ARTICLE **Communicated by Aapo Hyvarinen**

Resonator Networks, 2: Factorization Performance and Capacity Compared to Optimization-Based Methods

Spencer J. Kent
spencer.kent@berkeley.edu
Redwood Center for Theoretical Neuroscience and Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720, U.S.A.

E. Paxon Frady
epaxon@berkeley.edu
Friedrich T. Sommer
fsommer@berkeley.edu
Redwood Center for Theoretical Neuroscience and Helen Wills Neuroscience Institute, University of California, Berkeley, Berkeley, CA 94720, U.S.A., and Intel Laboratories, Neuromorphic Computing Lab, San Francisco, CA 94111, U.S.A.

Bruno A. Olshausen
baolshausen@berkeley.edu
Redwood Center for Theoretical Neuroscience, Helen Wills Neuroscience Institute, and School of Optometry, University of California, Berkeley, Berkeley, CA 94720, U.S.A.

We develop theoretical foundations of resonator networks, a new type of recurrent neural network introduced in Frady, Kent, Olshausen, and Sommer (2020), a companion article in this issue, to solve a high-dimensional vector factorization problem arising in Vector Symbolic Architectures. Given a composite vector formed by the Hadamard product between a discrete set of high-dimensional vectors, a resonator network can efficiently decompose the composite into these factors. We compare the performance of resonator networks against optimization-based methods, including Alternating Least Squares and several gradient-based algorithms, showing that resonator networks are superior in several important ways. This advantage is achieved by leveraging a combination of nonlinear dynamics and searching in superposition, by which estimates of the correct solution are formed from a weighted superposition of all possible solutions. While the alternative methods also search in superposition, the dynamics of resonator networks allow them to strike a more effective balance between exploring the solution space and exploiting local information to drive the network toward probable solutions. Resonator networks are not guaranteed to converge, but within a particular regime they almost always do. In exchange for relaxing the guarantee of global

Neural Computation **32, 2332–2388** (2020)
https://doi.org/10.1162/neco_a_01329

convergence, resonator networks are dramatically more effective at finding factorizations than all alternative approaches considered.

1 Introduction

This article is the second in a two-part series on resonator networks. "Resonator Networks, 1" shows how distributed representations of data structures may be formed using the algebra of Vector Symbolic Architectures and that decoding these representations often requires solving a vector factorization problem. We introduced resonator networks as a neural solution to this problem and demonstrated with two examples. Here, our primary objective is to establish the theoretical foundations of resonator networks and to perform a more comprehensive analysis of their convergence and capacity properties in comparison to optimization-based methods.

We limit our analysis 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. Part 1 included commentary on the historical context and representational power of VSAs, which we will not cover here. For the purposes of this article, it is sufficient to stipulate that wherever VSAs are used to encode complex hierarchical data structures, a factorization problem must be solved. By solving this problem, resonator networks make the VSA framework scalable to a larger range of problems.

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 3. There are in fact many ways to search in superposition, and we introduce a number of them in section 5 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 4.1 and 4.2, 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 (see section 6.1).

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 6.2).
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 6.2).
4. We propose a theory for why resonator networks perform well on this problem (see section 6.6).

2 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 $\mathbb{X}_1,\ \mathbb{X}_2, \ldots, \mathbb{X}_F$:

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

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

The factorization problem we wish to study is

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

Our assumption in this article 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 problem 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 approximately 125 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.

3 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 simultaneously 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 article. 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 article, we consider the identity $g : \mathbf{x} \mapsto \mathbf{x}$, the sign function $g : \mathbf{x} \mapsto \operatorname{sgn}(\mathbf{x})$, and nothing else. Other activation functions are appropriate for the other variants of resonator networks (e.g., 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 $\hat{\mathbf{c}}$:

$$\hat{\mathbf{c}} := \hat{\mathbf{x}}^{(1)} \odot \hat{\mathbf{x}}^{(2)} \odot \cdots \odot \hat{\mathbf{x}}^{(F)} = g(\mathbf{X}_1\mathbf{a}_1) \odot g(\mathbf{X}_2\mathbf{a}_2) \odot \cdots \odot g(\mathbf{X}_F\mathbf{a}_F). \quad (3.1)$$

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

$$\hat{\mathbf{c}} = \hat{\mathbf{x}}^{(1)} \odot \hat{\mathbf{x}}^{(2)} \odot \cdots \odot \hat{\mathbf{x}}^{(F)} = \mathbf{X}_1\mathbf{a}_1 \odot \mathbf{X}_2\mathbf{a}_2 \odot \cdots \odot \mathbf{X}_F\mathbf{a}_F. \quad (3.2)$$

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 $\mathbf{c}$:

$$\hat{\mathbf{c}} = \sum_{d_1,d_2,\ldots,d_F} \Big((\mathbf{a}_1)_{d_1}(\mathbf{a}_2)_{d_2}\ldots(\mathbf{a}_F)_{d_F}\Big)\, \mathbf{x}^{(1)}_{d_1} \odot \mathbf{x}^{(2)}_{d_2} \odot \cdots \odot \mathbf{x}^{(F)}_{d_F}, \quad (3.3)$$

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 $\big((\mathbf{a}_1)_{d_1}(\mathbf{a}_2)_{d_2}\ldots(\mathbf{a}_F)_{d_F}\big)$ can be approximately recovered from $\hat{\mathbf{c}}$ alone by computing the cosine similarity between $\hat{\mathbf{c}}$ and the vector $\mathbf{x}^{(1)}_{d_1} \odot \mathbf{x}^{(2)}_{d_2} \odot \cdots \odot \mathbf{x}^{(F)}_{d_F}$. The source of noise in this approximation is the fact that $\mathbf{x}^{(1)}_{d_1} \odot \mathbf{x}^{(2)}_{d_2} \odot \cdots \odot \mathbf{x}^{(F)}_{d_F}$ 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 $\big((\mathbf{a}_1)_{d_1}(\mathbf{a}_2)_{d_2}\ldots(\mathbf{a}_F)_{d_F}\big)$ 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:

$$\hat{\mathbf{o}}^{(f)} := \hat{\mathbf{x}}^{(1)} \odot \cdots \odot \hat{\mathbf{x}}^{(f-1)} \odot \hat{\mathbf{x}}^{(f+1)} \odot \cdots \odot \hat{\mathbf{x}}^{(F)}. \quad (3.4)$$

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 article 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

$$\hat{\mathbf{o}}^{(f)}[t] := \hat{\mathbf{x}}^{(1)}[t+1] \odot \cdots \odot \hat{\mathbf{x}}^{(f-1)}[t+1] \odot \hat{\mathbf{x}}^{(f+1)}[t] \odot \cdots \odot \hat{\mathbf{x}}^{(F)}[t], \tag{3.5}$$

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.

4 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

$$\hat{\mathbf{x}}^{(f)}[t+1] = \mathrm{sgn}\Big(\mathbf{X}_f \mathbf{X}_f^\top \big(\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}\big)\Big). \tag{4.1}$$

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 $\mathrm{sgn}(\cdot)$ 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 data set—in this case, the codevectors.

Prior to thresholding in equation 4.1, the matrix-vector product $\mathbf{X}^\top \big(\hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{c}\big)$ produces coefficients $\mathbf{a}_f[t]$ that, 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 (OLS) projection onto $\mathcal{R}(\mathbf{X}_f)$. This motivates the second variant of our model, resonator networks with OLS weights:

$$\begin{aligned}\hat{\mathbf{x}}^{(f)}[t+1] &= \operatorname{sgn}\Big(\mathbf{X}_f\big(\mathbf{X}_f^\top\mathbf{X}_f\big)^{-1}\mathbf{X}_f^\top\big(\hat{\mathbf{o}}^{(f)}[t]\odot\mathbf{c}\big)\Big)\\ &:= \operatorname{sgn}\Big(\mathbf{X}_f\mathbf{X}_f^\dagger\big(\hat{\mathbf{o}}^{(f)}[t]\odot\mathbf{c}\big)\Big),\end{aligned} \tag{4.2}$$

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

If, contrary to what we have defined in equations 4.1 and 4.2, the input to each subpopulation 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 elaborate on in the following sections. We imagine F separate subpopulations of neurons that 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 equation 3.5 follows the asynchronous convention, which we find to converge faster. The formulation given in the companion article in this issue employed the synchronous convention for pedagogical reasons, but the distinction between the two vanishes in continuous time, where updates are instantaneous.

In practice, we 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 $\hat{\mathbf{x}}^{(f)}[0] = \operatorname{sgn}\Big(\sum_j \mathbf{x}_j^{(f)}\Big)$, which, as we have said, has approximately equal cosine similarity to each term in the sum.

