Kanerva, P. (in press). “Hyperdimensional Computing: An algebra for computing with vectors”; in A. Chen (ed.), *Advances in Semiconductor Technologies*; ISBN: 9781119869580; Wiley, 2022.

April 10, 2022 manuscript

Hyperdimensional Computing:
An Algebra for Computing with Vectors

Pentti Kanerva

Redwood Center for Theoretical Neuroscience
University of California at Berkeley, Berkeley, CA 94720-3198, USA
pkanerva@berkeley.edu

Abstract: Hyperdimensional computing extends the traditional (von Neumann) model of computing with numbers, to computing with wide vectors, e.g., 10,000-bit. Operations that correspond to the addition and multiplication of numbers, augmented by permutations of vector coordinates, allow us to build computers for tasks that are served poorly by today's computers. The hardware requirements are a unique match to 3D nanoscale circuit technology. The vector operations can be built into circuits and programmed in traditional ways. The encoding of information, however, is totally different and takes getting used to. Multiple items of information are encoded into a single vector and distributed over all vector components in a kind of holographic representation. Computing with holographically encoded vectors depends on the superabundance of approximately orthogonal vectors and on the properties of the operations, namely, that some are invertible, some distributive, and some distance-preserving. Such properties are familiar to us from computing with numbers; now they form a foundation for computing with vectors. The goal of computing with vectors is to interpret and to act fast on rich sensory input. Sensory data and commands to actuators are coordinated in the high-dimensional vector space, but raw sensory input must first be brought into the space. It is done with sensor-specific pre-processors that can be designed by experts or trained as traditional neural nets and then frozen. Similarly, high-dimensional vectors for actions are converted to commands for motors by actuator-specific post-processors.

Keywords: Von Neumann architecture; Computing in superposition; Holographic reduced representation; Vector-symbolic architecture; Semantic pointer architecture

0.1 Introduction

The digital computer, enabled by semiconductor technology, has become an ever-present part of our lives. The relative ease of programming it for well-specified tasks has us expecting that we should be able to program it for about any task. However, experience has shown otherwise. What comes naturally to us and may appear easy, such as understanding a scene or learning a language, has eluded programming into computers [Mitchell, 2019]. Since such things are accomplished by brains, we look for computing architectures that work somewhat like brains. Computing of that kind would have wide-ranging uses.

This chapter describes computing with vectors that is modeled after traditional (von Neumann) computing with numbers. Its origins are in the artificial neural systems of the 1970s and '80s that have evolved into today's deep-learning nets, but it is fundamentally different from them.

Similarity to traditional computing relates to how computing is organized. Traditionally there is an arithmetic/logic unit (ALU) with circuits for number arithmetic and Boolean logic, a large random-access memory for storing numbers, and flow control for running a program one step at a time. The same logical organization now refers to vectors as basic objects. The mathematics of the vector operations is the main topic of this chapter.

Why compute with vectors; don't we already have vector processors for tasks heavy with vector operations? The reason has to do with the nature of the vectors and operations on them. The new algorithms rely on truly high dimensionality—a thousand or more—but the vector components need not be precise nor ultrareliable. In contrast, the algorithms for vector processors assume high precision and reliability, the engineering requirements of which are very different. Being able to compute with less-than-perfect circuits makes it possible to fully benefit from the miniaturization of circuits and the inclusion of analog components in them. We will be able to build ever larger circuits that operate with very little energy, eventually approaching the efficiency of the brain which does amazing things with a mere 20

Watts.

Attempts to understand brains in computing terms go back at least to the birth of the digital computer. Early AI consisted of rule-based manipulation of symbols, which mirrors the logic of programming and appeals to our facility for language. However, it fails to explain learning, particularly of language. Artificial neural nets have tried to fill the void by focusing on learning from examples, but has insufficient means for representing and manipulating structure such as the grammar of a language. It is becoming clear that we need systems capable of both statistical learning and computing with discrete symbols. Computing with high-dimensional vectors is aimed at developing systems of that kind.

The first system of the kind was described by Plate in a PhD thesis in 1994, later published in the book *Holographic Reduced Representation* [Plate, 1994, 2003]. It combines ideas from Hinton's [1990] reduced representation, Smolensky's [1990] tensor-product variable binding, and Murdock's [1982] convolution-based memory. Systems that compute with high-dimensional random vectors go by various names: Holographic Reduced Representation, Binary Spatter Code [Kanerva, 1996], MAP (for Multiply–Add–Permute) [Gayler, 1998], Context-Dependent Thinning [Rachkovskij & Kussul, 2001], Vector-Symbolic Architectures (VSA) [Gayler, 2003; Levy & Gayler, 2008], Hyperdimensional Computing [Kanerva, 2009], Semantic Pointer Architecture [Eliasmith, 2013], and Matrix Binding of Additive Terms [Gallant & Okaywe, 2013].

0.2 Overview: Three Examples

Computing is based on three operations. Two correspond to *addition* and *multiplication* of numbers and are called by the same names, and the third is *permutation* of coordinates. All three take vectors as input and produce vectors of the *same dimensionality* as output. The operations are programmed into algorithms as in traditional computing.

The representation of information is very different from what we are used to. In traditional computing, variables are represented by locations in memory and values by the bit patterns in those locations. In computing with vectors, both the variables and the values are vectors of a common high-dimensional space, and variable x having value a is also a vector in that space. Moreover, any piece of information encoded into a vector is *distributed uniformly* over the entire vector, in what is called holographic or holistic representation. We will demonstrate these ideas with three example: (1) variable x having value a, (2) a data record for three variables, and (3) sequence-learning. The dimensionality of the vectors will be denoted by H ($H = 10{,}000$ for example) and the vectors are also called *HD vectors* or *hypervectors*. Variables and values are represented by random H-dimensional vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{a}, \mathbf{b}, \mathbf{c}$ of equally probable 1s and -1s, called *bipolar*.

