for real-valued HRRs

Similarity-preservation of binding

Given that we are advocating the binding operation of a VSA to capture relations between things in an analogical reasoning setting, we would like to highlight a propery of this operation which may bear on how appropriate it is from a *geometric* perspective.

The binding operator $\otimes$ produces a vector dissimilar to each input, which Plate has called its "randomizing" property (Plate, 1994). This is important because it otherwise would not represent a *conjunction* of two concepts so much as it would represent a *union*. Researchers have sometimes (ourselves included) refered to this as "destroying" similarity.

However, the binding operation *preserves* similarity in the following sense: $\mathbf{x} \approx \mathbf{z} \iff \mathbf{x} \otimes \mathbf{y} \approx \mathbf{z} \otimes \mathbf{y}$. For several VSAs, namely MAP and "Unitary" HRRs, the similarity preservation is *exact*. One geometric intuition for what is happening is that by binding it with $\mathbf{y}$, $\mathbf{x}$ gets mapped into a completely different part of the space, but this *preserves* the relative similarity structure among all the vectors exposed to this mapping (such as $\mathbf{z}$). This is a fairly remarkable fact. It means that we may hope to encode analogies for which the source domain is very different from the target domain, but where the key relations, captured by relative similarity structure, remain intact. We show a simulation of this phenomenon for real-valued HRRs in Figure (3.19). Notice that the similarity preservation for these vectors is "noisy" but that the variance of this noise appears to decrease with increasing vector dimensionality. Plate derived expressions for this variance in his thesis (Plate, 1994), finding that it scales as $\mathcal{O}(N^{-1})$.

3.5.4 Analogical reasoning with a learned encoding

We have done some experiments with using a sub-symbolic superposition encoder (Section 3.4) in conjunction with VSA arithmetic to capture visual analogies. We have run these experiments on a customized version of the CLEVR dataset (Johnson et al., 2017) and use complex-valued HRRs. Suppose the analogy is defined by `A : B :: C : D` which indicates

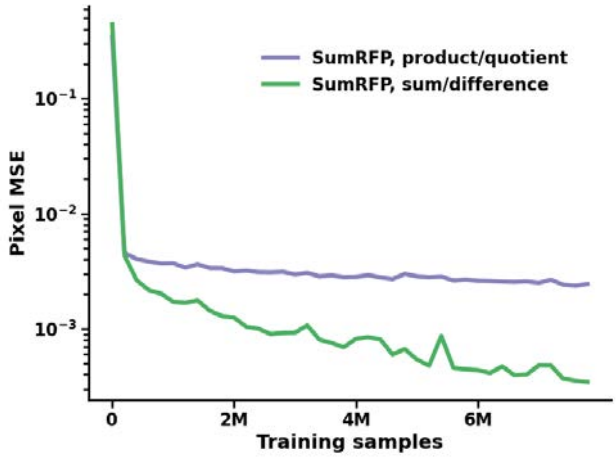


Figure 3.20: Using two different types of arithmetic on the sum_rfp encoding

that A and B have a relation that is shared by C and D. Following the example of Reed et al. (2015), we pass three images (A, B, and C) through the encoder, then we do vector arithmetic on them to generate a represenation of D, and then we render this image with a deconvolutional network. Let us first note evidence for the importance of *matching the arithmetic* to the encoding. The sum/difference way of encoding this analogy is $\mathbf{s}_\mathtt{B} \ominus \mathbf{s}_\mathtt{A} \oplus \mathbf{s}_\mathtt{C}$ while the product/quotient way is $\mathbf{s}_\mathtt{B} \otimes \mathbf{s}_\mathtt{A}^{-1} \otimes \mathbf{s}_\mathtt{C}$. What is shown by Figure 3.20 is that the sum/difference arithmetic is better for the sum of role-filler pairs encoding, as we suggested in the previous section. While we do not show the reconstructed images, the subjective quality of the reconstructions for the sum/difference variant is much better, actually capturing the different factors of variation in the dataset, which is lost for the product/quotient arithmetic.

We tend to prefer the product of fillers encoding for its noise properties, but it is not yet clear which is best in general. The fact that most deep learning works have had success capturing analogy with additive and subtractive arithmetic, rather than multiplicative, may give evidence for the sum_rfp encoding being easier to learn with gradient descent.

We compared our VSA analogy pipeline with the deep learning model from Reed et al. (2015). What we found was that our approach, which uses many fewer trainable parameters, was able to generate significantly better image analogies. In particular, the structure imposed by our sub-symbolic superposition encoder regularizes the latent scene encodings so that they actually capture the geometry of the analogy ($\mathbf{s}_\mathtt{B} \otimes \mathbf{s}_\mathtt{A}^{-1} \otimes \mathbf{s}_\mathtt{C} \approx \mathbf{s}_\mathtt{D}$). This is shown in Figure 3.21.

3.6 Summary

This chapter has covered some applications of Vector Symbolic Architectures and Resonator Networks. Ultimately it has been about composing and decomposing VSA data structures that represent visual scenes. We have seen that mappings can be learned from the space of

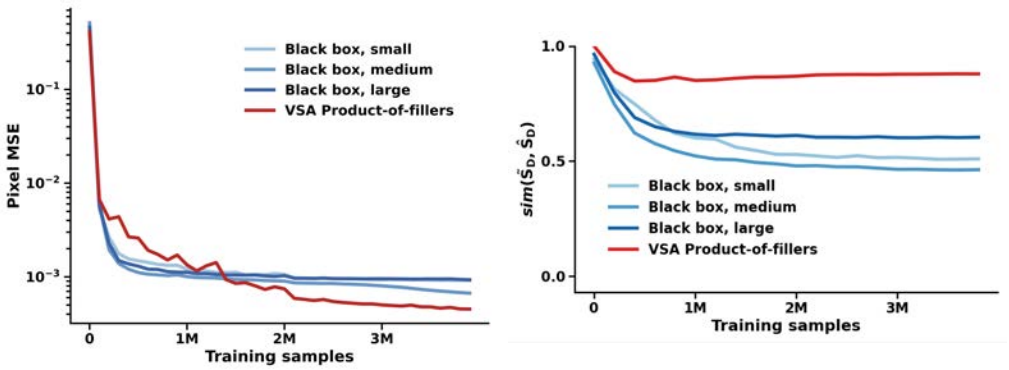


Figure 3.21: Training an end-to-end analogical reasoning pipeline à la Reed et al. (2015) works better with a sub-symbolic superposition encoder and VSA arithmetic.

images into the space of symbolic vectors, and that these representations have some desirable properties. We have used the binding operator of VSAs to capture conjuctions of features, and Resonator Networks to decompose these conjunctions. Such an approach of explicitly coding multiplicative interactions helps one to get a handle on the combinatorics of visual scenes. We outlined a VSA encoding that elegantly handles real numbers and may be critical to scaling these applications. We also showed how transformation is another way to impose certain structure on a latent VSA representation. Analogical reasoning is in some ways the ultimate test of any structure-embedding theory, and we have just scratched the surface here. We do think that explicitness about encoded structure, which is afforded by the VSA perspective, is an interesting view on analogy and may help to demystify some of the prior analogical reasoning work from deep learning.

At some level, we have been asking our various encoders to solve the factorization problem for us, before ever involving a Resonator Network. One could view this as recoding the image, in which factors are more easily discoverable, but collide more easily, into a VSA representation that is more portable and supports downstream reasoning. Ultimately, our objective is to integrate the encoding more tightly with Resonator Network dynamics, but this is a challenge for another day.

Appendix A

Appendix to chapter 2

A.1 Implementation details

This section includes a few comments relevant to the implementation of Resonator Networks. Algorithm 1 gives psuedocode for Ordinary Least Squares weights–the only change for outer product weights is to use $\mathbf{X}^\top$ instead of $\mathbf{X}^\dagger$. So long as $D_f < N/2$, computing $\mathbf{X}_f\mathbf{X}_f^\dagger\big(\hat{\mathbf{o}} \odot \mathbf{c}\big)$ has lower computational complexity than actually forming a single "synaptic matrix" $\mathbf{T}_f := \mathbf{X}_f\mathbf{X}_f^\dagger$ and then computing $\mathbf{T}_f\big(\hat{\mathbf{o}} \odot \mathbf{c}\big)$ in each iteration–it is faster to keep the matrices $\mathbf{X}_f$ and $\mathbf{X}_f^\dagger$ separate. This, of course, assumes that implementation is on a conventional computer. If one can use specialized analog computation, such as large mesh circuits that directly implement matrix-vector multiplication in linear time (Cannon, 1969), then it would be preferable to store the synaptic matrix directly.

Lines 11 - 13 in Algorithm 1 "clean up" $\hat{\mathbf{x}}^{(f)}$ using the nearest neighbor in the codebook, and also resolve a sign ambiguity inherent to the factorization problem. The sign ambiguity is simply this: while $\mathbf{c} = \mathbf{x}_\star^{(1)} \odot \mathbf{x}_\star^{(2)} \odot \ldots \odot \mathbf{x}_\star^{(F)}$ is the factorization we are searching for, we also have $\mathbf{c} = -\mathbf{x}_\star^{(1)} \odot -\mathbf{x}_\star^{(2)} \odot \ldots \odot \mathbf{x}_\star^{(F)}$, and, more generally, any *even* number of factors can have their signs flipped but still generate the correct $\mathbf{c}$. Resonator Networks will sometimes find these solutions. We clean up using the codevector with the largest *unsigned* similarity to the converged $\hat{\mathbf{x}}^{(f)}$, which remedies this issue. One will notice that we have written Algorithm 1 to update factors in order from 1 to F. This is completely arbitrary, and any ordering is fine. We have experimented with choosing a random update order during each iteration, but this did not seem to significantly affect performance.

Computing $\hat{\mathbf{o}}$ with the most-recently updated values for factors 1 to $f-1$ (see equation (2.6)) is a convention we call 'asynchronous' updates, in rough analogy to the same term used in the context of Hopfield Networks. An alternative convention is to, when computing $\hat{\mathbf{o}}$, not used freshly updated values for factors 1 to $f-1$, but rather their values before the update. This treats each factor as if it is being updated simultaneously, a convention we call 'synchronous' updates. This distinction is an idiosyncrasy of modeling Resonator Networks in discrete-time, and the difference between the two disappears in continuous-time, where

things happens instantaneously. Throughout this chapter, our analysis and simulations have been with 'asynchronous' updates, which we find to converge significantly faster.

Not shown in Algorithm 1 is the fact that, in practice, we record a buffer of past states, allowing us to detect when the dynamics fall into a limit cycle, and to terminate early.

Algorithm 1 Resonator Network with Ordinary Least Squares weights

Require: $\mathbf{c}$ ▷ Composite vector to be factored
Require: $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_F$ ▷ Codebook matrices $\left(\mathbf{x}_j^{(f)} = \mathbf{X}_f[\,:, j\,]\right)$
Require: k ▷ Maximum allowed iterations

$\hat{\mathbf{x}}^{(f)} \leftarrow \operatorname{sgn}\left(\sum_j \mathbf{x}_j^{(f)}\right) \quad \forall f = 1, \ldots, F$
$\mathbf{X}_f^{\dagger} \leftarrow \operatorname{pinv}(\mathbf{X}_f) \quad \forall f = 1, \ldots, F$
$i \leftarrow 0$
while not converged **and** $i < k$ **do**
  for $f = 1$ **to** F **do**
    $\hat{\mathbf{o}} \leftarrow \hat{\mathbf{x}}^{(1)} \odot \ldots \odot \hat{\mathbf{x}}^{(f-1)} \odot \hat{\mathbf{x}}^{(f+1)} \odot \ldots \odot \hat{\mathbf{x}}^{(F)}$
    $\hat{\mathbf{x}}^{(f)} \leftarrow \operatorname{sgn}\Big(\mathbf{X}_f \mathbf{X}_f^{\dagger}\big(\hat{\mathbf{o}} \odot \mathbf{c}\big)\Big)$
  end for
  $i \leftarrow i + 1$
end while
for $f = 1$ **to** F **do** ▷ Nearest Neighbor decoding
  $u \leftarrow \arg\max_j |\operatorname{sim}(\hat{\mathbf{x}}^{(f)}, \mathbf{x}_j^{(f)})|$ ▷ *Un-signed* NN w.r.t cos-similarity
  $\hat{\mathbf{x}}^{(f)} \leftarrow \mathbf{x}_u^{(f)}$
end for
return $\hat{\mathbf{x}}^{(f)} \; \forall f = 1, \ldots, F$

A.2 Operational Capacity

The main text of Chapter 2 introduced our definition of operational capacity and highlighted our two main results–that Resonator Networks have superior operational capacity compared to the benchmark algorithms, and that Resonator Network capacity scales as a quadratic function of N. This appendix provides some additional support and commentary on these findings.