4.1 Difference between OP Weights and OLS Weights. The difference between outer product weights and OLS weights is via $\big(\mathbf{X}_f^\top\mathbf{X}_f\big)^{-1}$, 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 above 5000) and the codevectors are chosen randomly independent and identicaly distributed (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 6.2, we define and measure a performance metric called operational capacity in such a way that does not particularly highlight the difference between the dynamics, that is, 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 OLS 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 establish in section 6.5, this makes resonator networks (and, more generally, VSAs) a good fit for emerging device nanotechnologies (Rahimi et al., 2017).

5 The Optimization Approach

An alternative strategy for solving the factorization problem is to define a loss function that compares the current estimate $\hat{\mathbf{c}} := \hat{\mathbf{x}}^{(1)} \odot \hat{\mathbf{x}}^{(2)} \odot \cdots \odot \hat{\mathbf{x}}^{(F)}$ 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 like 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:

$$\begin{aligned} &\underset{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_F}{\text{minimize}} \quad \mathcal{L}\big(\mathbf{c}, g(\mathbf{X}_1\mathbf{a}_1) \odot g(\mathbf{X}_2\mathbf{a}_2) \odot \cdots \odot g(\mathbf{X}_F\mathbf{a}_F)\big) \\ &\text{subject to} \quad \mathbf{a}_1 \in C_1, \mathbf{a}_2 \in C_2, \ldots, \mathbf{a}_F \in C_F. \end{aligned} \tag{5.1}$$

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 article, 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 develop in section 6, is that resonator networks strike a more natural balance between exploring the high-dimensional state space and using local information to move toward the solution. We briefly introduce the benchmark algorithms in section 5.1, but discuss each at some length in the appendixes, including Table 2, which compiles the dynamics specified by each. We provide implementations of each algorithm in the small software library that accompanies this article.[1]

5.1 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 3. 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 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: 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 problem 5.1 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).

5.1.1 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] This is through the composition of an affine function with a convex function.

next factor and repeats. Formally, the updates given by ALS are

$$\mathbf{a}_f[t+1] = \arg\min_{\mathbf{a}_f} \ \tfrac{1}{2} \left\| \mathbf{c} - \hat{\mathbf{o}}^{(f)}[t] \odot \mathbf{X}_f \mathbf{a}_f[t] \right\|_2^2$$
$$= \left(\boldsymbol{\xi}^\top \boldsymbol{\xi}\right)^{-1} \boldsymbol{\xi}^\top \mathbf{c} \quad \mid \quad \boldsymbol{\xi} := \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big)\, \mathbf{X}_f. \tag{5.2}$$

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, a few details make our problem different from what is normally studied (see appendix D). 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.

5.1.2 Gradient-Following Algorithms. Another natural strategy for solving problem 5.1 is to make updates that incorporate the gradient of $\mathcal{L}$ with respect to the coefficients; each of the next five algorithms does this in a particular way (we write out the gradients for both loss functions in appendix E). 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 section E.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 step size will converge. Instead, the algorithms we apply to the squared error loss utilize a dynamic step size:

Iterative Soft Thresholding: The global minimizers of equation 5.1 are maximally sparse, $\|\mathbf{a}^\star_f\|_0 = 1$. 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 Thresholding (ISTA). The dynamics for ISTA are given by equation C.1 in Table 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 C.2.

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 (see equation C.3). A detailed discussion of this approach can be found in appendix G.

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 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 C.5 for the dynamics of Multiplicative Weights, as well as appendix H.

Map-seeking Circuits: The final algorithm that we considered is map-seeking circuits, 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 C.5 and appendix I).

5.2 Contrasting Resonator Networks with the Benchmarks.

5.2.1 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 appendixes D through I. That is, given any starting coefficients $\mathbf{a}_f[0]$, their dynamics reach fixed points that 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 nonconvexity of the optimization problem implies that this may not guarantee good local minima in practice. In section 6, 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 it is a main point of comparison with resonator networks.

5.2.2 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, Lugosi, & Massart, 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 section 6.

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 equation 5.2 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 $\left(\boldsymbol{\xi}^\top \boldsymbol{\xi}\right)^{-1}\boldsymbol{\xi}^\top$:

$$
\begin{aligned}
\hat{\mathbf{o}}^{(f)}[t] \in \{-1, 1\}^N \iff \left(\boldsymbol{\xi}^\top \boldsymbol{\xi}\right)^{-1}\boldsymbol{\xi}^\top &= \left(\mathbf{X}_f^\top \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big)^2 \mathbf{X}_f\right)^{-1} \mathbf{X}_f^\top \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big) \\
&= \left(\mathbf{X}_f^\top \mathbf{X}_f\right)^{-1} \mathbf{X}_f^\top \, \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big) \\
&= \mathbf{X}_f^\dagger \, \mathrm{diag}\big(\hat{\mathbf{o}}^{(f)}[t]\big). \qquad (5.3)
\end{aligned}
$$

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 4.2 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 ALS, 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 ALS, but *this is an exchange well worth making*, as we will show in section 6.

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 state space. 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 preactivation 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 nonsymmetric, which has a simple but important consequence: autoassociative networks with nonsymmetric weights cannot be guaranteed to converge in general. This result, first established by Cohen and 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, Hu, & Kwong, 1996), and, moreover, adding a degree of asymmetry has been advocated as a technique to reduce the influence of spurious fixed points (Hertz, Grinstein, & Solla, 1986; Singh, Chengxiang, & Dasgupta, 1995; Chengxiang, Dasgupta, & Singh, 2000).

Resonator networks have a large and practical regime of operation, where *M* (the problem size) is small enough, in which nonconverging 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 & 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 that 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 5.1, 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 toward 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 sizable 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 establish in section 6.6.

6 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. We choose vectors randomly 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.

6.1 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—that is, 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 6.2.

We first note that this analysis is necessary only for resonator networks with outer product weights; OLS 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 $\boldsymbol{\Gamma} := \mathbf{X}_f\mathbf{X}_f^{\top}\big(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\big)$ 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 $\big(\mathbf{x}_{\star}^{(f)}\big)_i$, that is, that there is a bit flip 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 (1985, 1987) established that in Hopfield networks, an avalanche phenomenon occurs where bit flips accumulate and the network becomes essentially useless for values of $D_f > 0.138N$, at which point the approximate bit flip probability is 0.0036. While we don't attempt any of this complicated analysis on resonator networks, we do derive an expression for the bit flip probability of any particular factor that accounts for bit flips that "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 bit flip 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 J for a simple derivation.) This is summarized as the *Hopfield bit flip probability* h_f:

$$\begin{aligned} h_f &:= Pr\left[\, \left(\hat{\mathbf{x}}^{(f)}[1]\right)_i \neq \left(\mathbf{x}_\star^{(f)}\right)_i \,\right] \\ &= \Phi\left(\frac{-N - D_f + 1}{\sqrt{(N-1)(D_f-1)}} \right), \end{aligned} \tag{6.1}$$

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 bit flip probability is $\Phi\left(\frac{-N}{\sqrt{(N-1)(D_f-1)}}\right)$. 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 J), 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 J, we derive the bit flip 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 bit flip probability. There are four relevant probabilities:

$$r_f := Pr\left[\, \left(\hat{\mathbf{x}}^{(f)}[1]\right)_i \neq \left(\mathbf{x}_\star^{(f)}\right)_i \,\right], \tag{6.2}$$

$$n_f := Pr\left[\, \left(\hat{\mathbf{o}}^{(f+1)}[0] \odot \mathbf{c}\right)_i \neq \left(\mathbf{x}_\star^{(f+1)}\right)_i \,\right], \tag{6.3}$$

$$r_{f'} := Pr\left[\, \left(\hat{\mathbf{x}}^{(f)}[1]\right)_i \neq \left(\mathbf{x}_\star^{(f)}\right)_i \;\middle|\; \left(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\right)_i = \left(\mathbf{x}_\star^{(f)}\right)_i \,\right], \tag{6.4}$$

$$r_{f''} := Pr\left[\, \left(\hat{\mathbf{x}}^{(f)}[1]\right)_i \neq \left(\mathbf{x}_\star^{(f)}\right)_i \;\middle|\; \left(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\right)_i \neq \left(\mathbf{x}_\star^{(f)}\right)_i \,\right]. \tag{6.5}$$

Equation 6.2 is the probability of a bit flip compared to the correct value, the *resonator bit flip probability*. Equation 6.3 gives the probability that the *next* factor will see a net bit flip, a bit flip that has percolated through the previous factors. Equations 6.4 and 6.5 give the probability of a bit flip conditioned on whether this factor sees a net bit flip, and they are *different*. It should be obvious that

$$r_f = r_{f'}(1 - n_{f-1}) + r_{f''}n_{f-1} \tag{6.6}$$

and also that

$$n_f = r_{f'}(1 - n_{f-1}) + (1 - r_{f''})n_{f-1}. \tag{6.7}$$

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

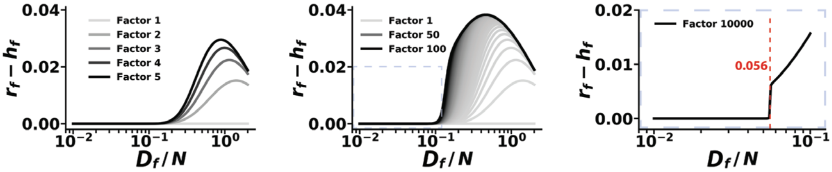
$$r_{f'} = \Phi\left(\frac{-N(1 - 2n_{f-1}) - (D_f - 1)}{\sqrt{(N-1)(D_f - 1)}}\right), \tag{6.8}$$

$$r_{f''} = \Phi\left(\frac{-N(1 - 2n_{f-1}) + (D_f - 1)}{\sqrt{(N-1)(D_f - 1)}}\right). \tag{6.9}$$

The resonator bit flip 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 14 in the appendix, which measures r_f).