0.2.1 Binding and Releasing with Multiplication

Variable x having value a is encoded with multiplication, which is done coordinatewise and denoted with $*$: $\mathbf{p} = \mathbf{x} * \mathbf{a}$, where $p_h = x_h a_h,\ h = 1, 2, \ldots, H$. We can also find the vector $\mathbf{a}$ that is encoded in $\mathbf{p}$ by multiplying $\mathbf{p}$ with the *inverse* of $\mathbf{x}$. The inverse of a bipolar vector is the vector itself, and thus we have that

$$\mathbf{x} * \mathbf{p} = \mathbf{x} * (\mathbf{x} * \mathbf{a}) = (\mathbf{x} * \mathbf{x}) * \mathbf{a} = \mathbf{1} * \mathbf{a} = \mathbf{a}$$

where $\mathbf{1}$ is the vector of H 1s. Combining a variable and its value in a single vector is called *binding*, and decoding the value as “unbinding” or *releasing*.

0.2.2 Superposing with Addition

Combining the values of several variables in a single vector begins with binding each variable–value pair as above. The vectors for the bound pairs are *superposed* by adding them into a single vector $\mathbf{r} = (\mathbf{x} * \mathbf{a}) + (\mathbf{y} * \mathbf{b}) + (\mathbf{z} * \mathbf{c})$. However, the sum is a vector of integers $\{-3, -1, 1, 3\}$. To make it bipolar, we take the sign of each component, to get $\mathbf{s} = \mathrm{sign}(\mathbf{r})$. Ties are a problem when the number of vectors in the sum is even; we will discuss that later.

Can we decode the values of the variables in the composed vector **s**? We can, if the vectors for the variables are orthogonal to each other or approximately orthogonal. For example, to find the value of x we multiply **s** with (the inverse of) **x** as above, giving

$$\begin{aligned}
\mathbf{x} * \mathbf{s} &= \mathbf{x} * (\text{sign}((\mathbf{x} * \mathbf{a}) + (\mathbf{y} * \mathbf{b}) + (\mathbf{z} * \mathbf{c})) \\
&= \text{sign}(\mathbf{x} * ((\mathbf{x} * \mathbf{a}) + (\mathbf{y} * \mathbf{b}) + (\mathbf{z} * \mathbf{c}))) \\
&= \text{sign}((\mathbf{x} * \mathbf{x} * \mathbf{a}) + (\mathbf{x} * \mathbf{y} * \mathbf{b}) + (\mathbf{x} * \mathbf{z} * \mathbf{c})) \\
&= \text{sign}(\mathbf{a} + \text{noise} + \text{noise}) \\
&\approx \text{sign}(\mathbf{a}) \\
&= \mathbf{a}
\end{aligned}$$

The result is approximate but close enough to **a** to be identified as **a**.

The example relies on two properties of high-dimensional representation: approximate orthogonality and noise-tolerance. Pairs of random vectors are approximately orthogonal and so the vectors for the variables can be chosen at random, and the vectors $\mathbf{x} * \mathbf{y} * \mathbf{b}$ and $\mathbf{x} * \mathbf{z} * \mathbf{c}$ are meaningless and act as random noise.

The example demonstrates *distributivity* of multiplication over addition and the need for an *associative memory* that stores all known vectors and outputs vectors that match the input the best. The memory is also called *item memory* and *clean-up memory.* Figure 1 shows the encoding and decoding of a data record for three variables step by step.

0.2.3 Sequences with Permutation

The third operation is *permutation*; it reorders vector coordinates. Permutation is shown as $\rho(\mathbf{x})$ and its inverse as $\rho^{-1}(\mathbf{x})$, or simply as $\rho\mathbf{x}$ and $\rho^{-1}\mathbf{x}$. Permutations are useful for encoding sequences as seen in the following example on language identification. We look at text, letter by letter, without resorting to dictionaries—the identification is based on letter-use statistics peculiar to each language [Joshi, Halseth & Kanerva, 2017].

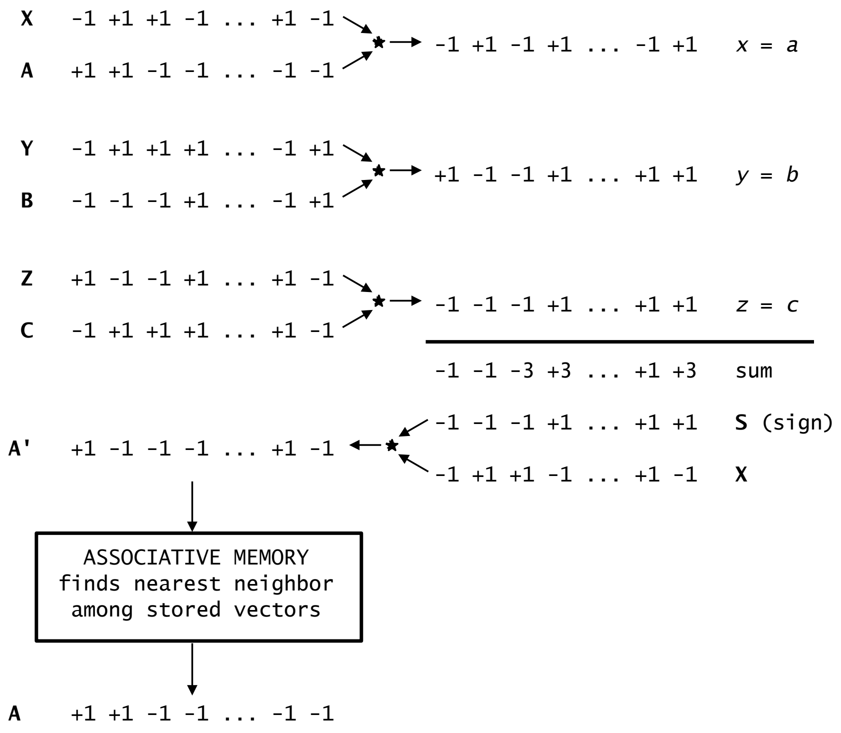

Figure 1: Encoding $\{x = a,\ y = b,\ z = c\}$ as **S** and releasing **A** with (the inverse of) **X**. * denotes coordinatewise multiplication.