Figure A.1 compares operational capacity among all of the considered algorithms when F, the number of factors, is 4. We previously showed this type of plot for $F = 3$, which was Figure 2.3 in the main text. Resonator Networks have an advantage of between two and three orders of magnitude compared to all of our benchmarks; the general size of this gap was consistent in all of our simulations.

We concluded in Section 2.5.2 that the operational capacity of Resonator Networks scales quadratically in N, which was shown in Figure 2.4. In Table A.1 we provide parameters of the

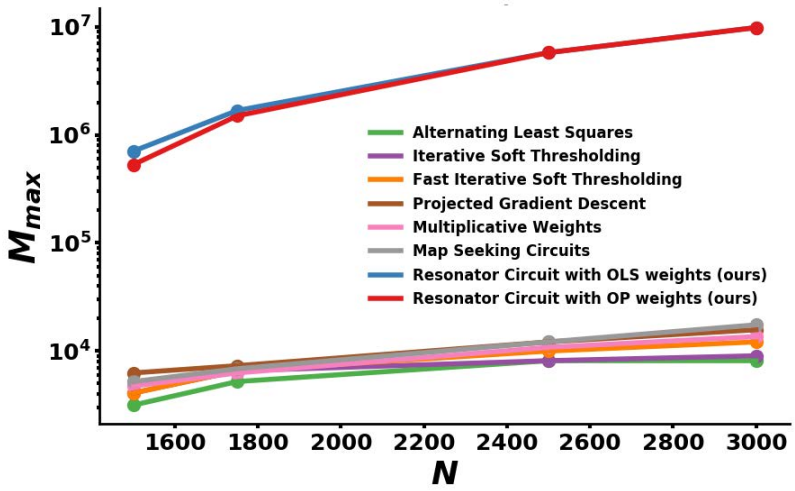


Figure A.1: Comparing operational capacity against the benchmarks for $F = 4$ (4 factors)

least-squares quadratic fits shown in that plot. One can see from Figure 2.4b that capacity is different depending on the number of factors involved, and in the limit of large N this difference is determined by the parameter c. c first rises from 2 to 3 factors, and then falls with increasing F. This implies that factorization is easiest for Resonator Networks when the decomposition is into 3 factors, an interesting phenomenon for which we do not have an explanation at this time.

Figure A.2 visualizes c as a function of F. The data indicates that for $F \geq 3$, c may follow an inverse power law: $c = \alpha_1 F^{-\alpha_2}$. The indicated linear fit, following a logarithmic transformation to each axis, suggests the following values for parameters of this power law: $\alpha_1 \approx 2^{3.014} = 8.078$, $\alpha_2 \approx 1.268$. It is with some reservation that we give these specific values for α_1 and α_2. Our estimates of operational capacity, while well-fit by quadratics, undoubtedly have small amounts of noise. This noise can have a big enough impact on fitted values for c that *fitting the fit* may not be fully justified. However, we do note for the sake of completeness that this scaling, if it holds for larger values of F, would allow us to write operational capacity in terms of both parameters N and F in the limit of large N:

$$M_{max} \approx \frac{8.078\, N^2}{F^{1.268}} \quad \forall F \geq 3 \tag{A.1}$$

F	Parameters of quadratic fit		
	a	b	c
2	1.677×10^5	-3.253×10^2	0.293
3	1.230×10^6	-3.549×10^3	2.002
4	-5.663×10^6	9.961×10^2	1.404
5	1.140×10^6	-2.404×10^3	1.024
6	5.789×10^6	-4.351×10^3	0.874
7	-1.503×10^7	-1.551×10^3	0.669

Table A.1: $M_{max} = a + bN + cN^2$

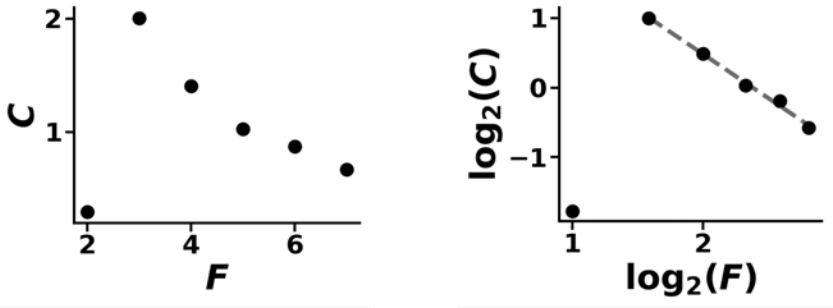


Figure A.2: Parameter c of the quadratic scaling depends on F. We find that it may follow an inverse power law for $F \geq 3$.

A.3 Table of benchmark algorithms

Algorithm	**Dynamics for updating $\mathbf{a}_f[t]$**	**Eq.**
Alternating Least Squares	$\mathbf{a}_f[t+1] = \big(\boldsymbol{\xi}^\top \boldsymbol{\xi}\big)^{-1} \boldsymbol{\xi}^\top \mathbf{c}$ $\boldsymbol{\xi} := \operatorname{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big)\, \mathbf{X}_f$	(2.10)
Iterative Soft Thresholding	$\mathbf{a}_f[t+1] = \mathcal{S}[\mathbf{a}_f[t] - \eta\, \nabla_{\mathbf{a}_f} \mathcal{L}\ ; \lambda\eta]$ $\big(\mathcal{S}[\,\mathbf{x}\,;\gamma\,]\big)_i := \operatorname{sgn}(x_i) \max(\lvert x_i\rvert - \gamma,\, 0\,)$	(A.3)
Fast Iterative Soft Thresholding	$\alpha_t = \dfrac{1+\sqrt{1+4\alpha_{t-1}^2}}{2}$ $\beta_t = \dfrac{\alpha_{t-1}-1}{\alpha_t}$ $\mathbf{p}_f[t+1] = \mathbf{a}_f[t] + \beta_t(\mathbf{a}_f[t] - \mathbf{a}_f[t-1])$ $\mathbf{a}_f[t+1] = \mathcal{S}[\mathbf{p}_f[t+1] - \eta\, \nabla_{\mathbf{p}_f} \mathcal{L}\ ; \lambda\eta]$ $\big(\mathcal{S}[\,\mathbf{x}\,;\gamma\,]\big)_i := \operatorname{sgn}(x_i) \max(\lvert x_i\rvert - \gamma,\, 0\,)$	(A.4)
Projected Gradient Descent	$\mathbf{a}_f[t+1] = \mathcal{P}_{C_f}\big[\mathbf{a}_f[t] - \eta \nabla_{\mathbf{a}_f} \mathcal{L}\big]$ $\mathcal{P}_{C_f}[\mathbf{x}] := \underset{\mathbf{z}\in C_f}{\arg\min}\, \frac{1}{2}\big\lvert\big\lvert \mathbf{x}-\mathbf{z}\big\rvert\big\rvert_2^2$	(A.5)
Multiplicative Weights	$\mathbf{w}_f[t+1] = \mathbf{w}_f[t] \odot \big(\mathbf{1} - \dfrac{\eta}{\rho} \nabla_{\mathbf{a}_f} \mathcal{L}\big)$ $\mathbf{a}_f[t+1] = \dfrac{\mathbf{w}_f[t+1]}{\sum_i w_{fi}[t+1]}$ $\rho := \max_i \big\lvert (\nabla_{\mathbf{a}_f} \mathcal{L})_i \big\rvert$	(A.6)
Map Seeking Circuits	$\mathbf{a}_f[t+1] = \mathcal{T}\Big(\mathbf{a}_f[t] - \eta\big(\mathbf{1} + \dfrac{1}{\rho} \nabla_{\mathbf{a}_f} \mathcal{L}\big)\,;\ \epsilon\ \Big)$ $\mathcal{T}\big(\mathbf{x}\,;\,\epsilon\,\big)_i := \begin{cases} x_i & \text{if } x_i \geq \epsilon \\ 0 & \text{otherwise} \end{cases}$ $\rho := \ \big\lvert \min_i (\nabla_{\mathbf{a}_f} \mathcal{L})_i \big\rvert$	(A.7)

Table A.2: Dynamics for $\mathbf{a}_f$, benchmark algorithms. (see Appendices A.4 - A.9 for discussion of each algorithm, including hyperparameters η, λ, and ϵ, as well as initial conditions).

A.4 Tensor Decompositions and Alternating Least Squares

Tensors are multidimensional arrays that generalize vectors and matrices. An Fth-order tensor has elements that can be indexed by F separate indexes–a vector is a tensor of order 1 and a matrix is a tensor of order 2. As devices for measuring multivariate time series have become more prevalent, the fact that this data can be expressed as a tensor has made the study of tensor decomposition a very popular subfield of applied mathematics. Hitchcock (Hitchcock, 1927) is often credited with originally formulating tensor decompositions, but modern tensor decomposition was popularized in the field of psychometrics by the work of Tucker (Tucker, 1966), Carroll and Chang (Carroll & Chang, 1970), and Harshman (Harshman, 1970). This section will highlight the substantial difference between tensor decomposition and the factorization problem solved by Resonator Networks.

The type of tensor decomposition most closely related to our factorization problem (given in equation (2.1)) decomposes an Fth-order tensor $\mathcal{C}$ into a sum of tensors each generated by the outer product $\circ$:

$$\mathcal{C} = \sum_{r=1}^{R} \mathbf{x}_r^{(1)} \circ \mathbf{x}_r^{(2)} \circ \ldots \circ \mathbf{x}_r^{(F)} \tag{A.8}$$

The outer product contains all pairs of components from its two arguments, so $\big(\mathbf{w} \circ \mathbf{x} \circ \mathbf{y} \circ \mathbf{z}\big)_{ijkl} = w_i x_j y_k z_l$. The interpretation is that each term in the sum is a "rank-one" tensor of order F and that $\mathcal{C}$ can be generated from the sum of R of these rank-one tensors. We say that $\mathcal{C}$ is "rank-R". This particular decomposition has at least three different names in the literature - they are Canonical Polyadic Decomposition, coined by Hitchcock, CANonical DECOMPosition (CANDECOMP), coined by Carroll and Chang, and PARAllel FACtor analysis (PARAFAC), coined by Harshman. We will simply call this the CP decomposition, in accordance with the convention used by Kolda (Kolda & Bader, 2009) and many others.