The main analytical result in this section is the sequence of equations 6.6 to 6.9, which allow one to compute the bit flip 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 bit flip probability for a Hopfield network (compare equations 6.8 and 6.9 against equation 6.1). 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 bit flip 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 bit flip in factor f gets passed onto



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

Figure 1: Extra bit flip 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 $r_f - h_f$).

factors $f+1$, $f+2$, and so on. This affects the bit flip 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 bit flip probability is 0.02275, which occurs at $D_f = N$. The ratio D_f/N is what determines the bit flip probability (see appendix J for an explanation). Percolated noise is measured by the difference $r_f - h_f$, which we plot in Figure 1. Panel (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 1. In the limit of large F, there appears to be a phase change in residual bit flip 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, and 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 $D_f/N \leq 0.056$, the stability of a resonator network with outer product weights is the same as the stability of a Hopfield network. For $D_f/N > 0.056$, percolated noise between the factors causes the resonator network to be strictly less stable than a Hopfield network**.

6.2 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 5.1. 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 1. *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 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. 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, 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 readers that our results in this article pertain to factorization of randomly drawn vectors that bear

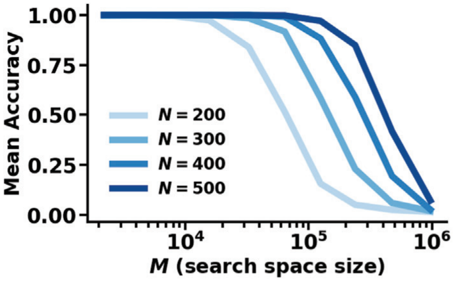


Figure 2: Accuracy as a function of M for resonator network with outer product weights. Three factors ($F = 3$); average over 5000 random trials.

no particular correlational structure, but that notions of total accuracy and operational capacity would be relevant, and specific, to factorization of nonrandom 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. 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 a range of values M for which the expected total accuracy is 1.0, beyond which this ceases to be the case. For all values of M within this range, the algorithm essentially always solves the factorization problem.

In this article, we estimate operational capacity when $p = 0.99$ (99% or more of factors were inferred correctly) and $k = 0.001M$ (the model can search over at most 1/1000 of the entire search space). These choices are largely practical: 99% or higher accuracy makes the model very reliable in practice, and this operating point can be estimated from a reasonable number (3000 to 5000) 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 consider only a small fraction of the possible factorizations. While a resonator network has no guarantee of convergence, it almost always converges in far fewer 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.

6.2.1 Resonator Networks Have Superior Operational Capacity. We estimated the operational capacity of the benchmark algorithms in addition to the two variants of resonator networks. Figure 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

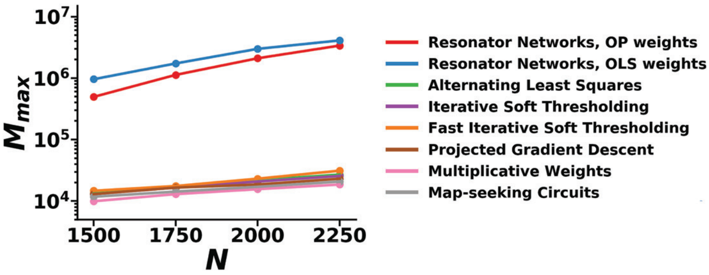


Figure 3: Operational capacity is dramatically higher for resonator networks (blue and red above) than for any of the benchmark algorithms. These points represent the size of factorization problem that can be solved reliably. Shown is operational capacity for $F = 3$ factors. The gap is similarly large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

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 B.)

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 article do not particularly highlight the difference between them. Except for ALS, 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 step sizes make a comparison of the number of iterations nonstandardized, 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.

6.2.2 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

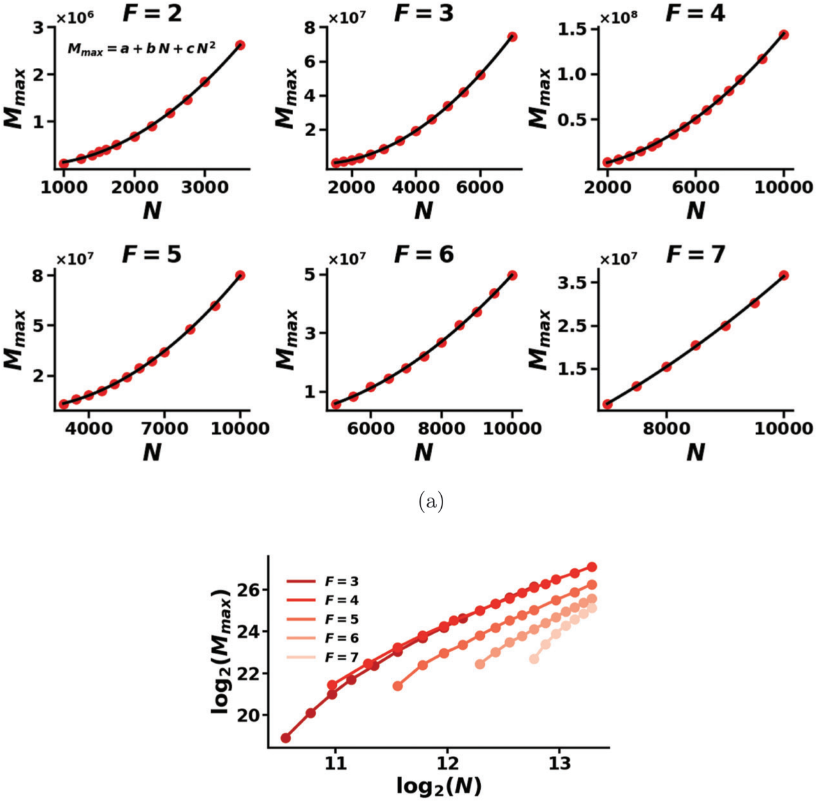
and k: p is the threshold for total accuracy, and k is the maximum number of iterations the algorithm is allowed to take (refer to definition 1). Here we report operational capacity for $p = 0.99$ and $k = 0.001M$ on randomly sampled codevectors. The operational capacity is specific to these choices, which are practical for Vector Symbolic Architectures.

Our simulations revealed that, empirically, **resonator network operational capacity $M_{\max}$ scales as a quadratic function of N**, which w e illustrate in Figure 4. The points in this figure are estimated from many thousands of random trials, over a range of values for F and N. In panel (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 panel (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 B 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 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 that accompanies this article[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 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 traveling 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 (see problem 2.1), can be solved by resonator

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



(a)

(b)

Figure 4: Operational capacity of resonator networks with OP weights. (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 B. (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 $F = 2$ were omitted to better convey the trend for $F = 3$ and higher, but see appendix B for the full picture.

networks. Supposing, however, that some other hard problem can be cast in the form of equation 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 ($D_1 = D_2 = \cdots = D_F = \sqrt[F]{M}$), 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 $D_1 \neq D_2 \neq \cdots \neq D_f$. 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:

$$\xi := \left(\min_f D_f \right) \Big/ \left(\max_f D_f \right). \tag{6.10}$$

For $\xi \geq 0.2$ we found that the effect on $M_{\max}$ was simply an additive factor that 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 that 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 4 is fairly suggestive of some underlying law, and we are hopeful that a theoretical explanation exists, waiting to be discovered.

6.3 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 5 shows that it is not. We plot the mean number of iterations over 5000 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 fewer than the $0.001M$ cutoff we used in the simulations of section 6.2; the algorithm is only ever considering a tiny fraction of the possible factorizations before it finds the solution.