For each language we compute a high-dimensional *profile vector* or *prototype* from about a million bytes of text. We use the same algorithm to compute a profile vector for a test sentence, which is then compared to the language profiles and the most similar one is chosen as the system's answer.

The profiles are based on three-letter sequences called *trigrams* and they are computed as follows. The 26 letters and the space are assigned H-dimensional random bipolar seed vectors like the ones for the variables above. The same *letter vectors* are used with all languages and test sentences. The letter vectors are used to make *trigram vectors* with permutation and multiplication. For example, the vector for the trigram *the* is computed by permuting the *t*-vector twice, permuting the *h*-vector once, taking the *e*-

vector as is, and multiplying the three componentwise: $\rho(\rho(\mathbf{t})) * \rho(\mathbf{h}) * \mathbf{e}$. This produces an H-dimensional trigram vectors of randomly placed ± 1s. Finally the trigram vectors are added together into a *profile vector* by stepping trough the text one trigram at a time. The result is an H-dimensional vector of integers. The cosine of profile vectors is used to measure their similarity.

Such an experiment with 21 European Union languages gave the following results [Joshi, Halseth & Kanerva, 2017]. All vectors were 10,000-dimensional. The language profiles clustered according to language families: Baltic, Germanic, Romance, Slavic. When test-sentence profiles were compared to the language profiles, the correct language was chosen 97% of the time, and when the language profile for English was queried for the letter most often following *th*, the answer was *e*.

The three examples illustrate computing with vectors, superposing them, and learning from data. Next we go over the operations in detail. Much of what traditional neural networks do—and fail to do—can be analyzed and understood in terms of the three vector operations, and of vector similarity and associative memory.

0.3 Operations on Vectors

We will continue with *bipolar* vectors because of the ease of working with them: components with mean = 0 and variance = 1 make it easy. However, the basic idea is the same when computing with high-dimensional random binary, real or complex vectors. Thus we are dealing with *general* properties of high-dimensional representation. Computing with vectors is just as natural and equally justified as computing with numbers.

- The bipolar **space of representations** rangers over H-dimensional vectors of 1s and -1s, with H a thousand or more: $\mathbf{a}, \mathbf{b}, \mathbf{c}, \ldots \in \{1, -1\}^H$. On occasion we consider also vectors of integers, $\mathbb{Z}^H$.

- **Associative memory** is the "RAM" for high-dimensional vectors.

It stores the vectors known to the system and recognizes or retrieves them from their noisy versions in what is called "clean up." The other use is that of an ordinary RAM: given an address, store or retrieve the vector associated with that exact address or one most similar to it. We can also think of it as a memory for key–value pairs where the keys can be noisy. The actual making of such a memory will be discussed below.

- The **similarity** ($\sim$) of vectors $\mathbf{a}$ and $\mathbf{b}$ implies a distance between them and is computed via their dot product $\mathbf{a} \cdot \mathbf{b}$. For H-dimensional bipolar vectors it varies from H when the vectors are the same, to $-H$ when they are opposites. Similarity is expressed conveniently with the cosine, given by $\cos(\mathbf{a}, \mathbf{b}) = (\mathbf{a} \cdot \mathbf{b})/H$ for the bipolar. Dot product or cosine $=$ 0 means that the vectors are orthogonal, i.e., unrelated, uncorrelated, dissimilar: $\mathbf{a} \not\sim \mathbf{b}$. Computing with vectors tries to capture similarity of meaning in the similarity of vectors.

The distribution of distances between high-dimensional vectors is remarkable. Given any vector, nearly all others are approximately orthogonal to it [Widdows & Cohen, 2015]. This is called *concentration of measure* and it means that large collections of random vectors—billions when $H = 10{,}000$—include no similar pairs. The easy availability of approximately orthogonal vectors is paramount to computing with vectors.

The somewhat imprecise terms "approximately equal" ($\approx$), "similar" ($\sim$), "dissimilar" ($\not\sim$) and "approximately orthogonal" need clarification. Approximately equal vectors are like noisy copies of each other and have the same meaning, similar vectors arise from constructions with common constituents (see Addition), dissimilar is used for vectors that are orthogonal or approximately orthogonal. Each bipolar vector has an opposite vector, but the two are not considered representing opposite meanings.

- **Addition** is coordinatewise vector addition and it *commutes.* The sum is a vector of integers in $\mathbb{Z}^H$ and is *normalized* by the sign function, with 0s mapped to 1s and -1s at random. Normalizing the sum is shown with brackets: $[-2,-6,0,0,2,4,\dots] = (-1,-1,1,-1,1,1,\dots)$ for example.