CP decomposition makes no mention of a codebook of vectors, such as we have in (2.1). In CP decomposition, the search is apparently over all of the vectors in a real-valued vector space. One very useful fact about CP decomposition is that under relatively mild conditions, *if the decomposition exists, it is unique* up to a scaling and permutation indeterminacy. Without going into the details, a result in Kruskal (1977) and extended by Sidiropoulos and Bro (2000) gives a sufficient condition for uniqueness of the CP decomposition based on what is known as the Kruskal rank $k_{\mathbf{X}_f}$ of the matrix $\mathbf{X}_f := [\mathbf{x}_1^{(f)}, \mathbf{x}_2^{(f)}, \ldots \mathbf{x}_R^{(f)}]$:

$$\sum_{f=1}^{F} k_{\mathbf{X}_f} \geq 2R + (F-1) \tag{A.9}$$

This fact of decomposition uniqueness illustrates one way that basic results from matrices fail to generalize to higher-order tensors (by higher-order we simply mean where the order is ≥ 3). Low-rank CP decomposition for matrices (tensors of order 2) may be computed

with the truncated Singular Value Decomposition (SVD). However, if $\mathcal{C}$ is a matrix and its truncated SVD is $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top := \mathbf{X}_1\mathbf{X}_2^\top$, then any non-singular matrix $\mathbf{M}$ generates an equally-good CP decomposition $(\mathbf{U}\mathbf{\Sigma}\mathbf{M})(\mathbf{V}\mathbf{M}^{-1})^\top$. The decomposition is *highly* non-unique. All matrices have an SVD, whereas generic higher-order tensors are not guaranteed to have a CP decomposition. And yet, if a CP decomposition exists, under the mild condition of equation (A.9), it is unique. This is a somewhat miraculous fact, suggesting that *in this sense, CP decompostion of higher-order tensors is easier than matrices. The higher order of the composite object imposes many more constraints that make the decomposition unique.*

Another interesting way that higher-order tensors differ from matrices is that computing matrix rank is easy, whereas in general computing tensor rank is NP-hard, along with many other important tensor problems (Hillar & Lim, 2013). Our intuition about matrices largely fails us when dealing with higher-order tensors. In some ways the problems are easier and in some ways they are harder. Please see Sidiropoulos et al. (2017) for a more comprehensive comparison.

The vector factorization problem defined by (2.1) differs from CP decomposition in three key ways:

1. The composite object to be factored is a vector, not a higher-order tensor. This is an even more extreme difference than between matrices and higher-order tensors. In CP decomposition, the arrangement and numerosity of tensor elements constitute many constraints on what the factorization can be, so much so that it resolves the uniqueness issue we outlined above. In this sense, *tensors contain much more information about the valid factorization, making the problem significantly easier.* The size and form of these tensors may make finding CP decompositions a computational challenge, but CP decomposition is analytically easier than our vector factorization problem.

2. Search is conducted over a discrete set of possible factors. This differs from the standard formulation of CP decomposition, which makes no restriction to a discrete set of factors. It is however worth noting that a specialization of CP decomposition called CANonical DEcomposition with LINear Constraints (CANDELINC) (Carroll et al., 1980) does in fact impose the additional requirement that factors are formed from a linear combination of some basis factors. In our setup the solutions are 'one-hot' linear combinations.

3. The factors are constrained to $\{-1, 1\}^N$, a small sliver of R^N. This difference should not be underestimated. We have shown in Section 2.5.6 that the interior of this hypercube is treacherous from an optimization perspective and Resonator Networks avoid it by using a highly nonlinear activation function. This would not make sense in the context of standard CP decomposition.

Perhaps the most convincing demonstration that (2.1) is *not* CP decomposition comes from the fact that we applied Alternating Least Squares to it and found that its performance was relatively poor. Alternating Least Squares is in fact the 'workhorse' algorithm of CP

decomposition (Kolda & Bader, 2009), but it cannot compete with Resonator Networks on our different factorization problem (2.1). The excellent review of Kolda and Bader (2009) covers CP decomposition and Alternating Least Squares in significant depth, including the fact that ALS always converges to a local minimum of the squared error reconstruction loss. See, in particular, section 3.4 of their paper for more detail.

One special case of CP decomposition involves rank-1 components that are *symmetric* and *orthogonal*. For this problem, a special case of ALS called the tensor power method can be used to iteratively find the best low-rank CP decomposition through what is known as 'deflation', which is identical to the explaining away we introduced in part one of this series (Frady et al., 2020). The tensor power method directly generalizes the matrix power method, and in this special case of symmetric, orthogonal tensors is effective at finding the CP decomposition. A good initial reference for the tensor power method is De Lathauwer et al. (2000b). A discussion of applying tensor decompositions to statistical learning problems is covered by Anandkumar et al. (2014), which develops a robust version of the tensor power method and contains several important probabilistic results for applying tensor decompositions to noisy data. The tensor power method differs from Resonator Networks in the same key ways as ALS–composite objects are higher-order tensors, not vectors, search is not necessarily over a discrete set, the vectors are not constrained to $\{-1, 1\}^N$, and the dynamics make *linear* least squares updates in each factor.

Another popular tensor decomposition is known as the Tucker Decomposition (Tucker, 1963, 1966). It adds to CP decomposition an order-F "core tensor" $\mathcal{G}$ that modifies the interaction between each of the factors:

$$\mathcal{C} = \sum_{p=1}^{P} \sum_{q=1}^{Q} \dots \sum_{r=1}^{R} g_{pq\dots r}\ \mathbf{x}_p^{(1)} \circ \mathbf{x}_q^{(2)} \circ \dots \circ \mathbf{x}_r^{(F)} \tag{A.10}$$

This adds many more parameters compared to CP decomposition, which is a special case of Tucker decomposition when $\mathcal{G}$ is the identity. For the purpose of illustration, we reprint in Figure A.3 (with a slight relabeling) a figure from Kolda and Bader (2009) that depicts an order-3 Tucker decomposition. This decomposition goes by many other names, most popularly the Higher-order SVD, coined in De Lathauwer et al. (2000a). The Tucker decomposition can also be found via Alternating Least Squares (see Kolda and Bader (2009), Section 4.2, for a tutorial), although the problem is somewhat harder than CP decomposition, both by being computationally more expensive and by being non-unique. Despite this fact, the applications of Tucker decomposition are wide-ranging–it has been used in psychometrics, signal processing, and computer vision. One well-known application of Tucker decomposition in computer vision was TensorFaces (Vasilescu & Terzopoulos, 2002). This model was able to factorize identity, illumination, viewpoint, and facial expression in a dataset consisting of face images.

The summary of this section is that vector factorization problem (2.1) is not tensor decomposition. In some sense it is more challenging. Perhaps not surprisingly, the standard algorithm for tensor decompositions, Alternating Least Squares, is not particularly compet-

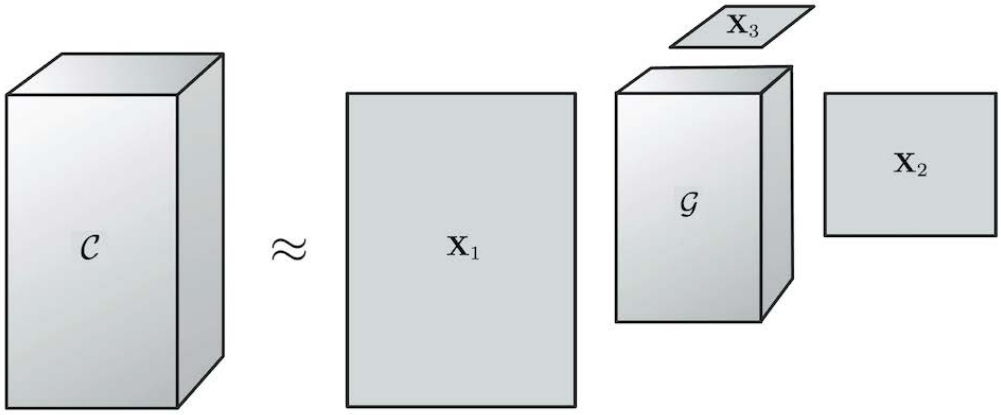


Figure A.3: Tucker decomposition with 3 factors

itive on this problem when compared to Resonator Networks. It is interesting to consider whether tensor decomposition might be cast into a form amenable to solution by Resonator Networks. Given the importance of tensor decomposition as a tool of data analysis, we believe this warrants a closer look.

A.5 General notes on gradient-based algorithms

When $\mathcal{L}$ is the negative inner product, the gradient with respect to $\mathbf{a}_f$ is:

$$\begin{aligned}\nabla_{\mathbf{a}_f}\mathcal{L} &= -\mathbf{X}_f^\top\left(\mathbf{c}\odot\hat{\mathbf{x}}^{(1)}\odot\ldots\odot\hat{\mathbf{x}}^{(f-1)}\odot\hat{\mathbf{x}}^{(f+1)}\odot\ldots\odot\hat{\mathbf{x}}^{(F)}\right)\\ &= -\mathbf{X}_f^\top\left(\mathbf{c}\odot\hat{\mathbf{o}}^{(f)}\right)\end{aligned} \tag{A.11}$$

The term $\mathbf{c}\odot\hat{\mathbf{o}}^{(f)}$ can be interpreted as an estimate for what $\hat{\mathbf{x}}^{(f)}$ should be based on the current estimates for the *other* factors. Multiplying by $\mathbf{X}_f^\top$ compares the similarity of this vector to each of the candidate codevectors we are entertaining, with the smallest element of $\nabla_{\mathbf{a}_f}\mathcal{L}$ (its value is likely to be negative with large absolute value) indicating the codevector which matches best. Following the negative gradient will cause this coefficient to increase more than the coefficients corresponding to the other codevectors. When $\mathcal{L}$ is the squared error, the gradient with respect to $\mathbf{a}_f$ is:

$$\begin{aligned}\nabla_{\mathbf{a}_f}\mathcal{L} &= \mathbf{X}_f^\top\left(\left(\mathbf{c}-\hat{\mathbf{x}}^{(1)}\odot\ldots\odot\hat{\mathbf{x}}^{(F)}\right)\odot\left(-\hat{\mathbf{x}}^{(1)}\odot\ldots\odot\hat{\mathbf{x}}^{(f-1)}\odot\hat{\mathbf{x}}^{(f+1)}\odot\ldots\odot\hat{\mathbf{x}}^{(F)}\right)\right)\\ &:= \mathbf{X}_f^\top\left(\hat{\mathbf{x}}^{(f)}\odot\left(\hat{\mathbf{o}}^{(f)}\right)^2-\mathbf{c}\odot\hat{\mathbf{o}}^{(f)}\right)\end{aligned} \tag{A.12}$$

This looks somewhat similar to the gradient for the negative inner product–they differ by an additive term given by $\mathbf{X}_f^\top\left(\hat{\mathbf{x}}^{(f)}\odot\left(\hat{\mathbf{o}}^{(f)}\right)^2\right)$. At the vertices of the hypercube all the elements

of $\hat{\mathbf{x}}^{(f)}$ are 1 or -1 and the term $\left(\hat{\mathbf{o}}^{(f)}\right)^2$ disappears, making the difference between the two gradients just $\mathbf{X}_f^\top \hat{\mathbf{x}}^{(f)}$. Among other things, this makes the gradient of the squared error equal to zero at the global minimizer $\mathbf{x}_\star^{(1)} \dots \mathbf{x}_\star^{(F)}$, which is not the case with the negative inner product. To be clear, (A.11) is the gradient when the loss function is the negative inner product, while (A.12) is the gradient when the loss function is the squared error.

A.5.1 Fixed-stepsize gradient descent on the squared error

In fixed-step-size gradient descent for unconstrained convex optimization problems, one must often add a restriction on the stepsize, related to the *smoothness* of the loss function, in order to ensure that the iterates converge to a fixed point. We say that a function $\mathcal{L}$ is L-smooth when its gradient is Lipschitz continuous with constant L:

$$||\nabla \mathcal{L}(\mathbf{x}) - \nabla \mathcal{L}(\mathbf{y})||_2 \leq L||\mathbf{x} - \mathbf{y}||_2 \;\; \forall\, \mathbf{x},\, \mathbf{y} \tag{A.13}$$

For a function that is twice-differentiable, this is equivalent to the condition

$$\mathbf{0} \preceq \nabla^2 \mathcal{L}(\mathbf{x}) \preceq L\mathbf{I} \quad \forall\, \mathbf{x} \tag{A.14}$$

Where $\mathbf{0}$ is the matrix of all zeros and and $\mathbf{I}$ is the identity. Absent some procedure for adjusting the stepsize η at each iteration to account for the degree of local smoothness, or some additional assumption we place on the loss to make sure that it is sufficiently smooth, we should be wary that convergence may not be guaranteed. On our factorization problem we find this to be an issue. Unconstrained gradient descent on the squared error works for the simplest problems, where M is small and the factorization can be easily found by any of the algorithms in this chapter. However, as M increases, the exceedingly "jagged" landscape of the squared error loss makes the iterates very sensitive to the step size η, and the components of $\mathbf{a}_f[t]$ can become very large. When this happens, the term $\hat{\mathbf{o}}^{(f)}[t]$ amplifies this problem (it multiplies all but one of the $\mathbf{a}_f[t]$'s together) and causes numerical instability issues. With the squared error loss, the smoothness is very poor: we found that fixed-stepsize gradient descent on the squared error was so sensitive to η that it made the method practically useless for solving the factorization problem. Iterative Soft Thresholding and Fast Iterative Soft Thresholding use a dynamic step size to avoid this issue (see equation (A.15)). In contrast, the negative inner product loss, with respect to each factor, is in some sense *perfectly smooth* (it is linear), so the step size does not factor into convergence proofs.