Section 6.2.1 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

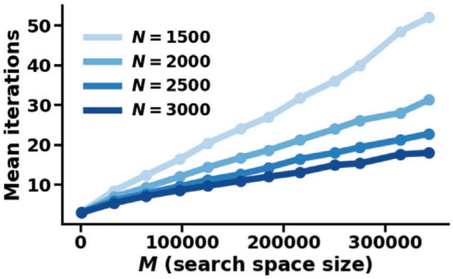


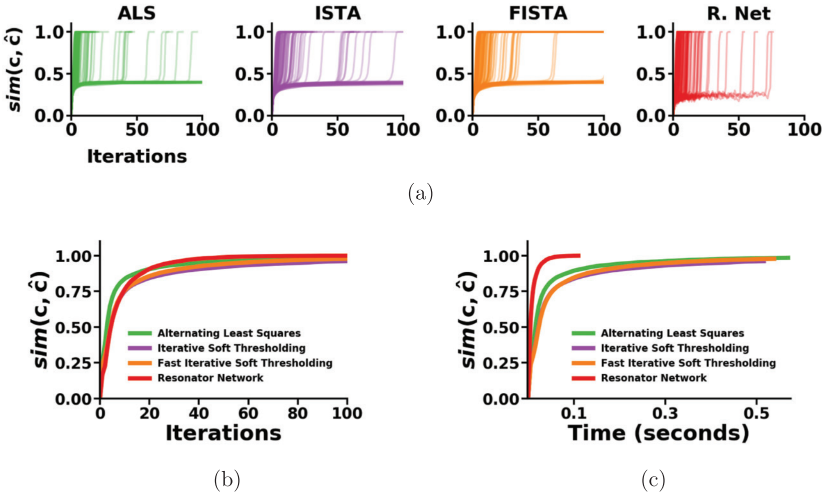
Figure 5: Iterations until convergence; resonator network with outer product weights and $F = 3$. The number of iterations is a very small compared to the size of the search space.

factorizations. This result, however, does not directly demonstrate the relative speed with which factorizations 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 6.

Measured in number of iterations, resonator networks are comparable to the benchmark algorithms. We noted that ALS is the greediest of the benchmarks, and one can see from Figure 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 6b and 6c.

6.4 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 7:

- For M small enough, almost all trajectories converge. 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 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.



(a)

(b)

(c)

Figure 6: Our benchmark of factorization speed. Implementation in Python with NumPy. Run on machine with Intel Core i7-6850k processor and 32 GB RAM. We generated 5000 random instantiations of the factorization problem with $N = 1500$, $F = 3$, and $D_f = 40$, running each of the four algorithms in turn. (a) Convergence traces for 100 randomly drawn factorization problems (out of total 5000); 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 sizable fraction (around 30%) of the instances are not solved by the benchmark algorithms, at least within 100 iterations. (b) Average cosine similarity versus iteration number (only trials with accuracy 1.0). (c) Average cosine similarity versus wall-clock time (only trials with accuracy 1.0).

- As M increases, nonconverging trajectories appear in greater proportion and yield incorrect factorizations. Any trajectories that 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 reemerge, and they yield the incorrect factorization.

In theory, limit cycles of any length may appear, although in practice, they tend to be skewed toward 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 7 is that the basins of attraction become very narrow and hard to find for the resonator network dynamics.

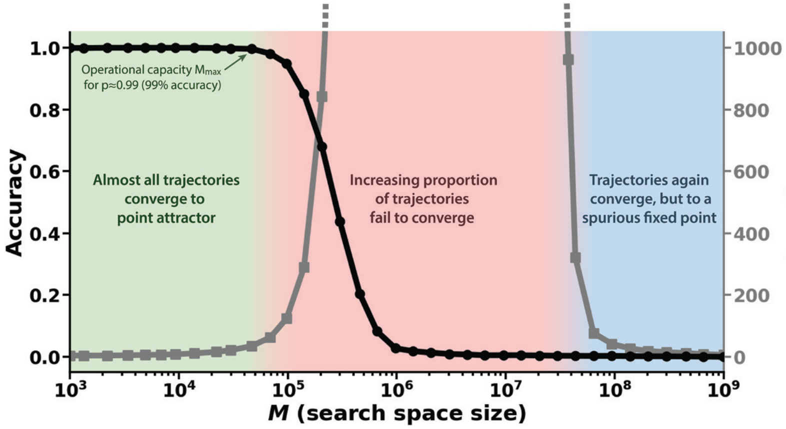


Figure 7: Regimes of different convergence behavior. Curves show measurement from simulation of an outer product resonator network with three factors and $N = 400$. This is also meant as a diagram of convergence behavior for resonator networks in general. Shown in black is the average decoding accuracy, 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 bit flip probability that is zero only asymptotically in M (see section 6.1 and appendix J).

The algorithm will wander, since it has so few spurious fixed points (see section 6.6), but not be able to find any basin of attraction.

6.5 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 8 that a resonator network can still

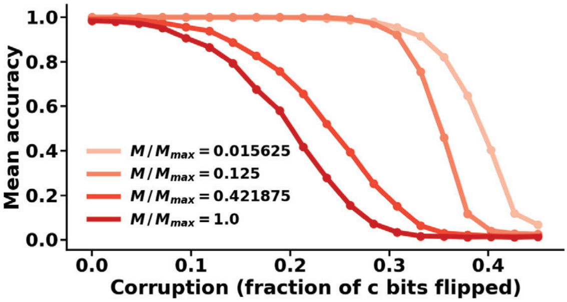


Figure 8: Factoring a corrupted **c**. For M well below capacity (lighter curves above), one can sustain heavy corruption to **c** and still find the correct factorization.

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.

6.6 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 (PGD; 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 PGD (as it is with the OLS variant of resonator networks), but that incorrect factorizations are much more likely to be fixed points under PGD. The definition of PGD can be found in Table 2, with some comments in appendix G.

6.6.1 Stability of the Correct Factorization. The vector of coefficients $\mathbf{a}_f$ is a fixed point of PGD dynamics when the gradient at this point is exactly $\mathbf{0}$ or when it is in the null space of the projection operator. We write

$$\mathcal{N}\big(\mathcal{P}_{C_f}[\mathbf{x}]\big) := \{\mathbf{z} \mid \mathcal{P}_{C_f}\big[\mathbf{x}+\mathbf{z}\big] = \mathcal{P}_{C_f}\big[\mathbf{x}\big]\} \tag{6.11}$$

to denote this set of points. The null space 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 null space of the projection operator at a vertex of the simplex is an intersection of half-spaces (each half-space given by an edge of the simplex). We can compactly represent it with the following expression:

$$\mathcal{N}\big(\mathcal{P}_{\Delta_{D_f}}[\mathbf{e}_i]\big) = \big\{\mathbf{z} \mid \bigcap_{j\neq i} (\mathbf{e}_i - \mathbf{e}_j)^\top \mathbf{z} \geq 1\big\}, \tag{6.12}$$

An equivalent way to express the null space is

$$\mathcal{N}\big(\mathcal{P}_{\Delta_{D_f}}[\mathbf{e}_i]\big) = \big\{\mathbf{z} \mid z_j \leq z_i - 1 \ \ \forall j \neq i\big\}. \tag{6.13}$$

In other words, for a vector to be in the null space 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 5.1) is always a fixed point under the dynamics of PGD.

This matches the stability of OLS resonator networks, which are, by construction, always stable at the correct factorization. We showed in section 6.1 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.

6.6.2 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 fixed points. The same is true for a resonator network with OP weights because each element of the vector $\mathbf{X}_f\mathbf{X}_f^\top\big(\hat{\mathbf{o}}^{(f)}[0] \odot \mathbf{c}\big)$ is approximately gaussian with mean zero (see section 6.1 and appendix J).

Contrast this against PGD. We recall from equation 6.13 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, PGD is much more stable at incorrect factorizations. Empirically, the failure mode of PGD involves it settling on one of these spurious fixed points.

6.6.3 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 PGD, 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, **PGD 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 nonuniform way, but it does suggest that PGD 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.

6.6.4 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 $\mathbf{a}^\star_f$), or we might nudge it toward any of the other vertices, away from $\mathbf{a}^\star_f$. We can parameterize this with a single scalar θ and $\mathbf{e}_i$ chosen uniformly among the possible vertices:

$$\mathbf{a}_f[0] = \theta \mathbf{e}_i + (1-\theta)\frac{1}{D_f}\mathbf{1} \quad | \quad \theta \in [0,1], \;\; i \sim \mathcal{U}\{1, D_f\}. \tag{6.14}$$

We ran a simulation with $N = 1500$ and $D_1 = D_2 = D_3 = 50$, at which PGD and Multiplicative Weights have a total accuracy of 0.625 and 0.525, respectively. We created 5000 random factorization problems, initializing the state according to equation 6.14 and allowing the dynamics to run until

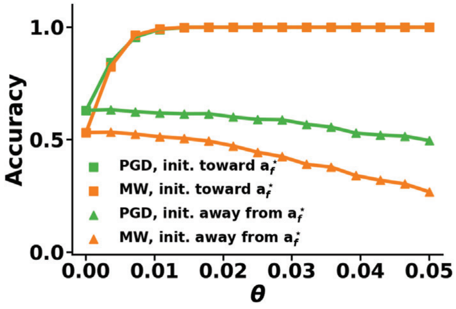


Figure 9: States in hypercube interior get pulled into spurious basins of attraction. PGD is in green and Multiplicative Weights in orange. Network is initialized at a distance θ from the center of the simplex (see equation 6.14), and allowed to converge. The y-axis is the accuracy of the factorization implied by the converged state. Triangles indicate initialization slightly away from $\mathbf{a}^\star_f$ 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.

convergence. We did this first with a nudge toward the correct factorization $\mathbf{a}^\star_f$ (squares in Figure 9) and then with a nudge away from $\mathbf{a}^\star_f$, toward a randomly chosen spurious factorization (triangles in Figure 9).