  The sum is *similar* to its inputs: $[\mathbf{a}+\mathbf{b}] \sim \mathbf{a},\mathbf{b}$. For example, $\cos(\mathbf{a}, [\mathbf{a}+\mathbf{b}]) = 0.5$ for random $\mathbf{a}$ and $\mathbf{b}$. Addition is *associative* before the sum is normalized but only approximately associative after:

  $$[[\mathbf{a}+\mathbf{b}]+\mathbf{c}] \sim [\mathbf{a}+[\mathbf{b}+\mathbf{c}]]$$

  Likewise, it is *invertible* before normalization but only approximately invertible after: $[[\mathbf{a}+\mathbf{b}]+(-\mathbf{b})] \sim \mathbf{a}$. Information is lost each time a sum is normalized and so normalizing should be delayed whenever possible. Addition is also called *bundling* and *superposing.*

- **Multiplication** is done coordinatewise, known as Hadamard product, and it *commutes.* The product $\mathbf{a} * \mathbf{b}$ of bipolar vectors is also bipolar and thus ready for use as input in subsequent operations. Multiplication is *invertible*—a bipolar vector is its own inverse—it *distributes* over addition: $\mathbf{x} * [\mathbf{a}+\mathbf{b}] = [(\mathbf{x}*\mathbf{a}) + (\mathbf{x}*\mathbf{b})]$; and it *preserves similarity*: $(\mathbf{x}*\mathbf{a}) \cdot (\mathbf{x}*\mathbf{b}) = \mathbf{a}\cdot\mathbf{b}$, which also means that a vector can be moved across the dot: $\mathbf{a}\cdot(\mathbf{b}*\mathbf{c}) = (\mathbf{a}*\mathbf{b})\cdot\mathbf{c}$. The product is *dissimilar* to its inputs: $\mathbf{a}*\mathbf{b} \not\sim \mathbf{a},\mathbf{b}$. See Figure 1 for examples of addition and multiplication.

- **Permutations** reorder vector coordinates. The number of permutations is enormous, $H!$ overall. Permutations are *invertible*: $\rho^{-1}(\rho(\mathbf{a})) = \mathbf{a}$; they *distribute* over both addition and multiplication: $\rho[\mathbf{a}+\mathbf{b}] = [\rho(\mathbf{a}) + \rho(\mathbf{b})]$ and $\rho(\mathbf{a}*\mathbf{b}) = \rho(\mathbf{a}) * \rho(\mathbf{b})$, in fact, permutations distribute over all coordinatewise operations, such as Booleans; they *preserve similarity*: $\rho(\mathbf{a})\cdot\rho(\mathbf{b}) = \mathbf{a}\cdot\mathbf{b}$; but most permutations do *not commute*: $\rho(\sigma(\mathbf{a})) \neq \sigma(\rho(\mathbf{a}))$. The output of a random permutation is *dissimilar* to the input: $\rho(\mathbf{a}) \not\sim \mathbf{a}$.

Table 1: Summary of Bipolar Vector Operations

Property	Dot product $\mathbf{a} \cdot \mathbf{b}$	Sum $\mathbf{a} + \mathbf{b}$	Normal'd sum $[\mathbf{a} + \mathbf{b}]$	Product $\mathbf{a} * \mathbf{b}$	Permu-tation $\rho(\mathbf{a})$
Associative	n/a	Yes	approx*	Yes	Yes
Commutative	Yes	Yes	Yes	Yes	No
Invertible	n/a	Yes	approx*	Yes	Yes
Similar to inputs, increases similarity	n/a	Yes	Yes	No	No
Preserves similarity, randomizes	n/a	No	No	Yes	Yes
Distributes over addition	Yes	n/a	n/a	Yes	Yes
Distributes over multiplication	n/a**	No	No	n/a	Yes

*Partly true

**However, $\mathbf{a} \cdot (\mathbf{b} * \mathbf{c}) = (\mathbf{a} * \mathbf{b}) \cdot \mathbf{c} = (\mathbf{a} * \mathbf{b} * \mathbf{c}) \cdot \mathbf{1}$

Permutations themselves are not elements of the vector space. In linear algebra they are represented by matrices and so ρ can be thought of as a permutation matrix; here they are *unary* operations on vectors. They are potentially very useful by incorporating all finite groups up to size H into the vector math. The permutation $\sigma(\rho(\mathbf{a}))$ is commonly abbreviated to $\sigma\rho\mathbf{a}$ and it equals $(\sigma\rho)\mathbf{a}$.

The operations and their properties for bipolar vectors are summarized in Table 1, as an example of things to consider when setting up a system of computing with vectors. A system for *binary* vectors is equivalent to the bipolar when 1 is replaced by 0, -1 by 1, multiplication by XOR, and the sign function by coordinatewise majority, and when similarity is based on the Hamming distance.

0.4 Data Structures

Computer programming consists of laying and tracing pathways to data, and then doing arithmetic and logic operations on the data. The pathways are called data structures and they include sets, sequences, lists, queues, stacks, arrays, graphs, heaps, and so forth. The data are the values attached to the structure, but to the computer the structure itself is also data. It is impossible to draw a sharp boundary between data and structure, ever more so when computing in holographic representation.

Since data structures are an essential part of programming and computing, we need to look at how to encode and operate with them in superposed vectors.

- **Seed Vectors.** Computing begins with the selection of vectors for basic entities such as variables and values. Bipolar seed vectors are made of random, independent, equally probable 1s and -1s. They are also called *atomic vectors* and *elemental vectors* because they are the stuff from which everything else is built. For example, in working with text, each letter of the alphabet can be represented by a seed vector. A set of seed vectors is called *alphabet* or *vocabulary* or *codebook*.