A.6 Iterative Soft Thresholding (ISTA) and Fast Iterative Soft Thresholding (FISTA)

Iterative Soft Thresholding is a type of *proximal gradient descent*. The proximal operator for any convex function $h(\cdot)$ is defined as

$$\text{prox}_h(\mathbf{x}) := \arg\min_{\mathbf{z}} \;\; \tfrac{1}{2}||\mathbf{z} - \mathbf{x}||_2^2 + h(\mathbf{z})$$

When $h(\mathbf{z})$ is $\lambda||\mathbf{z}||_1$, the proximal operator is the so-called "soft-thresholding" function, which we denote by $\mathcal{S}$:

$$\big(\mathcal{S}[\,\mathbf{x}\,;\gamma\,]\big)_i := \text{sgn}(x_i)\max(|x_i| - \gamma,\, 0\,)$$

Consider taking the squared error loss and adding to it $\lambda||\mathbf{a}_f||_1$:

$$\mathcal{L}(\mathbf{c}, \hat{\mathbf{c}}) + \lambda||\mathbf{a}_f||_1 = \tfrac{1}{2}||\mathbf{c} - \hat{\mathbf{c}}||_2^2 + \lambda||\mathbf{a}_f||_1$$

Applying soft thesholding clearly minimizes this augmented loss function. The strategy is to take gradient steps with respect to the squared error loss but then to pass those updates through the soft thresholding function $\mathcal{S}$. This flavor of proximal gradient descent, where $\hat{\mathbf{c}}$ is a linear function of $\mathbf{a}_f$ and $h(\cdot)$ is the ℓ_1 norm, is called the Iterative Soft Thresholding Algorithm (Daubechies et al., 2004), and is a somewhat old and popular approach for finding sparse solutions to large-scale linear inverse problems.

The dynamics of ISTA are given in equation (A.3) and there are a few parameters worth discussing. First, the dynamic stepsize η can be set via backtracking line search or, as we did, by computing the Lipschitz constant of the loss function gradient:

$$\eta = \frac{1}{L} \quad \Big| \quad ||\nabla_{\mathbf{a}}\mathcal{L}(\mathbf{x}) - \nabla_{\mathbf{a}}\mathcal{L}(\mathbf{y})||_2 \leq L||\mathbf{x} - \mathbf{y}||_2 \;\; \forall\, \mathbf{x},\, \mathbf{y} \tag{A.15}$$

The scalar λ is a hyperparameter that effectively sets the sparsity of the solutions considered– its value should be tuned in order to get good performance in practice. In the experiments we show in this chapter, λ was 0.01. The initial state $\mathbf{a}_f[0]$ is set to $\mathbf{1}$.

Convergence analysis of ISTA is beyond the scope of this dissertation, but it has been shown in various places (Bredies and Lorenz (2008), for instance) that ISTA will converge at a rate $\simeq \mathcal{O}(1/t)$. Iterative Soft Thresholding works well in practice, although for 4 or more factors we find that it is not quite as effective as the algorithms that do constrained descent on the negative inner product loss. By virtue of not directly constraining the coefficients, ISTA allows them to grow outside of $[0, 1]^N$. This may make it easier to find the minimizers $\mathbf{a}_1^\star, \mathbf{a}_2^\star, \ldots, \mathbf{a}_F^\star$, but it may also lead the method to encounter more suboptimal local minimizers, which we found to be the case in practice.

One common criticism of ISTA is that it can get trapped in shallow parts of the loss surface and thus suffers from slow convergence (Bredies & Lorenz, 2008). A straightforward improvement, based on Nesterov's momentum for accelerating first-order methods, was proposed by Beck and Teboulle (2009), which they call Fast Iterative Soft Thresholding (FISTA). The dynamics of FISTA are written in equation (A.4), and converge at the significantly better rate of $\simeq \mathcal{O}(1/t^2)$, a result proven in Beck and Teboulle (2009). Despite this difference in *worst-case* convergence rate, we find that the average-case convergence rate on our particular factorization problem does not significantly differ. Initial coefficients $\mathbf{a}_f[0]$ are set to $\mathbf{1}$ and auxiliary variable α_t is initialized to 1. For all experiments λ was set the same as for ISTA, to 0.01.

A.7 Projected Gradient Descent

Starting from the general optimization form of the factorization problem (2.9), what kind of constraint might it be reasonable to enforce on $\mathbf{a}_f$? The most obvious is that $\mathbf{a}_f$ lie on the simplex $\Delta_{D_f} := \{\mathbf{x} \in \mathbb{R}^{D_f} \mid \sum_i x_i = 1, x_i \geq 0 \; \forall i\}$. Enforcing this constraint means that $\hat{\mathbf{x}}^{(f)}$ stays within the $-1, 1$ hypercube at all times and, as we noted, the optimal values $\mathbf{a}_1^\star, \mathbf{a}_2^\star, \ldots, \mathbf{a}_F^\star$ happen to lie at vertices of the simplex, the standard basis vectors $\mathbf{e}_i$. Another constraint set worth considering is the ℓ_1 ball $\mathcal{B}_{||\cdot||_1}[1] := \{\mathbf{x} \in \mathbb{R}^{D_f} \mid ||\mathbf{x}||_1 \leq 1\}$. This set contains the simplex, but it encompasses much more of $\mathbb{R}^{D_f}$. One reason to consider the ℓ_1 ball is that it dramatically increases the number of feasible global optimizers of (2.9), from which we can easily recover the specific solution to (2.1). This is due to the fact that:

$$c = \mathbf{X}_1\mathbf{a}_1^\star \odot \mathbf{X}_2\mathbf{a}_2^\star \odot \ldots \odot \mathbf{X}_F\mathbf{a}_F^\star \iff c = \mathbf{X}_1(-\mathbf{a}_1^\star) \odot \mathbf{X}_2(-\mathbf{a}_2^\star) \odot \ldots \odot \mathbf{X}_F\mathbf{a}_F^\star$$

and moreover any number of distinct pairs of factor coefficients can be made negative–the sign change cancels out. The result is that while the simplex constraint only allows solution $\mathbf{a}_1^\star, \mathbf{a}_2^\star, \ldots, \mathbf{a}_F^\star$, the ℓ_1 ball constraint also allows solutions $-\mathbf{a}_1^\star, -\mathbf{a}_2^\star, \mathbf{a}_3^\star, \ldots, \mathbf{a}_F^\star$, and $\mathbf{a}_1^\star, \mathbf{a}_2^\star, -\mathbf{a}_3^\star, \ldots, -\mathbf{a}_F^\star$, and $-\mathbf{a}_1^\star, -\mathbf{a}_2^\star, -\mathbf{a}_3^\star, \ldots, -\mathbf{a}_F^\star$, etc. These spurious global minimizers can easily be detected by checking the sign of the largest-magnitude component of $\mathbf{a}_f$. If it is negative we can then multiply by -1 to get $\mathbf{a}_f^\star$. Choosing the ℓ_1 ball over the simplex is purely motivated from the perspective that increasing the size of the constraint set may make finding the global optimizers easier. However, we found that in practice, it did not significantly matter whether Δ_{D_f} or $\mathcal{B}_{||\cdot||_1}[1]$ was used to constrain $\mathbf{a}_f$.

There exist algorithms for efficiently computing projections onto both the simplex and the ℓ_1 ball (see Held et al. (1974), Duchi et al. (2008), and Condat (2016)). We use a variant summarized in Duchi et al. (2008) that has computational complexity $\mathcal{O}(D_f \log D_f)$–recall that $\mathbf{a}_f$ has D_f components, so this is the dimensionality of the simplex or the ℓ_1 ball being projected onto. When constraining to the simplex, we set the initial coefficients $\mathbf{a}_f[0]$ to $\frac{1}{D_f}\mathbf{1}$, the center of the simplex. When constraining to the unit ℓ_1 ball we set $\mathbf{a}_f[0]$ to $\frac{1}{2D_f}\mathbf{1}$, so that all coefficients are equal but the vector is on the interior of the ball. The only hyperparameter is η, which in all experiments was set to 0.01. We remind the reader that we defined the nullspace of the projection operation with equation (2.22) in Section 2.5.6, and the special case for the simplex constraint in (2.23) and (2.24).

Taking projected gradient steps on the negative inner product loss works well and is guaranteed to converge, whether we use the simplex or the ℓ_1 ball constraint. Convergence is guaranteed due to this intuitive fact: any part of $-\eta \nabla_{\mathbf{a}_f}\mathcal{L}$ not in $\mathcal{N}\big(\mathcal{P}_{C_f}[\mathbf{x}]\big)$, induces a change in $\mathbf{a}_f$, denoted by $\Delta\mathbf{a}_f[t]$ which must make an acute angle with $-\nabla_{\mathbf{a}_f}\mathcal{L}$. This is by *the definition of orthogonal projection*, and it is a sufficient condition for showing that $\Delta\mathbf{a}_f[t]$ decreases the value of the loss function. Projected Gradient Descent iterates always reduce the value of the negative inner product loss or leave it unchanged; the function is bounded below on the simplex and the ℓ_1 ball, so this algorithm is guaranteed to converge.

Applying projected gradient descent on the squared error did not work, which is related to the smoothness issue we discussed in Appendix A.5.1, although the behavior was not as

dramatic as with unconstrained gradient descent. We observed in practice that projected gradient descent on the squared error loss easily falls into limit cycles of the dynamics. It was for this reason that we restricted our attention with projected gradient descent to the negative inner product loss.

A.8 Multiplicative Weights

When we have simplex constraints $C_f = \Delta_{D_f}$, the Multiplicative Weights algorithm is an elegant way to perform the superposition search. It naturally enforces the simplex constraint by maintaining a set of auxiliary variables, the 'weights', which define the choice of $\mathbf{a}_f$ at each iteration. See equation (A.6) for the dynamics of Multiplicative Weights. We choose a fixed stepsize $\eta \leq 0.5$ and initial values for the weights all one: $\mathbf{w}_f[0] = \mathbf{1}$. In experiments in this paper we set $\eta = 0.3$. The variable ρ exists to normalize the term $\frac{1}{\rho}\nabla_{\mathbf{a}_f}\mathcal{L}$ so that each element lies in the interval $[-1, 1]$.

Multiplicative Weights is an algorithm primarily associated with game theory and online optimization, although it has been independently discovered in a wide variety of fields (Arora et al., 2012). Please see Arora's excellent review of Multiplicative Weights for a discussion of the fascinating historical and analytical details of this algorithm. Multiplicative Weights is often presented as a decision policy for discrete-time games. However, through a straightforward generalization of the discrete actions into directions in a continuous vector space, one can apply Multiplicative Weights to problems of *online convex optimization*, which is discussed at length in Arora et al. (2012) and Hazan et al. (2016). We can think of solving our problem (2.9) as if it were an online convex optimization problem where we update each factor $\hat{\mathbf{x}}^{(f)}$ according to its own Multiplicative Weights update, one at a time. The function $\mathcal{L}$ is convex with respect to $\mathbf{a}_f$, but is changing at each iteration due the updates for the *other* factors - it is in this sense that we are treating (2.9) as an online convex optimization problem.