What Figure 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 toward 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 PGD and, in Figure 9, Multiplicative Weights. This choice was made for clarity, and similar arguments apply for all of the benchmarks. While the details may differ slightly (e.g., 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, PGD has many more spurious fixed points than resonator networks**. We have also shown that **trajectories moving through**

the interior of the hypercube are easily pulled into these spurious basins of attraction.

7 Discussion

We studied a vector factorization problem that arises in the use of Vector Symbolic Architectures (as introduced in the companion article in this issue) 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 $\mathbb{R}^N$ (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 that 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 a 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 article) 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 greedier than resonator networks. It appears that resonator networks strike a more natural balance between making updates based on the best-available local information and 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 converging only 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 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 article.

Tensor decomposition is a family of problems that bear some resemblance to the factorization problem we have introduced, problem 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 D how our factorization problem is different from traditional tensor decompositions. Our benchmarks actually included the standard tensor decomposition algorithm, Alternating Least Squares, reexpressed 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 D. 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, Hattori, & Yamazaki, 2002). Subsequent efforts to extend this model to factorizations with three or more factors (Huang & Hagiwara, 1999; Kobayashi, Hattori, & Yamazaki, 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 article applies a different type of tensor decomposition, a Tucker decomposition (again see appendix D), to a variety of different real-valued data sets 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 ALS. 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) revisit the Tucker decomposition problem, but factor 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 $\mathbf{V} \approx \mathbf{W}\mathbf{H}$, the best known of which is probably nonnegative 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 **W** is often interpreted as a basis for the columns of **V**, with **H** containing the coefficients of each column with respect to this basis. In this sense, vectors are being added to explain **V** rather than combined multiplicatively; nonnegative matrix factorization is much closer to sparse coding (Hoyer, 2004).

The companion article in this issue 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 our companion article to more realistic data (e.g., complex three-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, an exciting avenue for future study. 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, a VSA developed by 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, introduced for the first time in this two-part series. We believe that as the theory and applications of resonator networks are further developed, they may help us understand factorization in the brain, which remains an important mystery.

Appendix A: Implementation Details

This appendix includes a few comments relevant to the implementation of resonator networks. Algorithm 1 gives pseudocode 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 $\mathbf{X}_f\mathbf{X}_f^\dagger(\hat{\mathbf{o}} \odot \mathbf{c})$ has lower computational complexity than actually forming a single synaptic matrix $\mathbf{T}_f := \mathbf{X}_f\mathbf{X}_f^\dagger$ and then computing $\mathbf{T}_f(\hat{\mathbf{o}} \odot \mathbf{c})$ 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.