  Because of high dimensionality, randomly chosen (seed) vectors are *approximately orthogonal*—the superabundance of approximately orthogonal vectors and the relative ease of making them is a primary reasons for high dimensionality. Orthogonality allows multiple vectors to be encoded into a single vector and subsequently decoded, making it possible to analyze and interpret the results of computing in superposition. The selection of a seed vector corresponds to assigning a memory location—an address—to a variable, or choosing a representation for its value.

- **Bound pairs** encode a *variable* and its *value*—or a *role* and a *filler*, or a *key* and a *value*—in a single vector. If binding is done with an invertible operation, the value can be recovered by decoding. An example of binding with multiplication is shown in the Introduction where $x = a$ is encoded with $\mathbf{x} * \mathbf{a}$.

- **Sets** and **multisets** name their members but do not specify their order: $\{a, a, b, c\} = \{a, b, a, c\}$. They can be encoded with addition because it commutes. Since the sum is similar to the vectors in it, it is possible to query whether a specific vector is included in the set or the multiset. That works reliably for small sets, but the adding of vectors makes the sum less similar to any one of them, and normalizing the sum makes it even less similar.

  To decode a sum, we look for vectors similar to it in the associative memory. Once a vector is assumed to be in a sum, it can be subtracted out and the remaining sum queried for further vectors. Peeling off vectors one at a time works with sums that have not been normalized, but poorly with normalized sum vectors because of information that has been lost (hence invertibility is called "partly true" in Table 1).

  Multiplication commutes and therefore also it can encode a set, but not a multiset because bipolar vectors are their own inverses and cancel multiple copies of themselves. The product is *dissimilar* to its inputs and can therefore be used as a *label* for a set. Decoding a product is problematic, however. There is no efficient way to do it in general, but if the inputs come from known sets of dissimilar vectors and there are not too many of them, a product can be broken down into its inputs with an iterative search, as discussed below under factorization.

- **Sequences** are ordered multisets, e.g. $(a, a, b, c) \neq (a, b, a, c)$. They can be encoded with permutations. If the application needs only one permutation, rotation of coordinates (cyclic shift) is usually most convenient. The sequence (a, b, c) can be encoded as a sum $\mathbf{s}_3 = \rho^2\mathbf{a} + \rho\mathbf{b} + \mathbf{c}$ or as a product $\mathbf{p}_3 = \rho^2\mathbf{a} * \rho\mathbf{b} * \mathbf{c}$ and extended to include $\mathbf{d}$ by permuting $\mathbf{s}_3$ and adding $\mathbf{d}$: $\mathbf{s}_4 = \rho\mathbf{s}_3 + \mathbf{d} = \rho(\rho^2\mathbf{a} + \rho\mathbf{b} + \mathbf{c}) + \mathbf{d} = \rho^3\mathbf{a} + \rho^2\mathbf{b} + \rho\mathbf{c} + \mathbf{d}$; and similarly $\mathbf{p}_4 = \rho\mathbf{p}_3 * \mathbf{d}$. The successive powers of the permutation act like an *index* into the sequence. Either kind of sequence can be extended recursively with only two operations because permutations distribute over both addition and multiplication.

Decoding the ith vector of the sequence uses the inverse permutation. For example, the first vector in $\mathbf{s}_3$ is found by searching the associative memory for the vector most similar to $\rho^{-2}\mathbf{s}_3$ because that equals $\rho^{-2}(\rho^2\mathbf{a} + \rho\mathbf{b} + \mathbf{c}) = \mathbf{a} + \rho^{-1}\mathbf{b} + \rho^{-2}\mathbf{c} = \mathbf{a} + \text{noise} + \text{noise} = \mathbf{a}' \approx \mathbf{a}$. A sequence encoded with multiplication is harder to decode because products don't resemble their inputs. To decode $\mathbf{a}$ from $\mathbf{p}_3$ requires that $\mathbf{b}$ and $\mathbf{c}$ are already known, in which case $\rho^{-2}(\mathbf{p}_3 * (\rho\mathbf{b} * \mathbf{c})) = \mathbf{a}$ because $\rho\mathbf{b}$ and $\mathbf{c}$ in $\mathbf{p}_3$ cancel out: $\rho^{-2}((\rho^2\mathbf{a} * \rho\mathbf{b} * \mathbf{c}) * (\rho\mathbf{b} * \mathbf{c})) = \rho^{-2}(\rho^2\mathbf{a}) = \mathbf{a}$.

- **Binary trees** can be encoded with two independent random permutations, ρ_1 and ρ_2, that do not commute—most permutations don't. If we encode the pair (a, b) with $\rho_1\mathbf{a} + \rho_2\mathbf{b}$ then the two-deep tree $((a, b), (c, d))$ can be encoded as $\mathbf{t} = \rho_1(\rho_1\mathbf{a} + \rho_2\mathbf{b}) + \rho_2(\rho_1\mathbf{c} + \rho_2\mathbf{d})$, which equals $\rho_1(\rho_1\mathbf{a}) + \rho_1(\rho_2\mathbf{b}) + \rho_2(\rho_1\mathbf{c}) + \rho_2(\rho_2\mathbf{d})$ and can be written as $\rho_{11}\mathbf{a} + \rho_{12}\mathbf{b} + \rho_{21}\mathbf{c} + \rho_{22}\mathbf{d}$, where ρ_{ij} is the permutation $\rho_i\rho_j$. To decode a tree, we follow the indices and apply the inverse permutations in the reverse order. For example,