A.8.1 Multiplicative Weights is a descent method

A descent method on $\mathcal{L}$ is any algorithm that iterates $\mathbf{a}_f[t+1] = \mathbf{a}_f[t] + \eta[t]\Delta\mathbf{a}_f[t]$ where the update $\Delta\mathbf{a}_f[t]$ makes an acute angle with $-\nabla_{\mathbf{a}_f}\mathcal{L}$: $\nabla_{\mathbf{a}_f}\mathcal{L}^\top \Delta\mathbf{a}_f[t] < 0$. In the case of Multiplicative Weights, we can equivalently define a descent method based on $\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top \Delta\mathbf{w}_f[t] < 0$ where $\tilde{\mathcal{L}}(\mathbf{w}_f)$ is the loss as a function of the *weights* and $\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}$ is its gradient with respect to those weights. The loss as a function of the weights comes via the substitution

$\mathbf{a}_f = \frac{\mathbf{w}_f}{\sum_i w_{fi}} := \frac{\mathbf{w}_f}{\Phi_f}$. We now prove that $\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top \Delta\mathbf{w}_f[t] < 0$:

$$\begin{aligned}
\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} &= \frac{\partial \mathbf{a}_f}{\partial \mathbf{w}_f}\frac{\partial \mathcal{L}}{\partial \mathbf{a}_f} \\
&= \begin{bmatrix} \frac{\Phi_f - w_{f1}}{\Phi_f^2} & \frac{-w_{f2}}{\Phi_f^2} & \dots & \frac{-w_{fk}}{\Phi_f^2} \\ \frac{-w_{f1}}{\Phi_f^2} & \frac{\Phi_f - w_{f2}}{\Phi_f^2} & \dots & \frac{-w_{fk}}{\Phi_f^2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{-w_{f1}}{\Phi_f^2} & \frac{-w_{f2}}{\Phi_f^2} & \dots & \frac{\Phi_f - w_{fk}}{\Phi_f^2} \end{bmatrix} \nabla_{\mathbf{a}_f}\mathcal{L} \\
&= \Big(\frac{1}{\Phi_f}\mathbf{I} - \frac{1}{\Phi_f^2}\mathbf{1}\mathbf{w}^\top\Big)\nabla_{\mathbf{a}_f}\mathcal{L} \\
&= \frac{1}{\Phi_f}\nabla_{\mathbf{a}_f}\mathcal{L} - \frac{\mathcal{L}(\mathbf{a}_f)}{\Phi_f}\mathbf{1}
\end{aligned} \tag{A.16}$$

This allows us to write down $\Delta\mathbf{w}_f[t]$ in terms of $\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}$:

$$\begin{aligned}
\Delta\mathbf{w}_f[t] &= -\frac{1}{\rho}\mathbf{w}_f[t] \odot \nabla_{\mathbf{a}_f}\mathcal{L} = -\frac{1}{\rho}\mathbf{w}_f[t] \odot \Big(\Phi_f \nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} + \mathcal{L}(\mathbf{a}_f[t])\mathbf{1}\Big) \\
&= -\frac{\Phi_f}{\rho}\mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\rho}\mathbf{w}_f[t]
\end{aligned} \tag{A.17}$$

And then we can easily show the desired result:

$$\begin{aligned}
\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top \Delta\mathbf{w}_f[t] &= -\frac{\Phi_f}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top \mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top \mathbf{w}_f[t] \\
&= -\frac{\Phi_f}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top \mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\rho}\Big(\frac{1}{\Phi_f}\nabla_{\mathbf{a}_f}\mathcal{L}^\top - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\Phi_f}\mathbf{1}^\top\Big)\mathbf{w}_f[t] \\
&= -\frac{\Phi_f}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top \mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\rho}\Big(\mathcal{L}(\mathbf{a}_f[t]) - \mathcal{L}(\mathbf{a}_f[t])\Big) \\
&= -\frac{\Phi_f}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top \mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} \\
&< 0
\end{aligned} \tag{A.18}$$

The last line follows directly from the fact that $\mathbf{w}_f$ are always positive by construction in Multiplicative Weights. Therefore, the matrix $\mathrm{diag}(\mathbf{w}_f[t])$ is positive definite and the term $\frac{\Phi_f}{\rho}$ is strictly greater than 0. We've shown that the iterates of Multiplicative Weights always make steps in descent directions. When the loss $\mathcal{L}$ is the negative inner product, it is guaranteed to decrease at each iteration. Empirically, multiplicative weights applied to the squared error loss *also always decreases the loss function.* We said in Appendix A.5.1 that descent on the squared error with a fixed step size is not in general guaranteed to converge. However, the behavior we observe with Multiplicative Weights descent on the squared error

might be explained by the fact that the stepsize is normalized by ρ at each iteration in this algorithm. Both functions are bounded below over the constraint set Δ_{D_f}, so therefore Multiplicative Weights must converge to a fixed point. In practice, we pick a step size η between 0.1 and 0.5 and run the algorithm until the normalized magnitude of the change in the coefficients is below some small threshold:

$$\frac{\left|\mathbf{a}_f[t+1] - \mathbf{a}_f[t]\right|}{\eta} < \epsilon$$

The simulations we showed in the Results section utilized $\eta = 0.3$ and $\epsilon = 10^{-5}$.

A.9 Map Seeking Circuits

Map Seeking Circuits (MSCs) are neural networks designed to solve invariant pattern recognition problems. Their theory and applications have been gradually developed by Arathorn and colleagues over the past 18 years (see, for example, D. W. Arathorn (2001, 2002), Gedeon and Arathorn (2007), and Harker et al. (2007)), but remain largely unknown outside of a small community of vision researchers. In their original conception, they solve a "correspondence maximization" or "transformation discovery" problem in which the network is given a visually transformed instance of some template object and has to recover the identity of the object as well as a set of transformations that explain its current appearance. The approach taken in Map Seeking Circuits is to superimpose the possible transformations in the same spirit as we have outlined for solving the factorization problem. We cannot give the topic a full treatment here but simply note that the original formulation of Map Seeking Circuits can be directly translated to our factorization problem wherein each type of transformation (e.g. translation, rotation, scale) is one of the F factors, and the particular values of the transformation are vectors in the codebooks $\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_F$. The loss function is $\mathcal{L} : \mathbf{x}, \mathbf{y} \mapsto -\langle \mathbf{x}, \mathbf{y} \rangle$ and the constraint set is $[0, 1]^{D_f}$ (both by convention in Map Seeking Circuits). The dynamics of Map Seeking Circuits are given in equation (A.7), with initial values $\mathbf{a}_f[0] = \mathbf{1}$ for each factor. The small threshold ϵ is a hyperparameter, which we set to 10^{-5} in experiments, along with the stepsize $\eta = 0.1$. Gedeon and Arathorn (2007) and Harker et al. (2007) proved (with some minor technicalities we will not detail here) that Map Seeking Circuits always converge to either a scalar multiple of a canonical basis vector, or the zero vector. That is, $\mathbf{a}_f[\infty] = \beta_f \mathbf{e}_i$ or $\mathbf{0}$ (where $(\mathbf{e}_i)_j = 1$ if $j = i$ and 0 otherwise, and β_f is a positive scalar).

Due to the normalizing term ρ, the updates to $\mathbf{a}_f$ *can never be positive*. Among the components of $\nabla_{\mathbf{a}_f}\mathcal{L}$ which are negative, the one with the largest magnitude corresponds to a component of $\mathbf{a}_f$ which sees an update of 0. All other components are decreased by an amount which is proportional to the gradient. We noted in comments on (A.11) that the smallest element of $\nabla_{\mathbf{a}_f}\mathcal{L}$ corresponds to the codevector which best matches $\mathbf{c} \odot \hat{\mathbf{o}}^{(f)}$, a "suggestion" for $\hat{\mathbf{x}}^{(f)}$ based on the current states of the other factors. The dynamics of Map

Seeking Circuits thus preserve the weight of the codevector which matches best and decrease the weight of the other codevectors, by an amount which is proportional to their own match. Once the weight on a codevector drops below the threshold, it is set to zero and no longer participates in the search. The phenomenon wherein the correct coefficient $a_{fi^\star}$ drops out of the search is called "sustained collusion" by Arathorn (D. W. Arathorn, 2002) and is a failure mode of Map Seeking Circuits.

A.10 Percolated noise in Outer Product Resonator Networks

A Resonator Network with outer product weights $\mathbf{X}_f\mathbf{X}_f^\top$ that is initialized to the correct factorization is not guaranteed to remain there, just as a Hopfield Network with outer product weights initialized to one of the 'memories' is not guaranteed to remain there. This is in contrast to a Resonator Network (and a Hopfield Network) with Ordinary Least Squares weights $\mathbf{X}_f\big(\mathbf{X}_f^\top\mathbf{X}_f\big)^{-1}\mathbf{X}_f^\top$, for which each of the codevectors are always fixed points. In this section, when we refer simply to a Resonator Network or a Hopfield Network we are referring to the variants of these models that use outer product weights.

The bitflip probability for the fth factor of a Resonator Network is denoted r_f and defined in (2.13). Section A.10.1 derives r_1, which is equal to the bitflip probability for a Hopfield network, first introduced by (2.12) in the main text. Section A.10.2 derives r_2, and then section A.10.3 collects all of the ingredients to express the general r_f.

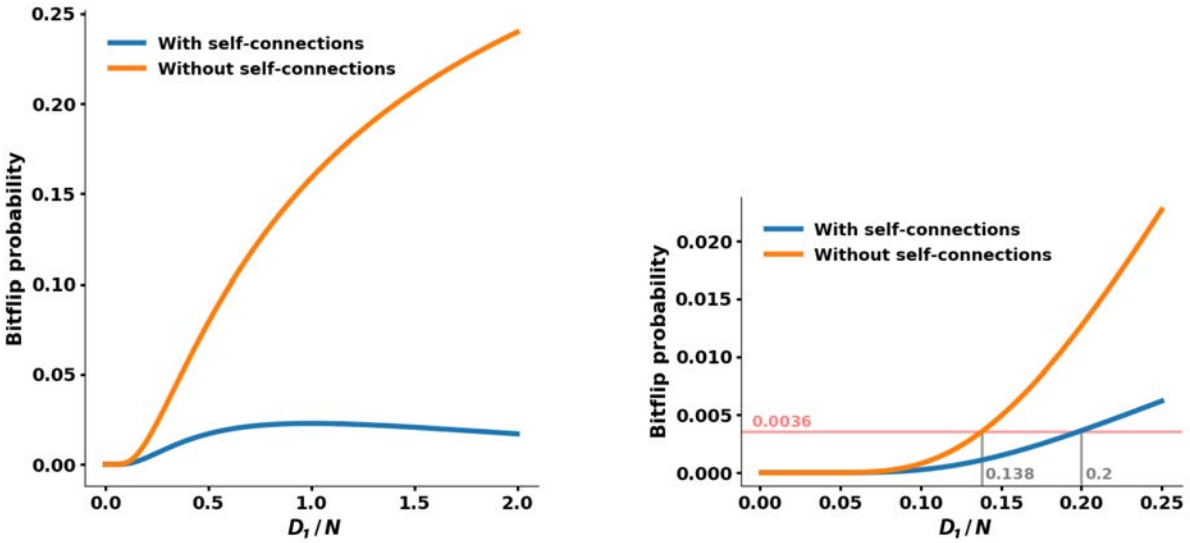
A.10.1 First factor

The stability of the first factor in a Resonator Network is the same as the stability of the state of a Hopfield network–at issue is the distribution of $\hat{\mathbf{x}}^{(1)}[1]$:

$$\hat{\mathbf{x}}^{(1)}[1] = \operatorname{sgn}\big(\mathbf{X}_1\mathbf{X}_1^\top\mathbf{x}_\star^{(1)}\big) := \operatorname{sgn}\big(\mathbf{\Gamma}\big)$$