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 $\big(\mathbf{x}_j^{(f)} = \mathbf{X}_f[\,:\,, j\,]\big)$
Require: k                                              ▷ Maximum allowed iterations
 1: $\hat{\mathbf{x}}^{(f)} \leftarrow \mathrm{sgn}\big(\sum_j \mathbf{x}_j^{(f)}\big) \quad \forall f = 1, \ldots, F$
 2: $\mathbf{X}_f^{\dagger} \leftarrow \mathrm{pinv}(\mathbf{X}_f) \quad \forall f = 1, \ldots, F$
 3: $i \leftarrow 0$
 4: while not converged and $i < k$ do
 5:     for $f = 1$ to $F$ do
 6:         $\hat{\mathbf{o}} \leftarrow \hat{\mathbf{x}}^{(1)} \odot \ldots \odot \hat{\mathbf{x}}^{(f-1)} \odot \hat{\mathbf{x}}^{(f+1)} \odot \ldots \odot \hat{\mathbf{x}}^{(F)}$
 7:         $\hat{\mathbf{x}}^{(f)} \leftarrow \mathrm{sgn}\Big(\mathbf{X}_f \mathbf{X}_f^{\dagger}\big(\hat{\mathbf{o}} \odot \mathbf{c}\big)\Big)$
 8:     end for
 9:     $i \leftarrow i + 1$
10: end while
11: for $f = 1$ to $F$ do                                ▷ Nearest Neighbor decoding
12:     $u \leftarrow \arg\max_j \, |\mathrm{sim}(\hat{\mathbf{x}}^{(f)}, \mathbf{x}_j^{(f)})|$   ▷ *Un-signed* NN w.r.t cos-similarity
13:     $\hat{\mathbf{x}}^{(f)} \leftarrow \mathbf{x}_u^{(f)}$
14: end for
15: return $\hat{\mathbf{x}}^{(f)} \; \forall f = 1, \ldots, F$
```

Lines 11 to 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 $\mathbf{c} = \mathbf{x}_{\star}^{(1)} \odot \mathbf{x}_{\star}^{(2)} \odot \cdots \odot \mathbf{x}_{\star}^{(F)}$ is the factorization we are searching for, we also have $\mathbf{c} = -\mathbf{x}_{\star}^{(1)} \odot -\mathbf{x}_{\star}^{(2)} \odot \cdots \odot \mathbf{x}_{\star}^{(F)}$, 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. Note 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 3.5) 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 use 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 happen instantaneously.

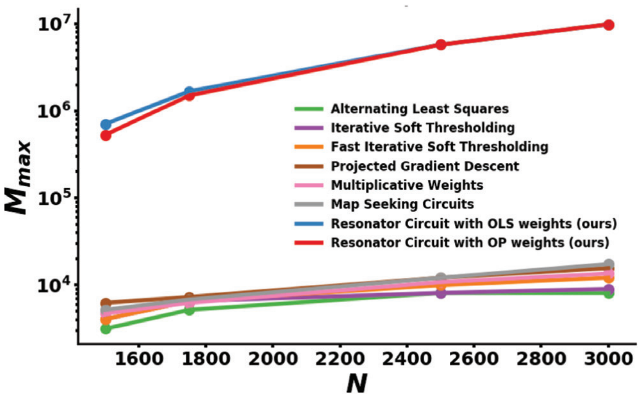


Figure 10: Comparing operational capacity against the benchmarks for $F = 4$ (four factors).

Throughout this article, 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.

Appendix B: Operational Capacity

The main text 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 10 compares operational capacity among all of the considered algorithms when F, the number of factors, is four. We previously showed this type of plot for $F = 3$, which was Figure 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 6.2 that the operational capacity of resonator networks scales quadratically in N, which was shown in Figure 4. In Table 1 we provide parameters of the least-squares quadratic fits shown in that plot. One can see from Figure 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 two to three factors and then

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

	Parameters of Quadratic Fit		
F	**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

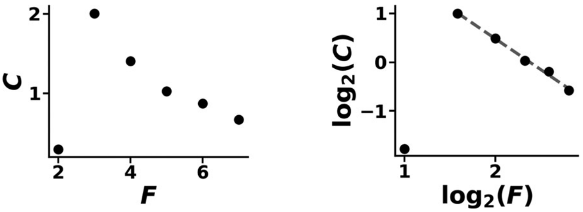


Figure 11: Parameter c of the quadratic scaling depends on F. We find that it may follow an inverse power law for $F \geq 3$.

falls with increasing F. This implies that factorization is easiest for resonator networks when the decomposition is into three factors, an interesting phenomenon for which we do not have an explanation at this time.

Figure 11 visualizes c as a function of F. The data indicate 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:

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

Appendix C: Table of Benchmark Algorithms

Table 2: Dynamics for $\mathbf{a}_f$, Benchmark Algorithms.

Algorithm	Dynamics for Updating $\mathbf{a}_f[t]$	Equation
Alternating Least Squares	$\mathbf{a}_f[t+1] = \left(\boldsymbol{\xi}^\top \boldsymbol{\xi}\right)^{-1} \boldsymbol{\xi}^\top \mathbf{c}$ $\boldsymbol{\xi} := \mathrm{diag}\left(\hat{\mathbf{o}}^{(f)}[t]\right) \mathbf{X}_f$	6.2
Iterative Soft Thresholding	$\mathbf{a}_f[t+1] = \mathcal{S}[\mathbf{a}_f[t] - \eta\, \nabla_{\mathbf{a}_f} \mathcal{L}\ ;\ \lambda\eta]$ $\left(\mathcal{S}[\,\mathbf{x}\,;\gamma\,]\right)_i := \mathrm{sgn}(x_i) \max(\lvert x_i \rvert - \gamma,\ 0)$	C.1
Fast Iterative Soft Thresholding	$\alpha_t = \dfrac{1 + \sqrt{1 + 4\alpha_{t-1}^2}}{2}$ $\beta_t = \dfrac{\alpha_{t-1} - 1}{\alpha_t}$ $\mathbf{p}_f[t+1] = \mathbf{a}_f[t] + \beta_t (\mathbf{a}_f[t] - \mathbf{a}_f[t-1])$ $\mathbf{a}_f[t+1] = \mathcal{S}[\mathbf{p}_f[t+1] - \eta\, \nabla_{\mathbf{p}_f} \mathcal{L}\ ;\ \lambda\eta]$ $\left(\mathcal{S}[\,\mathbf{x}\,;\gamma\,]\right)_i := \mathrm{sgn}(x_i) \max(\lvert x_i \rvert - \gamma,\ 0\,)$	C.2
Projected Gradient Descent	$\mathbf{a}_f[t+1] = \mathcal{P}_{C_f}\left[\mathbf{a}_f[t] - \eta \nabla_{\mathbf{a}_f} \mathcal{L}\right]$ $\mathcal{P}_{C_f}[\mathbf{x}] := \underset{\mathbf{z} \in C_f}{\arg\min}\ \dfrac{1}{2} \left\lVert \mathbf{x} - \mathbf{z} \right\rVert_2^2$	C.3
Multiplicative Weights	$\mathbf{w}_f[t+1] = \mathbf{w}_f[t] \odot \left(\mathbf{1} - \dfrac{\eta}{\rho} \nabla_{\mathbf{a}_f} \mathcal{L}\right)$ $\mathbf{a}_f[t+1] = \dfrac{\mathbf{w}_f[t+1]}{\sum_i w_{fi}[t+1]}$ $\rho := \max\limits_i \left\lvert (\nabla_{\mathbf{a}_f} \mathcal{L})_i \right\rvert$	C.4
Map Seeking Circuits	$\mathbf{a}_f[t+1] = \mathcal{T}\left(\mathbf{a}_f[t] - \eta\left(\mathbf{1} + \dfrac{1}{\rho} \nabla_{\mathbf{a}_f} \mathcal{L}\right);\ \epsilon\ \right)$ $\mathcal{T}\left(\mathbf{x}\,;\,\epsilon\,\right)_i := \begin{cases} x_i & \text{if } x_i \geq \epsilon \\ 0 & \text{otherwise} \end{cases}$ $\rho := \left\lvert \min\limits_i (\nabla_{\mathbf{a}_f} \mathcal{L})_i \right\rvert$	C.5

Note: See appendixes D to I for discussion of each algorithm, including hyperparameters η, λ, and ϵ, as well as initial conditions.

Appendix D: 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 these data can be expressed as a tensor has made the study of tensor decomposition a popular subfield of applied mathematics. 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 (1966), Carroll and Chang (1970), and Harshman (1970). This section highlights 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 2.1) decomposes an fth-order tensor $\mathcal{C}$ into a sum of tensors, each generated by the outer product $\circ$:

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

The outer product contains all pairs of components from its two arguments, so $\left(\mathbf{w} \circ \mathbf{x} \circ \mathbf{y} \circ \mathbf{z}\right)_{ijkl} = w_i x_j y_k z_l$. 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: Canonical Polyadic Decomposition, coined by Hitchcock; CANonical DECOMPosition (CANDECOMP), coined by Carroll and Chang (1970); and PARAllel FACtor analysis (PARAFAC), coined by Harshman (1970). We will simply call this the CP decomposition, in accordance with the convention used by Kolda and Bader (2009) and many others.

CP decomposition makes no mention of a codebook of vectors, such as we have in equation 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 $\mathbf{X}_f := [\mathbf{x}_1^{(f)}, \mathbf{x}_2^{(f)}, \ldots \mathbf{x}_R^{(f)}]$:

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

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 or more). 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 $\mathbf{U\Sigma V}^\top := \mathbf{X}_1\mathbf{X}_2^\top$, then any non-singular matrix $\mathbf{M}$ generates an equally good CP decomposition $(\mathbf{U\Sigma M})(\mathbf{VM}^{-1})^\top$. The decomposition is *highly* nonunique. 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 D.2, 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. See Sidiropoulos et al. (2017) for a more comprehensive comparison.

The vector factorization problem defined by equation 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, Pruzansky, & Kruskal, 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 showed in section 6.6 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 equation 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. ALS 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 ALS 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 the companion article in this issue. 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, De Moor, and Vandewalle (2000b). A discussion of applying tensor decompositions to statistical learning problems is covered by Anandkumar, Ge, Hsu, Kakade, and Telgarsky (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:

$$\mathcal{C} = \sum_{p=1}^{P} \sum_{q=1}^{Q} \cdots \sum_{r=1}^{R} g_{pq\ldots r} \, \mathbf{x}_p^{(1)} \circ \mathbf{x}_q^{(2)} \circ \cdots \circ \mathbf{x}_r^{(F)}. \tag{D.3}$$

This adds many more parameters compared to CP decomposition, a special case of Tucker decomposition when $\mathcal{G}$ is the identity. For the purpose of illustration, we reprint in Figure 12 (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, De Moor, and Vandewalle (2000a). The Tucker decomposition can also be found via ALS (see Kolda & Bader, 2009,

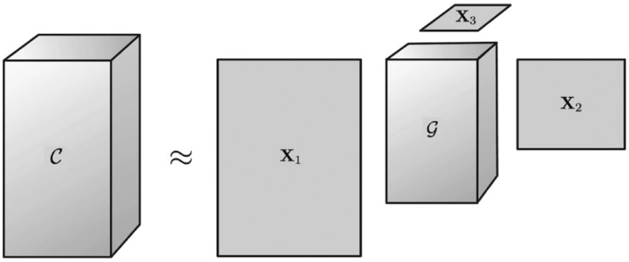


Figure 12: Tucker decomposition with three factors.

section 4.2, for a tutorial), although the problem is somewhat harder than CP decomposition, both by being computationally more expensive and by being nonunique. 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 data set 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, ALS, is not particularly competitive 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.

Appendix E: General Notes on Gradient-Based Algorithms

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

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

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 that 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

$$\begin{aligned}\nabla_{\mathbf{a}_f}\mathcal{L} &= \mathbf{X}_f^\top\Big(\big(\mathbf{c} - \hat{\mathbf{x}}^{(1)} \odot \cdots \odot \hat{\mathbf{x}}^{(F)}\big) \\ &\qquad \odot \big(- \hat{\mathbf{x}}^{(1)} \odot \cdots \odot \hat{\mathbf{x}}^{(f-1)} \odot \hat{\mathbf{x}}^{(f+1)} \odot \cdots \odot \hat{\mathbf{x}}^{(F)}\big)\Big) \\ &:= \mathbf{X}_f^\top\Big(\hat{\mathbf{x}}^{(f)} \odot \big(\hat{\mathbf{o}}^{(f)}\big)^2 - \mathbf{c} \odot \hat{\mathbf{o}}^{(f)}\Big). \end{aligned} \tag{E.2}$$