$$\begin{aligned}
\rho_1^{-1}(\rho_2^{-1}(\mathbf{t})) &= \rho_1^{-1}(\rho_2^{-1}(\rho_1(\rho_1\mathbf{a}) + \rho_1(\rho_2\mathbf{b}) + \rho_2(\rho_1\mathbf{c}) + \rho_2(\rho_2\mathbf{d}))) \\
&= (\rho_1^{-1}\rho_2^{-1}\rho_1\rho_1)\mathbf{a} + (\rho_1^{-1}\rho_2^{-1}\rho_1\rho_2)\mathbf{b} + (\rho_1^{-1}\rho_2^{-1}\rho_2\rho_1)\mathbf{c} \\
&\quad + (\rho_1^{-1}\rho_2^{-1}\rho_2\rho_2)\mathbf{d} \\
&= \text{noise} + \text{noise} + \mathbf{c} + \text{noise} \\
&\approx \mathbf{c}
\end{aligned}$$

where $\rho_2^{-1}(\mathbf{t})$ extracts the right half of the tree and ρ_1^{-1} extracts its left branch. An alternative encoding of binary trees uses one permutation and two seed vectors. Successive powers of the permutation encode depth, and they are multiplied by the two vectors that mean left and right [Frady et al., 2020].

- **Graphs** consist of a set of nodes and connecting links. Of the different kind, we consider directed graphs with loops, i.e., where nodes can have links to themselves. Graphs are used to depict relations between entities, for example, 'parent of', 'hears from' and 'communicates with'. The nodes can also represent the set of states $s_i \in S$ of a Markov chain or a finite-state automaton, and the links $t_{ij} \in T$ its state transitions, $T \in S \times S$. We will encode the states by random seed vectors $\mathbf{s}_i$—they name or label the states. The transition $t_{ij} = (s_i, s_j)$ can then be encoded with permutation and multiplication as $\mathbf{t}_{ij} = \rho\mathbf{s}_i * \mathbf{s}_j$, and the graph with the sum of all its transitions:

$$\mathbf{g} = \sum_{t_{ij} \in T} \mathbf{t}_{ij} = \sum_{t_{ij} \in T} \rho\mathbf{s}_i * \mathbf{s}_j.$$

Given the vector $\mathbf{g}$ for the graph, we can ask whether the state s_j can be reached from the state s_i in a single step. The answer is contained in the vector $\mathbf{g}_i = \rho\mathbf{s}_i * \mathbf{g}$ and it is found by comparing $\mathbf{g}_i$ to $\mathbf{s}_j$: it is 'yes' if the two are similar and 'no' if they are dissimilar. This follows from the distributivity of multiplication over addition, and from the sum being similar to its inputs: the vector $\mathbf{g}$ is a sum of transitions, and multiplying it by $\rho\mathbf{s}_i$ releases $\mathbf{s}_j$ if $\rho\mathbf{s}_i * \mathbf{s}_j$ is included in the sum. Notice that $\mathbf{g}_i$ releases (it brings to the surface) *all* the states (their labels) that can be reached from s_i in a single step; $\mathbf{g}_i$ also includes a noise vector for every transition from states other than s_i and so $\mathbf{g}_i$ is a noisy representation of the *set* of states one step from s_i.

We can go further and look for the (multi)set of states two steps from s_i. The answer is contained in—it's similar to—$\rho(\rho\mathbf{s}_i * \mathbf{g}) * \mathbf{g} = \rho^2\mathbf{s}_i * \rho\mathbf{g} * \mathbf{g}$; or three steps from s_i: $\rho^3\mathbf{s}_i * \rho^2\mathbf{g} * \rho\mathbf{g} * \mathbf{g}$, but it gets noisier at each step, overpowering the signal.

Linear algebra gives us an exact answer in terms of the $|S| \times |S|$ state-transition matrix $\mathbf{T}$ where links are represented by 1s (and non-links by 0s). The set of states reached from s_i in a single step is given by the $|S|$-dimensional vector $\mathbf{iT}$ where $\mathbf{i}$ is an $|S|$-dimensional indicator vector whose ith component equals 1. The (multi)set of states reached in exactly three steps is given by $\mathbf{iT}^3$. We can see that $\rho^2\mathbf{g} * \rho\mathbf{g} * \mathbf{g}$ serves a function similar to $\mathbf{T}^3$, and also that its form is similar to the encoding of a sequence with multiplication, as shown in the discussion of sequences.

Examples of computing with graphs include graph isomorphism [Gayler & Levy, 2009] and a finite automaton [Osipov, Kleyko & Legalov, 2017].

0.5 Vector Sums Encode Probabilities

Probabilities can be included in and inferred from high-dimensional vectors without explicit counting and bookkeeping. This opens the door to statistical learning from data, which is traditionally the domain of artificial neural nets. It is clearly seen in the unnormalized representation of a multiset, i.e., when the vectors of the multiset are simply added into a sum vector $\mathbf{f}$.

The dot product of a bipolar H-dimensional vector $\mathbf{x}$ with itself is H: $\mathbf{x} \cdot \mathbf{x} = H$. If $\mathbf{f}$ is the sum of k copies of $\mathbf{x}$, the dot product $\mathbf{x} \cdot \mathbf{f} = kH$. If other vectors are added to $\mathbf{f}$ and they all are orthogonal to $\mathbf{x}$, the dot product $\mathbf{x} \cdot \mathbf{f}$ still is kH, and it is approximately kH if the other vectors are approximately orthogonal to $\mathbf{x}$. Thus the dot product of a bipolar vector $\mathbf{x}$ with a sum vector $\mathbf{f}$, divided by H, is an estimate of the number of times $\mathbf{x}$ has been added into the sum. This explains the identification of languages from their profile vectors in example 3 of the Overview. It goes as follows [Joshi, Halseth & Kanerva, 2017]:

Each language and each test sentence is represented as a sum of trigrams that have been encoded as sequences of three letter vectors, e.g., $\rho^2\mathbf{t} * \rho\mathbf{h} * \mathbf{e}$. The letter vectors are dissimilar to each other, and since the outputs of permutation and multiplication are dissimilar to their inputs, the trigram vectors are also dissimilar—approximately orthogonal. A profile vector $\mathbf{f}$, which is their sum, can then be used to estimate the frequencies of the trigrams in the text. The trigram statistics for different languages apparently are different enough to allow test sentences to be identified correctly as to language, but similar enough within language families to produce Baltic, Germanic, Romance, and Slavic clusters. Numerically, 27 letters give rise to $27^3 = 19{,}683$ possible trigrams, and so the algorithm projects a histogram of 19,683 trigram frequencies randomly to 10,000 dimensions ($H = 10{,}000$ was used in the example). The experiment was repeated with tetragrams and gave a slightly better result (97.8% vs. 97.3% correct). In that case a histogram of $27^4 = 531{,}441$ possible frequencies is projected randomly to 10,000 dimensions.

Finally, the letter most often following *th* in English is found by multiplying the profile for English, $\mathbf{f}_{\text{English}}$, with (the inverse of) $\rho^2\mathbf{t} * \rho\mathbf{h}$. The multiplication distributes over every trigram vector added into $\mathbf{f}_{\text{English}}$ and cancels out the initial *th* wherever it occurs. In particular, it releases $\mathbf{e}$ from $\rho^2\mathbf{t} * \rho\mathbf{h} * \mathbf{e}$. It also releases every other letter that comes after *th*, but since *e* is the most frequent, $\rho^2\mathbf{t} * \rho\mathbf{h} * \mathbf{f}_{\text{English}}$ has a higher dot product with $\mathbf{e}$ than with any other letter vector. The dot product is the same as between $\mathbf{f}_{\text{English}}$ and the vector for the trigram *the*: $(\rho^2\mathbf{t} * \rho\mathbf{h} * \mathbf{f}_{\text{English}}) \cdot \mathbf{e} = (\rho^2\mathbf{t} * \rho\mathbf{h} * \mathbf{e}) \cdot \mathbf{f}_{\text{English}}$—as vectors of a product move across the dot. Its expected value is the number of times *e* occurs after *th*, multiplied by 10,000.

The language example suggests the possibility of representing a Markov chain as a high-dimensional vector learned from data—in this case a second-order chain. A language profile made of trigrams allows us to estimate letter frequencies following a pair of letters such as *th*. The vector $\mathbf{e}$ was found by searching the 27 letter vectors for the best match to the query $\mathbf{q} = \rho^2\mathbf{t} * \rho\mathbf{h} * \mathbf{f}_{\text{English}}$. If the letter vectors are treated as a $H \times 27$ matrix $\mathbf{A}$ for the alphabet, then multiplying $\mathbf{q}$ with $\mathbf{A}^{\mathrm{T}}$ approximates each latter's relative weight in candidacy for the next letter. However, since the alphabet vectors and the trigram vectors are only approximately orthogonal, the weights are approximate. Yet the transition frequencies have been captured by the model and can govern the probability of choosing the next letter: choose it in proportion to the 27 elements of $\mathbf{A}^{\mathrm{T}}\mathbf{q}$, adjusted for the randomness due to orthogonality being approximate. How actually to choose the next letter based on the probabilities or their estimates, other than by traditional programming, remains an open question.

The language example demonstrates the power of computing with vectors when it is based on a comprehensive arithmetic of vectors. It uses all three operations: a language profile vector is a *sum of products of permutations.* The algorithm for "training" is the *same* as for making profiles of test sentences. It is simple and easily adapted to classification problems at large, and it works in one pass over the data, meaning that the algorithm is *incremental.* Frequencies and probabilities can be recovered approximately from a profile vector by inverting the operations used to encode a profile: the representation is *explainable.*

0.6 Decoding a Product

Unlike a sum vector which is similar to its inputs, the product vector is dissimilar. Thus $\mathbf{x} * \mathbf{y}$ gives us no clue as to its originating from $\mathbf{x}$ and $\mathbf{y}$, nor that $\mathbf{w} = \rho^2\mathbf{t} * \rho\mathbf{h} * \mathbf{e}$ consists of $\mathbf{t}$, $\mathbf{h}$ and $\mathbf{e}$. In fact, any product vector can be factored to possible input vectors in countless ways. However, if we know that the vector $\mathbf{w}$ represents a sequence of three letters, we can examine all 27^3 possible sequences systematically to see which of them yields $\mathbf{w}$. That is up to 19,683 tests. As an alternative to a systematic search, we can search for the answer through successive approximations or educated guesses of $\mathbf{t}$, $\mathbf{h}$ and $\mathbf{e}$ with an algorithm called the *resonator.* We will explain the algorithm by referring to the product $\mathbf{p}$ of three vectors, $\mathbf{p} = \mathbf{x} * \mathbf{y} * \mathbf{z}$, drawn from three different dictionaries or codebooks, $\mathbf{X}, \mathbf{Y}$ and $\mathbf{Z}$—their columns are the codevectors [Frady et al., 2020].