Assuming each codevector (each column of $\mathbf{X}_1$, including the vector $\mathbf{x}_\star^{(1)}$) is a random bipolar vector, each component of $\mathbf{\Gamma}$ is a random variable. Its distribution can be deduced from writing it out in terms of constant and random components:

$$\begin{aligned}
\Gamma_i &= \sum_m^{D_1}\sum_j^N \big(\mathbf{x}_m^{(1)}\big)_i \big(\mathbf{x}_m^{(1)}\big)_j \big(\mathbf{x}_\star^{(1)}\big)_j \\
&= N\big(\mathbf{x}_\star^{(1)}\big)_i + \sum_{m\neq\star}^{D_1}\sum_j^N \big(\mathbf{x}_m^{(1)}\big)_i \big(\mathbf{x}_m^{(1)}\big)_j \big(\mathbf{x}_\star^{(1)}\big)_j \\
&= N\big(\mathbf{x}_\star^{(1)}\big)_i + (D_1-1)\big(\mathbf{x}_\star^{(1)}\big)_i + \sum_{m\neq\star}^{D_1}\sum_{j\neq i}^N \big(\mathbf{x}_m^{(1)}\big)_i \big(\mathbf{x}_m^{(1)}\big)_j \big(\mathbf{x}_\star^{(1)}\big)_j
\end{aligned} \tag{A.19}$$



(a) Bitflip prob. for $D_1 / N \in (0, 2]$ (b) Bitflip prob. for $D_1 / N \in (0, 0.25]$

Figure A.4: Effect of self-connections on bitflip probability

The third term is a sum of $(N-1)(D_1-1)$ i.i.d. Rademacher random variables, which in the limit of large ND_1 can be well-approximated by a Gaussian random variable with mean zero and variance $(N-1)(D_1-1)$. Therefore, Γ_i is approximately Gaussian with mean $(N+D_1-1)\big(\mathbf{x}_\star^{(1)}\big)_i$ and variance $(N-1)(D_1-1)$. The probability that $\big(\hat{\mathbf{x}}^{(1)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(1)}\big)_i$ is given by the cumulative density function of the Normal distribution:

$$\begin{aligned} h_1 &:= Pr\big[\,\big(\hat{\mathbf{x}}^{(1)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(1)}\big)_i\,\big] \\ &= \Phi\Big(\frac{-N - D_1 + 1}{\sqrt{(N-1)(D_1-1)}}\Big) \end{aligned} \tag{A.20}$$

We care about the ratio D_1 / N and how the bitflip probability h_1 scales with this number. We've called this h_1 to denote the Hopfield bitflip probability but it is also r_1, the bitflip probability for the *first* factor of a Resonator Network. We'll see that for the second, third, fourth, and other factors, h_f will not equal r_f, which is what we mean by percolated noise, the focus of Section 2.5.1 in the main text. If we eliminate all "self-connection" terms from $\mathbf{X}_1\mathbf{X}_1^\top$, by setting each element on the diagonal to zero, then the second term in (A.19) is eliminated and the bitflip probability is $\Phi\big(\frac{-N}{\sqrt{(N-1)(D_1-1)}}\big)$. This is actually significantly different from (A.20), which we can see in Figure A.4. With self-connections, the bitflip probability is maximized when $D_1 = N$ (the reader can verify this via simple algebra), and its maximum value is ≈ 0.023. Without self-connections, the bitflip probability asymptotes at 0.5. The actual useful operating regime of both these networks is where D_1 is significantly less than N, which we zoom in on in Figure A.4b. A "mean-field" analysis of Hopfield Networks developed by Amit, Gutfreund, and Sompolinsky (Amit et al., 1985, 1987) showed

that when $D_1 \,/\, N > 0.138$, a phase-change phenomenon occurs in which a small number of initial bitflips (when the probability is 0.0036 according to the above approximation) build up over subsequent iterations and the network almost always moves far away from $\mathbf{x}_\star^{(1)}$, making it essentially useless. We can see that the same bitflip probability is suffered at a significantly higher value for $D_1 \,/\, N$ when we have self-connections–the vector $\mathbf{x}_\star^{(1)}$ is significantly more stable in this sense. We also found that a Resonator Network has higher operational capacity (see Section 2.5.2) when we leave in the self-connections. As a third point of interest, computing $\mathbf{X}_f \mathbf{X}_f^\top \mathbf{x}_\star^{(1)}$ is often much faster when we keep each codebook matrix separate (instead of forming the synaptic matrix $\mathbf{X}_f \mathbf{X}_f^\top$ directly), in which case removing the self-connection terms involves extra computation in each iteration of the algorithm. For all of these reasons, we choose to keep self-connection terms in the Resonator Network.

A.10.2 Second factor

When we update the second factor, we have

$$\hat{\mathbf{x}}^{(2)}[1] = \operatorname{sgn}\Big(\mathbf{X}_2 \mathbf{X}_2^\top \big(\hat{\mathbf{o}}^{(2)}[1] \odot \mathbf{c}\big)\Big) := \operatorname{sgn}\big(\mathbf{\Gamma}\big)$$

Here we're just repurposing the notation $\mathbf{\Gamma}$ to indicate the vector which gets thresholded to -1 and $+1$ by the sign function to generate the new state $\hat{\mathbf{x}}^{(2)}[1]$. Some of the components of the vector $\hat{\mathbf{o}}^{(2)}[1] \odot \mathbf{c}$ will be the same as $\mathbf{x}_\star^{(2)}$ and some (hopefully small) number of the components will have been flipped compared to $\mathbf{x}_\star^{(2)}$ by the update to factor 1. Let us denote the set of components which flipped as $\mathbb{Q}$. The set of components that did not flip is $\mathbb{Q}^c$. The number of bits that did or did not flip is the *size* of these sets, denoted by $|\mathbb{Q}|$ and $|\mathbb{Q}^c|$, respectively. We have to keep track of these two sets separately because it will affect the probability that a component of $\hat{\mathbf{x}}^{(2)}[1]$ is flipped relative to $\mathbf{x}_\star^{(2)}$. We can write out the constant and random parts of Γ_i along the same lines as what we did in (A.19).

$$\begin{aligned}
\Gamma_i &= \sum_m^{D_2} \sum_j^N \big(\mathbf{x}_m^{(2)}\big)_i \big(\mathbf{x}_m^{(2)}\big)_j \big(\hat{\mathbf{o}}^{(2)}[1] \odot \mathbf{c}\big)_j \\
&= \sum_m^{D_2} \sum_{j \in \mathbb{Q}^c}^N \big(\mathbf{x}_m^{(2)}\big)_i \big(\mathbf{x}_m^{(2)}\big)_j \big(\mathbf{x}_\star^{(2)}\big)_j - \sum_m^{D_2} \sum_{j \in \mathbb{Q}}^N \big(\mathbf{x}_m^{(2)}\big)_i \big(\mathbf{x}_m^{(2)}\big)_j \big(\mathbf{x}_\star^{(2)}\big)_j \\
&= |\mathbb{Q}^c| \big(\mathbf{x}_\star^{(2)}\big)_i + \sum_{m \neq \star}^{D_2} \sum_{j \in \mathbb{Q}^c}^N \big(\mathbf{x}_m^{(2)}\big)_i \big(\mathbf{x}_m^{(2)}\big)_j \big(\mathbf{x}_\star^{(2)}\big)_j - |\mathbb{Q}| \big(\mathbf{x}_\star^{(2)}\big)_i - \sum_{m \neq \star}^{D_2} \sum_{j \in \mathbb{Q}}^N \big(\mathbf{x}_m^{(2)}\big)_i \big(\mathbf{x}_m^{(2)}\big)_j \big(\mathbf{x}_\star^{(2)}\big)_j \\
&= (N - 2|\mathbb{Q}|) \big(\mathbf{x}_\star^{(2)}\big)_i + \sum_{m \neq \star}^{D_2} \sum_{j \in \mathbb{Q}^c}^N \big(\mathbf{x}_m^{(2)}\big)_i \big(\mathbf{x}_m^{(2)}\big)_j \big(\mathbf{x}_\star^{(2)}\big)_j - \sum_{m \neq \star}^{D_2} \sum_{j \in \mathbb{Q}}^N \big(\mathbf{x}_m^{(2)}\big)_i \big(\mathbf{x}_m^{(2)}\big)_j \big(\mathbf{x}_\star^{(2)}\big)_j
\end{aligned} \tag{A.21}$$

If i is in the set of bits which did not flip previously, then there is a constant $(D_2-1)\big(\mathbf{x}_\star^{(2)}\big)_i$ which comes out of the second term above. If i is in the set of bits which did flip previously, then there is a constant $-(D_2-1)\big(\mathbf{x}_\star^{(2)}\big)_i$ which comes out of the third term above. The remaining contribution to Γ_i is, in either case, a sum of $(N-1)(D_2-1)$ i.i.d. Rademacher random variables, analogously to what we had in (A.19). Technically $|\mathbb{Q}|$ is a random variable but when N is of any moderate size it will be close to $r_1 N$, the bitflip probability for the first factor. Therefore, Γ_i is approximately Gaussian with mean either $\big(N(1-2r_1)+(D_2-1)\big)\big(\mathbf{x}_\star^{(2)}\big)_i$ or $\big(N(1-2r_1)-\big(D_2-1\big)\big)\big(\mathbf{x}_\star^{(2)}\big)_i$, depending on whether $i\in\mathbb{Q}^c$ or $i\in\mathbb{Q}$. We call the *conditional* bitflip probabilities that result from these two cases $r_{2'}$ and $r_{2''}$:

$$\begin{aligned} r_{2'} &:= Pr\big[\,\big(\hat{\mathbf{x}}^{(2)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(2)}\big)_i \;\big|\; \big(\hat{\mathbf{o}}^{(2)}[1]\odot\mathbf{c}\big)_i = \big(\mathbf{x}_\star^{(2)}\big)_i\,\big] \\ &= \Phi\Big(\frac{-N(1-2r_1)-(D_2-1)}{\sqrt{(N-1)(D_2-1)}}\Big) \end{aligned} \tag{A.22}$$

$$\begin{aligned} r_{2''} &:= Pr\big[\,\big(\hat{\mathbf{x}}^{(2)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(2)}\big)_i \;\big|\; \big(\hat{\mathbf{o}}^{(2)}[1]\odot\mathbf{c}\big)_i \neq \big(\mathbf{x}_\star^{(2)}\big)_i\,\big] \\ &= \Phi\Big(\frac{-N(1-2r_1)+(D_2-1)}{\sqrt{(N-1)(D_2-1)}}\Big) \end{aligned} \tag{A.23}$$

The total bitflip probability for updating the second factor, r_2, is then $r_{2'}(1-h_1)+r_{2''}h_1$.