This looks somewhat similar to the gradient for the negative inner product: they differ by an additive term given by $\mathbf{X}_f^\top\Big(\hat{\mathbf{x}}^{(f)} \odot \big(\hat{\mathbf{o}}^{(f)}\big)^2\Big)$. At the vertices of the hypercube, all the elements of $\hat{\mathbf{x}}^{(f)}$ are 1 or -1 and the term $\big(\hat{\mathbf{o}}^{(f)}\big)^2$ 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 $\mathbf{x}_\star^{(1)}, \ldots, \mathbf{x}_\star^{(F)}$, which is not the case with the negative inner product. To be clear, equation E.1 is the gradient when the loss function is the negative inner product, while equation E.2 is the gradient when the loss function is the squared error.

E.1 Fixed-Step-Size 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 step size, 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:

$$\|\nabla\mathcal{L}(\mathbf{x}) - \nabla\mathcal{L}(\mathbf{y})\|_2 \leq L\|\mathbf{x} - \mathbf{y}\|_2 \;\; \forall\, \mathbf{x},\, \mathbf{y}. \tag{E.3}$$

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

$$\mathbf{0} \preceq \nabla^2\mathcal{L}(\mathbf{x}) \preceq L\mathbf{I} \quad \forall\, \mathbf{x}, \tag{E.4}$$

where $\mathbf{0}$ is the matrix of all zeros and and $\mathbf{I}$ is the identity. Absent some procedure for adjusting the step size 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 article. 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-step-size 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 F.1). 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.

Appendix F: Iterative Soft Thresholding and Fast Iterative Soft Thresholding

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

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

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 $\lambda\|\mathbf{a}_f\|_1$:

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

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 ISTA (Daubechies, Defrise, & De Mol, 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 C.1 and there are a few parameters worth discussing. First, the dynamic step size η can be set via backtracking line search or, as we did, by computing the Lipschitz constant of the loss function gradient:

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

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 article, 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 article, but it has been shown in various places (Bredies & Lorenz, 2008, for instance) that ISTA will converge at a rate $\simeq \mathcal{O}(1/t)$. ISTA works well in practice, although for four 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 C.2, 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.

Appendix G: Projected Gradient Descent

Starting from the general optimization form of the factorization problem 5.1, 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 5.1, from which we can easily recover the specific solution to 2.1. This is due to the fact that

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

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 $-\mathbf{a}_1^\star, -\mathbf{a}_2^\star, -\mathbf{a}_3^\star, \ldots, -\mathbf{a}_F^\star$, and so on. 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, Wolfe, & Crowder, 1974; Duchi, Shalev-Shwartz, Singer, & Chandra, 2008; 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. Recall that we defined the null space of the projection operation with equation 6.11 in section 6.6, and the special case for the simplex constraint in equations 6.12 and 6.13.

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 section E.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.

Appendix H: 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 C.4 for the dynamics of Multiplicative Weights.) We choose a fixed step size $\eta \leq 0.5$ and initial values for the weights all one: $\mathbf{w}_f[0] = \mathbf{1}$. In experiments in this article 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, Hazan, and Kale, 2012). 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, Hazan, and Kale (2012) and Hazan (2016). We can think of solving our problem 5.1 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 to the updates for the other factors. It is in this sense that we are treating problem 5.1 as an online convex optimization problem.

H.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 $\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top\Delta\mathbf{w}_f[t] < 0$:

$$\begin{aligned}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} &= \frac{\partial\mathbf{a}_f}{\partial\mathbf{w}_f}\frac{\partial\mathcal{L}}{\partial\mathbf{a}_f}\\ &= \begin{bmatrix} \dfrac{\Phi_f - w_{f1}}{\Phi_f^2} & \dfrac{-w_{f2}}{\Phi_f^2} & \cdots & \dfrac{-w_{fk}}{\Phi_f^2} \\ \dfrac{-w_{f1}}{\Phi_f^2} & \dfrac{\Phi_f - w_{f2}}{\Phi_f^2} & \cdots & \dfrac{-w_{fk}}{\Phi_f^2} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{-w_{f1}}{\Phi_f^2} & \dfrac{-w_{f2}}{\Phi_f^2} & \cdots & \dfrac{\Phi_f - w_{fk}}{\Phi_f^2} \end{bmatrix}\nabla_{\mathbf{a}_f}\mathcal{L}.\end{aligned}$$

$$= \left(\frac{1}{\Phi_f}\mathbf{I} - \frac{1}{\Phi_f^2}\mathbf{1}\mathbf{w}^\top\right)\nabla_{\mathbf{a}_f}\mathcal{L}$$

$$= \frac{1}{\Phi_f}\nabla_{\mathbf{a}_f}\mathcal{L} - \frac{\mathcal{L}(\mathbf{a}_f)}{\Phi_f}\mathbf{1}. \tag{H.1}$$

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

$$\begin{aligned}\Delta\mathbf{w}_f[t] &= -\frac{1}{\rho}\mathbf{w}_f[t] \odot \nabla_{\mathbf{a}_f}\mathcal{L} = -\frac{1}{\rho}\mathbf{w}_f[t] \odot \left(\Phi_f \nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} + \mathcal{L}(\mathbf{a}_f[t])\mathbf{1}\right)\\ &= -\frac{\Phi_f}{\rho}\mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\rho}\mathbf{w}_f[t]. \end{aligned} \tag{H.2}$$

And then we can easily show the desired result:

$$\begin{aligned}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top\Delta\mathbf{w}_f[t] &= -\frac{\Phi_f}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top\mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}} - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top\mathbf{w}_f[t]\\ &= -\frac{\Phi_f}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top\mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}\\ &\quad - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\rho}\left(\frac{1}{\Phi_f}\nabla_{\mathbf{a}_f}\mathcal{L}^\top - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\Phi_f}\mathbf{1}^\top\right)\mathbf{w}_f[t]\\ &= -\frac{\Phi_f}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top\mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}\\ &\quad - \frac{\mathcal{L}(\mathbf{a}_f[t])}{\rho}\Big(\mathcal{L}(\mathbf{a}_f[t]) - \mathcal{L}(\mathbf{a}_f[t])\Big)\\ &= -\frac{\Phi_f}{\rho}\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}^\top\mathrm{diag}(\mathbf{w}_f[t])\nabla_{\mathbf{w}_f}\tilde{\mathcal{L}}\\ &< 0. \end{aligned} \tag{H.3}$$

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 section E.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 step size 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 section 6 used $\eta = 0.3$ and $\epsilon = 10^{-5}$.

Appendix I: 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, e.g., Arathorn, 2001, 2002; Gedeon & Arathorn, 2007; 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 MSC 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 MSC 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 MSC). The dynamics of MSC are given in equation C.5, 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 step size $\eta = 0.1$. Gedeon and Arathorn (2007) and Harker et al. (2007) proved (with some minor technicalities we will not detail here) that MSC 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 $\nabla_{\mathbf{a}_f}\mathcal{L}$ that are negative, the one with the largest magnitude corresponds to a component of $\mathbf{a}_f$ that sees an update of 0. All other components are decreased by an amount proportional to the gradient. We noted in comments on equation E.1 that the smallest element of $\nabla_{\mathbf{a}_f}\mathcal{L}$ corresponds to the codevector that 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 MSC thus preserve the weight of the codevector that matches best and decrease the weight of the other codevectors by an amount 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 (2002) and is a failure mode of MSC.

Appendix J: 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 is always a 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 bit flip probability for the fth factor of a resonator network is denoted r_f and defined in equation 6.2. Section J.1 derives r_1, which is equal to the bit flip probability for a Hopfield network, introduced by equation 6.1 in the main text. Section J.2 derives r_2, and then section J.3 collects all of the ingredients to express the general r_f.

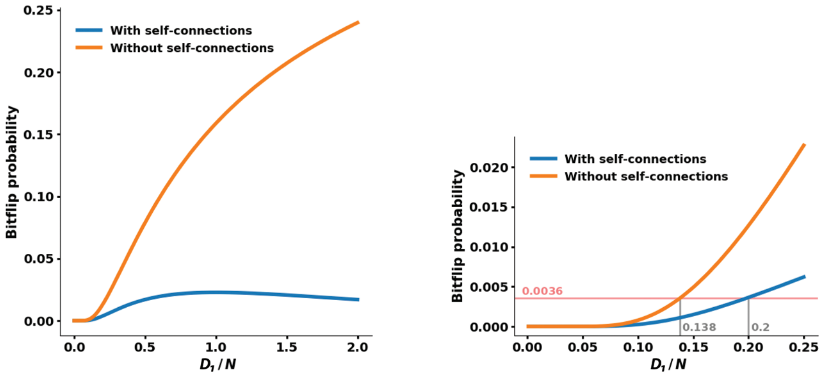
J.1 First Factor.

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:

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



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

Figure 13: Effect of self-connections on bit flip 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:

$$\begin{aligned} h_1 &:= Pr\big[\, \big(\hat{\mathbf{x}}^{(1)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(1)}\big)_i \,\big] \\ &= \Phi\left(\frac{-N - D_1 + 1}{\sqrt{(N-1)(D_1-1)}} \right). \end{aligned} \tag{J.2}$$

We care about the ratio $D_1 \,/\, N$ and how the bit flip probability h_1 scales with this number. We've called this h_1 to denote the Hopfield bit flip probability, but it is also r_1, the bit flip 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 6.1. 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 equation J.1 is eliminated and the bit flip probability is $\Phi\big(\frac{-N}{\sqrt{(N-1)(D_1-1)}}\big)$. This is actually significantly different from equation J.2, which we can see in Figure 13. With self-connections, the bit flip probability is maximized when $D_1 = N$ (readers can verify this via simple algebra), and its maximum value is approximately 0.023. Without self-connections, the bit flip 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 13b. A mean-field analysis of Hopfield networks developed by Amit et al. showed that when $D_1\,/\,N > 0.138$, a phase-change phenomenon occurs in which a small number of initial bit flips (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 bit flip 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 6.2) 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.

J.2 Second Factor. When we update the second factor, we have

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

Here we're just repurposing the notation $\boldsymbol{\Gamma}$ to indicate the vector that 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 number of the components (a small number, we hope) will have been flipped compared to $\mathbf{x}_\star^{(2)}$ by the update to factor 1. Let us denote the set of components that 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 equation J.1.

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

$$
\begin{aligned}
&= |\mathbb{Q}^c|\big(\mathbf{x}_\star^{(2)}\big)_i + \sum_{m\neq\star}^{D_2}\sum_{j\in\mathbb{Q}^c}^{N}\big(\mathbf{x}_m^{(2)}\big)_i\big(\mathbf{x}_m^{(2)}\big)_j\big(\mathbf{x}_\star^{(2)}\big)_j - |\mathbb{Q}|\big(\mathbf{x}_\star^{(2)}\big)_i \\
&\quad - \sum_{m\neq\star}^{D_2}\sum_{j\in\mathbb{Q}}^{N}\big(\mathbf{x}_m^{(2)}\big)_i\big(\mathbf{x}_m^{(2)}\big)_j\big(\mathbf{x}_\star^{(2)}\big)_j \\
&= (N-2|\mathbb{Q}|)\big(\mathbf{x}_\star^{(2)}\big)_i + \sum_{m\neq\star}^{D_2}\sum_{j\in\mathbb{Q}^c}^{N}\big(\mathbf{x}_m^{(2)}\big)_i\big(\mathbf{x}_m^{(2)}\big)_j\big(\mathbf{x}_\star^{(2)}\big)_j \\
&\quad - \sum_{m\neq\star}^{D_2}\sum_{j\in\mathbb{Q}}^{N}\big(\mathbf{x}_m^{(2)}\big)_i\big(\mathbf{x}_m^{(2)}\big)_j\big(\mathbf{x}_\star^{(2)}\big)_j. \qquad \text{(J.3)}
\end{aligned}
$$

If i is in the set of bits that did not flip previously, then there is a constant $(D_2-1)\big(\mathbf{x}_\star^{(2)}\big)_i$ that comes out of the second term above. If i is in the set of bits that did flip previously, then there is a constant $-(D_2-1)\big(\mathbf{x}_\star^{(2)}\big)_i$ that 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, analogous to what we had in equation J.1. Technically $|\mathbb{Q}|$ is a random variable, but when N is of any moderate size, it will be close to $r_1 N$, the bit flip 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 bit flip probabilities that result from these two cases $r_{2'}$ and $r_{2''}$:

$$
\begin{aligned}
r_{2'} &:= Pr\big[\,\big(\hat{\mathbf{x}}^{(2)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(2)}\big)_i \;\big|\; \big(\hat{\mathbf{o}}^{(2)}[1]\odot\mathbf{c}\big)_i = \big(\mathbf{x}_\star^{(2)}\big)_i\,\big] \\
&= \Phi\left(\frac{-N(1-2r_1)-(D_2-1)}{\sqrt{(N-1)(D_2-1)}}\right), \qquad \text{(J.4)} \\
r_{2''} &:= Pr\big[\,\big(\hat{\mathbf{x}}^{(2)}[1]\big)_i \neq \big(\mathbf{x}_\star^{(2)}\big)_i \;\big|\; \big(\hat{\mathbf{o}}^{(2)}[1]\odot\mathbf{c}\big)_i \neq \big(\mathbf{x}_\star^{(2)}\big)_i\,\big] \\
&= \Phi\left(\frac{-N(1-2r_1)+(D_2-1)}{\sqrt{(N-1)(D_2-1)}}\right). \qquad \text{(J.5)}
\end{aligned}
$$

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

J.3 All Other Factors. It should be clear that the general development above for the bit flip probability of the second factor will apply to all subsequent factors; we just need to make one modification to notation. We saw that bit flip probability was different depending on whether the component had flipped in the previous factor (the difference between equations J.4 and J.5). In the general case, what really matters is whether the factor sees a *net*

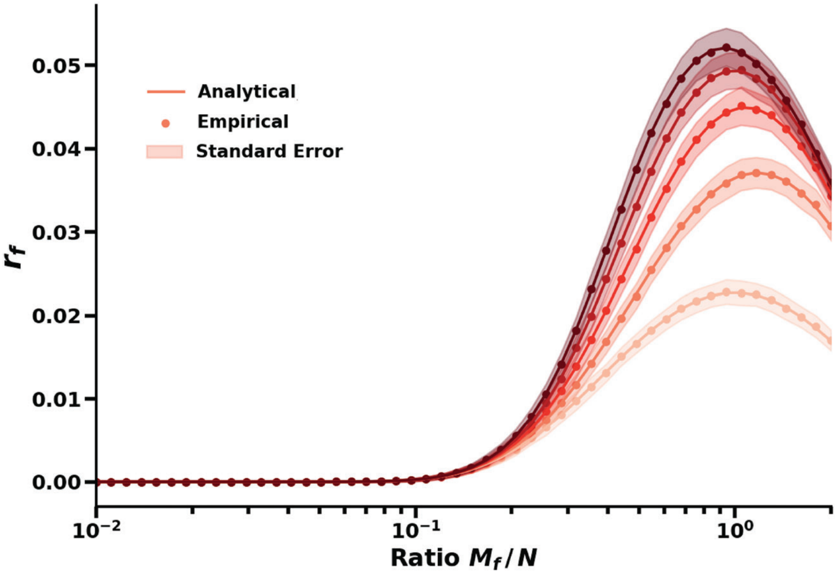


Figure 14: Agreement between simulation and theory for r_f. Shades indicate factors 1 to 5 (light to dark).

bit flip 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 bit flip probability that is passed on to the next factor (this is equation 6.3):

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

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 6.6 in the main text. This expression is just marginalizing over the probability that a net bit flip was not seen (first term) and the probability that a net bit flip was seen (second term). The expression for the general n_f is slightly different:

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

which is equation 6.7. The base of the recursion is $n_0 = 0$, which makes intuitive sense because factor 1 sees no percolated noise.

In equations J.4 and J.5, 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 bit flip probabilities $r_{f'}$ and $r_{f''}$, equations 6.8 and 6.9:

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

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

What we have derived here in appendix J are equations 6.1 to 6.9. This result agrees very well with data generated in experiments where one actually counts the bit flips in a randomly instantiated resonator network. In Figure 14 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 1 standard deviation about the mean, the standard error of this sampling distribution. We generated this plot with 250 i.i.d. 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.

Acknowledgments

We thank members of the Redwood Center for Theoretical Neuroscience for helpful discussions, in particular Pentti Kanerva, whose work on Vector Symbolic Architectures originally motivated this project. This work was generously supported by the National Science Foundation under graduate research fellowship DGE1752814 and research grant IIS1718991, the National Institute of Health, grant 1R01EB026955-01, the Semiconductor Research Corporation under E2CDA-NRI, DARPA's Virtual Intelligence Processing program, and AFOSR FA9550-19-1-0241.

References

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

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.

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

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

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

Arora, S., Hazan, E., & Kale, S. (2012). The multiplicative weights update method: A meta-algorithm and applications. *Theory of Computing*, *8*(1), 12–164.

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

Boucheron, S., Lugosi, G., & Massart, P. (2013). *Concentration inequalities*. Oxford: Oxford University Press.

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

Cannon, L. E. (1969). *A cellular computer to implement the Kalman filter algorithm*. PhD diss. Montana State University–Bozeman.

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.

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

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 ℓ_1 ball. *Mathematical Programming*, *158*(1–2), 575–585.

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*, *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, . . . R_N) approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, *21*(4), 1324–1342.

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* (pp. 272–279). New York: ACM.

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

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.

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

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

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

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. (UCLA Working Papers in Phonetics, 16, 1–84.) University of California, Los Angeles.

Hazan, E. (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: Wiley.

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*, vol. 151 (pp. 212–218). College Park, MD: American Institute of Physics.

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

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.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. In *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. In *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*, 1457–1469.

Huang, J., & Hagiwara, M. (1999). A new multidimensional associative memory based on distributed representation and its applications. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics* (vol. 1, pp. 194–199). Piscataway, NJ: IEEE.

Kanerva, P. (1996). Binary spatter-coding of ordered k-tuples. In *Proceedings of the International Conference on Artificial Neural Networks* (pp. 869–873). Berlin: Springer.

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.

Lee, D. D., & Seung, H. S. (2001). Algorithms for non-negative matrix factorization. In T. K. Leen, T. G. Dietterich, & V. Tresp (Eds.), *Advances in neural information processing systems*, *13* (pp. 556–562). Cambridge, MA: MIT Press.

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

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

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

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.

Sidiropoulos, N. D., & Bro, R. (2000). On the uniqueness of multilinear decomposition of n-way arrays. *Journal of Chemometrics*, *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.

Spall, J. C. (2005). *Introduction to stochastic search and optimization: Estimation, simulation, and control*. New York: Wiley.

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

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: University of Wisconsin Press.

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

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 *Proceedings of the European Conference on Computer Vision* (pp. 447–460). Berlin: Springer.

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

Received July 30, 2019; accepted March 4, 2020.