A.10.3 All other factors

It hopefully should be clear that the general development above for the bitflip probability of the second factor will apply to all subsequent factors–we just need to make one modification to notation. We saw that bitflip probability was different depending on whether the component had flipped in the previous factor (the difference between (A.22) and (A.23)). In the general case, what really matters is whether the factor sees a *net* bitflip from the other factors. It might be the case that the component had initially flipped but was flipped back by subsequent factors–all that matters is whether an *odd* number of previous factors flipped the component. To capture this indirectly, we define the quantity n_f to be the net bitflip probability that is passed on to the next factor (this is equation 2.14 in the main text):

$$n_f := Pr\big[\,\big(\hat{\mathbf{o}}^{(f+1)}[t]\odot\mathbf{c}\big)_i \neq \big(\mathbf{x}_\star^{(f+1)}\big)_i\,\big]$$

For the first factor, $r_1 = n_1$ but in the general case it should be clear that

$$r_f = r_{f'}(1-n_{f-1}) + r_{f''}n_{f-1}$$

which is equation (2.17) in the main text. This expression is just marginalizing over the probability that a net biflip was not seen (first term) and the probability that a net bitflip was seen (second term). The expression for the general n_f is slightly different:

$$n_f = r_{f'}(1-n_{f-1}) + (1-r_{f''})n_{f-1}$$

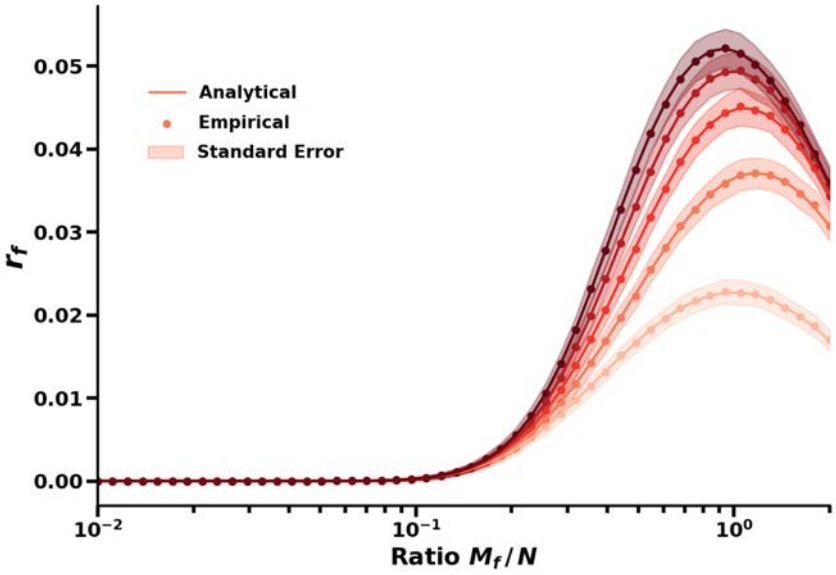


Figure A.5: Agreement between simulation and theory for r_f. Shades indicate factors 1-5 (light to dark).

which is equation (2.18) in the main text. The base of the recursion is $n_0 = 0$, which makes intuitive sense because factor 1 sees no percolated noise.

In (A.22) and (A.23) above we had r_1 but what really belongs there in the general case is n_{f-1}. This brings us to our general statement for the conditional bitflip probabilities $r_{f'}$ and $r_{f''}$, which are equations 2.19 and 2.20 in the main text:

$$r_{f'} = \Phi\Big(\frac{-N(1-2n_{f-1}) - (D_f - 1)}{\sqrt{(N-1)(D_f-1)}}\Big)$$

$$r_{f''} = \Phi\Big(\frac{-N(1-2n_{f-1}) + (D_f - 1)}{\sqrt{(N-1)(D_f-1)}}\Big)$$

What we have derived here in Appendix A.10 are the equations (2.12) - (2.20). This result agrees very well with data generated in experiments where one actually counts the bitflips in a randomly instantiated Resonator Network. In Figure A.5 we show the sampling distribution of r_f from these experiments compared to the analytical expresssion for r_f. Dots indicate the mean value for r_f and the shaded region indicates one standard deviation about the mean, the standard error of this sampling distribution. We generated this plot with 250 iid random trials for each point. Solid lines are simply the analytical values for r_f, which one can see are in very close agreement with the sampling distribution.

Bibliography

Adelson, E. H., & Pentland, A. P. (1996). The perception of shading and reflectance. *Perception as Bayesian inference*, 409–423.

Amari, S.-I., & Maginu, K. (1988). Statistical neurodynamics of associative memory. *Neural Networks*, *1*(1), 63–73.

Amit, D. J., Gutfreund, H., & Sompolinsky, H. (1987). Information storage in neural networks with low levels of activity. *Physical Review A*, *35*(5), 2293.

Amit, D. J., Gutfreund, H., & Sompolinsky, H. (1985). Storing infinite numbers of patterns in a spin-glass model of neural networks. *Physical Review Letters*, *55*(14), 1530.

Anandkumar, A., Ge, R., Hsu, D., Kakade, S. M., & Telgarsky, M. (2014). Tensor decompositions for learning latent variable models. *Journal of Machine Learning Research*, *15*, 2773–2832.

Arathorn, D. W. (2002). *Map-seeking circuits in visual cognition: A computational mechanism for biological and machine vision*. Stanford, CA, Stanford University Press.

Arathorn, D. W. (2001). Recognition under transformation using superposition ordering property. *Electronics Letters*, *37*(3), 164–166.

Arora, S., Hazan, E., & Kale, S. (2012). The multiplicative weights update method: A meta-algorithm and applications. *Theory of Computing*, *8*(1), 121–164.

Barron, J. T., & Malik, J. (2015). Shape, illumination, and reflectance from shading. *IEEE transactions on pattern analysis and machine intelligence*, *37*(8), 1670–1687.

Barrow, H., & Tenenbaum, J. (1978). Recovering intrinsic scene characteristics from images. *Computer Vision Systems*, *2*, 3–26.

Beck, A., & Teboulle, M. (2009). A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, *2*(1), 183–202.

Blanz, V., & Vetter, T. (1999). A morphable model for the synthesis of 3d faces, In *Proceedings of the 26th annual conference on computer graphics and interactive techniques*.

Boucheron, S., Lugosi, G., & Massart, P. (2013). *Concentration inequalities* [A nonasymptotic theory of independence, with a foreword by Michel Ledoux]. Oxford University Press, Oxford. https://doi.org/10.1093/acprof:oso/9780199535255.001.0001

Bredies, K., & Lorenz, D. A. (2008). Linear convergence of iterative soft-thresholding. *Journal of Fourier Analysis and Applications*, *14*(5-6), 813–837.

Cadieu, C. F., & Olshausen, B. A. (2012). Learning intermediate-level representations of form and motion from natural movies. *Neural computation*, *24*(4), 827–866.

Cannon, L. E. (1969). *A cellular computer to implement the kalman filter algorithm* (Doctoral dissertation). Montana State University-Bozeman, College of Engineering.

Carroll, J. D., & Chang, J.-J. (1970). Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, *35*(3), 283–319.

Carroll, J. D., Pruzansky, S., & Kruskal, J. B. (1980). Candelinc: A general approach to multidimensional analysis of many-way arrays with linear constraints on parameters. *Psychometrika*, *45*(1), 3–24.

Chen, Y., Paiton, D. M., & Olshausen, B. A. (2018). The sparse manifold transform, In *Advances in neural information processing systems*.

Chengxiang, Z., Dasgupta, C., & Singh, M. P. (2000). Retrieval properties of a hopfield model with random asymmetric interactions. *Neural computation*, *12*(4), 865–880.

Cheung, B., Livezey, J., Bansal, A., & Olshausen, B. (2015). Discovering hidden factors of variation in deep networks, In *International conference on learning representations (iclr 2015), workshop*.

Cheung, B., Weiss, E., & Olshausen, B. (2017). Emergence of foveal image sampling from learning to attend in visual scenes, In *International conference on learning representations (iclr)*.

Cohen, M. A., & Grossberg, S. (1983). Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE transactions on systems, man, and cybernetics*, (5), 815–826.

Condat, L. (2016). Fast projection onto the simplex and the l1 ball. *Mathematical Programming*, *158*(1-2), 575–585.

Cox, G. E., Kachergis, G., Recchia, G., & Jones, M. N. (2011). Toward a scalable holographic word-form representation. *Behavior research methods*, *43*(3), 602–615.

Culpepper, B., & Olshausen, B. A. (2009). Learning transport operators for image manifolds, In *Advances in neural information processing systems*.

da Silva, A. P., Comon, P., & de Almeida, A. L. (2015). An iterative deflation algorithm for exact cp tensor decomposition, In *2015 ieee international conference on acoustics, speech and signal processing (icassp)*. IEEE.

Daubechies, I., Defrise, M., & De Mol, C. (2004). An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, *57*(11), 1413–1457.

De Lathauwer, L., De Moor, B., & Vandewalle, J. (2000a). A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications*, *21*(4), 1253–1278.

De Lathauwer, L., De Moor, B., & Vandewalle, J. (2000b). On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors. *SIAM journal on Matrix Analysis and Applications*, *21*(4), 1324–1342.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Duchi, J., Shalev-Shwartz, S., Singer, Y., & Chandra, T. (2008). Efficient projections onto the l 1-ball for learning in high dimensions, In *Proceedings of the 25th international conference on machine learning*. ACM.

Fiete, I. R., Burak, Y., & Brookings, T. (2008). What grid cells convey about rat location. *Journal of Neuroscience*, *28*(27), 6858–6871.

Fodor, J. A. (1975). *The language of thought*. Cambridge, MA, Harvard university press.

Fodor, J. A., & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: A critical analysis. *Cognition*, *28*(1-2), 3–71.

Forbus, K. D., Gentner, D., & Law, K. (1995). Mac/fac: A model of similarity-based retrieval. *Cognitive science*, *19*(2), 141–205.

Frady, E. P., Kleyko, D., & Sommer, F. T. (2018). A theory of sequence indexing and working memory in recurrent neural networks. *Neural Computation*, *30*(6), 1449–1513.

Frady, E. P., Kent, S. J., Kanerva, P., Olshausen, B., & Sommer, F. (2018). Cognitive neural systems for disentangling compositions, In *Cognitive computing 2018 (extended abstract)*.

Frady, E. P., Kent, S. J., & Olshausen, B. A. (2018). A recurrent neural network model for factoring distributed representations, In *Computational and systems neuroscience (cosyne '18) (extended abstract) [link]*.

Frady, E. P., Kent, S. J., Olshausen, B. A., & Sommer, F. T. (2020). Resonator networks, 1: An efficient solution for factoring high-dimensional, distributed representations of data structures. *Neural Computation*, *32*(12), 2311–2331. https://doi.org/10.1162/neco_a_01331

Frady, E. P., & Sommer, F. T. (2019). Robust computation with rhythmic spike patterns. *Proceedings of the National Academy of Sciences*, *116*(36), 18050–18059.

Gayler, R. W. (1998). Multiplicative binding, representation operators & analogy (workshop poster) (K. Holyoak, D. Gentner, & B. Kokinov, Eds.). In K. Holyoak, D. Gentner, & B. Kokinov (Eds.), *Advances in analogy research: Integration of theory and data from the cognitive, computational, and neural sciences*.

Gayler, R. W. (2004). Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience. *arXiv preprint arXiv:cs/0412059*.

Gedeon, T., & Arathorn, D. (2007). Convergence of map seeking circuits. *Journal of Mathematical Imaging and Vision*, *29*(2-3), 235–248.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive science*, *7*(2), 155–170.

Gentner, D., Rattermann, M. J., & Forbus, K. D. (1993). The roles of similarity in transfer: Separating retrievability from inferential soundness. *Cognitive psychology*, *25*(4), 524–575.

Gupta, S., Imani, M., & Rosing, T. (2018). Felix: Fast and energy-efficient logic in memory, In *2018 ieee/acm international conference on computer-aided design (iccad)*. IEEE.

Hafting, T., Fyhn, M., Molden, S., Moser, M.-B., & Moser, E. I. (2005). Microstructure of a spatial map in the entorhinal cortex. *Nature*, *436*(7052), 801–806.

Harker, S., Vogel, C. R., & Gedeon, T. (2007). Analysis of constrained optimization variants of the map-seeking circuit algorithm. *Journal of Mathematical Imaging and Vision*, *29*(1), 49–62.

Harshman, R. A. (1970). *Foundations of the parafac procedure: Models and conditions for an 'explanatory' multimodal factor analysis* (tech. rep. UCLA Working Papers in Phonetics, 16, 1-84). University of California, Los Angeles.

Hazan, E. et al. (2016). Introduction to online convex optimization. *Foundations and Trends in Optimization*, *2*(3-4), 157–325.

Hebb, D. O. (1949). *The organization of behavior; a neuropsychological theory.* New York, NY, John Wiley & Sons.

Held, M., Wolfe, P., & Crowder, H. P. (1974). Validation of subgradient optimization. *Mathematical programming*, *6*(1), 62–88.

Hertz, J., Grinstein, G., & Solla, S. (1986). Memory networks with asymmetric bonds, In *Aip conference proceedings*. American Institute of Physics.

Hillar, C. J., & Lim, L.-H. (2013). Most tensor problems are np-hard. *Journal of the ACM (JACM)*, *60*(6), 1–39.

Hinton, G. E., Rumelhart, D. E., & McClelland, J. L. (1986). Distributed representations. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition: Foundations* (pp. 77–109). Cambridge, MA, MIT Press.

Hinton, G. E. (1981a). Implementing semantic networks in parallel hardware. In G. E. Hinton & J. A. Anderson (Eds.), *Parellel models of associative memory* (pp. 191–217). Hillsdale, N.J., Laurence Erlbaum Associates.

Hinton, G. E. (1990). Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, *46*(1-2), 47–75.

Hinton, G. E. (1981b). A parallel computation that assigns canonical object-based frames of reference, In *Proceedings of the 7th international joint conference on artificial intelligence-volume 2*. Morgan Kaufmann Publishers Inc.

Hitchcock, F. L. (1927). The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, *6*(1-4), 164–189.

Hofstadter, D., & Sander, E. (2013). *Surfaces and essences: Analogy as the fuel and fire of thinking*. New York, NY, Basic Books.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, *79*(8), 2554–2558.

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the national academy of sciences*, *81*(10), 3088–3092.

Hopfield, J. J., & Tank, D. W. (1985). Neural computation of decisions in optimization problems. *Biological cybernetics*, *52*(3), 141–152.

Hoyer, P. O. (2004). Non-negative matrix factorization with sparseness constraints. *Journal of machine learning research*, *5*(Nov), 1457–1469.

Huang, J., & Hagiwara, M. (1999). A new multidimensional associative memory based on distributed representation and its applications, In *Ieee international conference on systems, man, and cybernetics*.

Janner, M., Wu, J., Kulkarni, T. D., Yildirim, I., & Tenenbaum, J. (2017). Self-supervised intrinsic image decomposition. *Advances in Neural Information Processing Systems*, *30*, 5936–5946.

Johnson, J., Hariharan, B., van der Maaten, L., Fei-Fei, L., Lawrence Zitnick, C., & Girshick, R. (2017). Clevr: A diagnostic dataset for compositional language and elementary visual reasoning, In *Proceedings of the ieee conference on computer vision and pattern recognition*.

Julesz, B. (1971). *Foundations of cyclopean perception*. Chicago, IL, University of Chicago Press.

Kainen, P. C., & Kurkova, V. (1993). Quasiorthogonal dimension of euclidean spaces. *Applied mathematics letters*, *6*(3), 7–10.

Kanerva, P. (1996). Binary spatter-coding of ordered k-tuples, In *International conference on artificial neural networks*.

Kanerva, P. (2009). Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, *1*(2), 139–159.

Kanerva, P. (2010). What we mean when we say "what's the dollar of mexico?": Prototypes and mapping in concept space, In *2010 aaai fall symposium series*.

Kent, S. J., Frady, E. P., Sommer, F. T., & Olshausen, B. A. (2020). Resonator networks, 2: Factorization performance and capacity compared to optimization-based methods. *Neural Computation*, *32*(12), 2332–2388. https://doi.org/10.1162/neco_a_01329

Kent, S. J., & Olshausen, B. A. (2017). A vector symbolic approach to scene transformation, In *Cognitive computational neuroscience (ccn '17) (extended abstract) [link]*.

Kobayashi, M., Hattori, M., & Yamazaki, H. (2002). Multidirectional associative memory with a hidden layer. *Systems and Computers in Japan*, *33*(3), 1494–1502.

Kolda, T. G., & Bader, B. W. (2009). Tensor decompositions and applications. *SIAM review*, *51*(3), 455–500.

Kosko, B. (1988). Bidirectional associative memories. *IEEE Transactions on Systems, man, and Cybernetics*, *18*(1), 49–60.

Kruskal, J. B. (1977). Three-way arrays: Rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear algebra and its applications*, *18*(2), 95–138.

Land, M., & Nilsson, D. (2012). *Animal eyes*. Oxford, UK, OUP Oxford.

LeCun, Y. (1998). The mnist database of handwritten digits. http://yann.%20lecun.%20com/exdb/mnist/

Lee, D. D., & Seung, H. S. (2001). Algorithms for non-negative matrix factorization, In *Advances in neural information processing systems*.

Lettvin, J. Y., Maturana, H. R., McCulloch, W. S., & Pitts, W. H. (1959). What the frog's eye tells the frog's brain. *Proceedings of the IRE*, *47*(11), 1940–1951.

Marr, D. (1982). *Vision*. San Francisco, CA, W. H. Freeman & Company.

Maudgalya, N., Olshausen, B. A., & Kent, S. J. (2020). Vector symbolic visual analogies, In *Aaai symposium on conceptual abstraction and analogy in natural and artificial intelligence*.

McClelland, J. L., & Kawamoto, A. H. (1986). Mechanisms of sentence processing: Assigning roles to constituents. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition, vol. 2: Psychological and biological models* (pp. 272–325). Cambridge, MA, MIT Press.

McCoy, R. T., Linzen, T., Dunbar, E., & Smolensky, P. (2019). RNNs implicitly implement tensor-product representations, In *International conference on learning representations*. https://openreview.net/forum?id=BJx0sjC5FX

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Memisevic, R., & Hinton, G. E. (2010). Learning to represent spatial transformations with factored higher-order boltzmann machines. *Neural computation*, *22*(6), 1473–1492.

Metcalf Eich, J. (1982). A composite holographic associative recall model. *Psychological Review*, *89*(6), 627.

Metcalf Eich, J. (1985). Levels of processing, encoding specificity, elaboration, and charm. *Psychological Review*, *92*(1), 1.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, *26*, 3111–3119.

Milne, E., Swettenham, J., Hansen, P., Campbell, R., Jeffries, H., & Plaisted, K. (2002). High motion coherence thresholds in children with autism. *Journal of Child Psychology and Psychiatry*, *43*(2), 255–263.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Moradshahi, M., Palangi, H., Lam, M. S., Smolensky, P., & Gao, J. (2019). Hubert untangles bert to improve transfer across nlp tasks. *arXiv preprint arXiv:1910.12647*.

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

Murdock, B. B. (1982). A theory for the storage and retrieval of item and associative information. *Psychological Review*, *89*(6), 609.

Olshausen, B. A., Anderson, C. H., & Van Essen, D. C. (1993). A neurobiological model of visual attention and invariant pattern recognition based on dynamic routing of information. *Journal of Neuroscience*, *13*(11), 4700–4719.

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation, In *Proceedings of the 2014 conference on empirical methods in natural language processing (emnlp)*.

Personnaz, L., Guyon, I., & Dreyfus, G. (1986). Collective computational properties of neural networks: New learning mechanisms. *Physical Review A*, *34*(5), 4217.

Pitts, W., & McCulloch, W. S. (1947). How we know universals the perception of auditory and visual forms. *The Bulletin of mathematical biophysics*, *9*(3), 127–147.

Plate, T. A. (2000). Analogy retrieval and processing with distributed vector representations. *Expert systems*, *17*(1), 29–40.

Plate, T. A. (1994). *Distributed representations and nested compositional structure* (Doctoral dissertation). University of Toronto.

Plate, T. A. (2003). *Holographic reduced representation: Distributed representation of cognitive structure*. Stanford, CA, CSLI Publications.

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

Plate, T. A. (1991). Holographic reduced representations: Convolution algebra for compositional distributed representations, In *International joint conference on artificial intelligence*.

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

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Rahimi, A., Datta, S., Kleyko, D., Frady, E. P., Olshausen, B., Kanerva, P., & Rabaey, J. M. (2017). High-dimensional computing as a nanoscalable paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *64*(9), 2508–2521.

Reed, S., Sohn, K., Zhang, Y., & Lee, H. (2014). Learning to disentangle factors of variation with manifold interaction, In *International conference on machine learning*.

Reed, S., Zhang, Y., Zhang, Y., & Lee, H. (2015). Deep visual analogy-making, In *Advances in neural information processing systems*.

Ross, B. H. (1989). Distinguishing types of superficial similarities: Different effects on the access and use of earlier problems. *Journal of Experimental Psychology: Learning, memory, and cognition*, *15*(3), 456.

Sabour, S., Frosst, N., & Hinton, G. E. (2017). Dynamic routing between capsules, In *Advances in neural information processing systems*.

Sidiropoulos, N. D., & Bro, R. (2000). On the uniqueness of multilinear decomposition of n-way arrays. *Journal of Chemometrics: A Journal of the Chemometrics Society*, *14*(3), 229–239.

Sidiropoulos, N. D., De Lathauwer, L., Fu, X., Huang, K., Papalexakis, E. E., & Faloutsos, C. (2017). Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing*, *65*(13), 3551–3582.

Singh, M. P., Chengxiang, Z., & Dasgupta, C. (1995). Fixed points in a hopfield model with random asymmetric interactions. *Physical Review E*, *52*(5), 5261.

Smolensky, P. (1992). *Principles for an integrated connectionist symbolic theory of higher cognition* (tech. rep. Technical Report CU-CS-600-92). University of Colorado at Boulder.

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

Sohl-Dickstein, J., Wang, C. M., & Olshausen, B. A. (2010). An unsupervised algorithm for learning lie group transformations. *arXiv preprint arXiv:1001.1027*.

Spall, J. C. (2005). *Introduction to stochastic search and optimization: Estimation, simulation, and control* (Vol. 65). John Wiley & Sons.

Tenenbaum, J. B., & Freeman, W. T. (2000). Separating style and content with bilinear models. *Neural computation*, *12*(6), 1247–1283.

Thagard, P., Holyoak, K. J., Nelson, G., & Gochfeld, D. (1990). Analog retrieval by constraint satisfaction. *Artificial intelligence*, *46*(3), 259–310.

Treisman, A. M., & Gelade, G. (1980). A feature-integration theory of attention. *Cognitive psychology*, *12*(1), 97–136.

Tucker, L. R. (1963). Implications of factor analysis of three-way matrices for measurement of change. In C. W. Harris (Ed.), *Problems in measuring change* (pp. 122–137). Madison WI, University of Wisconsin Press.

Tucker, L. R. (1966). Some mathematical notes on three-mode factor analysis. *Psychometrika*, *31*(3), 279–311.

Tulsiani, S., Gupta, S., Fouhey, D., Efros, A. A., & Malik, J. (2018). Factoring shape, pose, and layout from the 2d image of a 3d scene, In *Computer vision and pattern regognition (cvpr)*.

Van Vreeswijk, C., & Sompolinsky, H. (1996). Chaos in neuronal networks with balanced excitatory and inhibitory activity. *Science*, *274*(5293), 1724–1726.

Vasilescu, M. A. O., & Terzopoulos, D. (2002). Multilinear analysis of image ensembles: Tensorfaces, In *European conference on computer vision*. Springer.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*, 5998–6008.

Von der Malsburg, C. (1995). Binding in models of perception and brain function. *Current opinion in neurobiology*, *5*(4), 520–526.

Whitehead, A. N., & Russell, B. (1925). *Principia mathematica*. Cambridge, England, Cambridge University Press.

Willshaw, D. (1981). Holography, associative memory, and inductive generalization. In G. E. Hinton & J. A. Anderson (Eds.), *Parellel models of associative memory* (pp. 103–124). Hillsdale, N.J., Laurence Erlbaum Associates.

Wolfe, J. M., & Cave, K. R. (1999). The psychophysical evidence for a binding problem in human vision. *Neuron*, *24*(1), 11–17.

Wu, T. F., Li, H., Huang, P.-C., Rahimi, A., Rabaey, J. M., Wong, H.-S. P., Shulaker, M. M., & Mitra, S. (2018). Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study, In *2018 ieee international solid-state circuits conference-(isscc)*. IEEE.

Xu, Z.-B., Hu, G.-Q., & Kwong, C.-P. (1996). Asymmetric hopfield-type networks: Theory and applications. *Neural Networks*, *9*(3), 483–501.