If the product and all its inputs but one are known, the "unknown" input is simply the product of the known vectors, e.g., $\mathbf{x} = \mathbf{p} * \mathbf{y} * \mathbf{z}$. If $\mathbf{y}$ and $\mathbf{z}$ are noisy, however, the "unknown" $\mathbf{x}' = \mathbf{p} * \mathbf{y}' * \mathbf{z}'$ is even more noisy on the average. However, it can be used to compute a new estimate $\mathbf{x}''$ that has a higher probability of being one of the vectors in $\mathbf{X}$. The vector $\mathbf{x}''$ is computed as the weighted sum of the codevectors in $\mathbf{X}$, normalized to bipolar, where the similarity of $\mathbf{x}'$ to the vectors in $\mathbf{X}$ serve as the weights. This can be expressed as $\mathbf{x}'' = [\mathbf{X}(\mathbf{X}^{\mathrm{T}}\mathbf{x}')]$, where $\mathbf{X}^{\mathrm{T}}\mathbf{x}'$ are the weights and $[\ldots]$ makes the result bipolar. If $\mathbf{x}''$ is in the codebook $\mathbf{X}$ we accept it and continue to search for the remaining inputs. If it is not in $\mathbf{X}$, we continue to search for the inputs one codebook at a time by computing $\mathbf{y}'' = [\mathbf{Y}(\mathbf{Y}^{\mathrm{T}}\mathbf{y}')] = [\mathbf{Y}(\mathbf{Y}^{\mathrm{T}}(\mathbf{x}'' * \mathbf{p} * \mathbf{z}'))]$, computing $\mathbf{z}'' = [\mathbf{Z}(\mathbf{Z}^{\mathrm{T}}\mathbf{z}')] = [\mathbf{Z}(\mathbf{Z}^{\mathrm{T}}(\mathbf{x}'' * \mathbf{y}'' * \mathbf{p}))]$, back to computing $\mathbf{x}'''$ from $\mathbf{x}''$ as before, and so on.

We still need to choose vectors $\mathbf{y}'$ and $\mathbf{z}'$ to get started. Recalling that a sum vector is similar to each of its inputs, the normalized sums of the codevectors in $\mathbf{Y}$ and $\mathbf{Z}$ are used. The probability and rate of convergence to the correct vectors depend on the number of inputs in the product, the sizes of the codebooks, and the dimensionality H [Kent et al., 2020].

0.7 High-Dimensional Vectors at Large

The idea of computing with high-dimensional vectors is simplest to convey with the bipolar, and bipolar vectors are also useful in applications. However, the idea is general and depends more on the abundance of nearly orthogonal vectors, than on the nature of the vector components. The abundance comes from high dimensionality. It also matters greatly to have a useful set of operations on the vectors, akin to add, multiply and permute for bipolar vectors. In fact, the corresponding add and multiply of numbers constitute an algebraic field. The vector math adds to it all finite groups up to size H.

We have already commented on the equivalence of the *binary* with the bipolar when coordinatewise multiplication is replaced by bitwise Exclusive-Or (XOR). To convert a binary sum vector to the exact bipolar sum vector, and vice versa, we also need to keep count of the vectors in the sum.

The original Holographic Reduced Representation [Plate, 1994] is based on *real* vectors with random independent normally distributed components with mean = 0 and variance = $1/H$. Addition is vector addition followed by normalization (to Euclidean length 1), and multiplication is by circular convolution; its approximate inverse is called "circular correlation." Similarity of vectors is based on the Euclidean distance, dot product or cosine, all of them being essentially the same when the vectors are normalized to unit length.

Holographic Reduced Representation with *complex* vectors uses random phase angles as vector components. Addition is by vector addition followed by normalization (coordinatewise projection to the unit circle), multiplication is by coordinatewise addition of phase angles (i.e., complex multiplication), and similarity is based on the magnitude of the difference between H-dimensional complex vectors.

All these frameworks are related and their properties are essentially the same. The binary and the bipolar are equivalent, the complex becomes the bipolar when the phase angles are restricted to 0 and 180°, and the real and the complex are related by FFT. The choice of representation can depend on a variety of factors. For example, binary vectors are the simplest to realize in hardware, and complex vectors (phase angles) provide a model for computing with the timing of spikes.

Computing with vectors, as describes in this chapter, assumes *dense* vector: half 1s and half -1s (or half 0s and half 1s for binary). Addition and multiplication automatically tend toward dense vectors, which is mathematically convenient but may not be desirable otherwise and will be commented on below.

Computing can also be based on Boolean operations on bit vectors and on permutations. These operations are common in hashing for distributing data in a high-dimensional space. They are also used in *Context-Dependent Thinning* [Rachkovskij & Kussul, 2001] to encode structure with sparse binary vectors. *Geometric Algebra* offers a further possibility to compute with high-dimensional vectors, called multivectors [Aerts et al., 2009].

The activity of neurons in the brain is *very sparse*, which is partly responsible for the remarkable energy efficiency of brains. Sparse representation is also the most efficient for storing information, so why not compute with sparse vectors? The answer is simply that we have not found operations for sparse vectors that work as well as the combination of add, multiply and permute for dense vectors. This is a worthy challenge for mathematicians to take on.

0.8 Memory for High-Dimensional Vectors

Computing with vectors is premised on the dimensionality H remaining constant. The choice of H can vary over a wide range, however, and the exact value is not critical (e.g., $1{,}000 \leq H \leq 10{,}000$). Mainly, it needs to be large enough to give us a sufficient supply of random, approximately orthogonal vectors. That number grows exponentially with H [Gallant & Okaywe, 2013].

Whatever the dimensionality within a reasonable range, a single vector can reliably encode only a limited amount of information. In psychologists' models of cognition, such a vector is called a working memory or a short-term memory, implying the existence also of a long-term memory. That distinction agrees with the traditional organization of a computer where the arithmetic/logic unit and its "active" registers comprise the working memory, and the random-access memory (RAM) is the long-term memory.

The same idea applies to computing with vectors. Operations on vectors output new vectors of the same kind, and a memory stores them for future use. Like the RAM, the memory can be made as large as needed, and large memories are necessary in systems that learn over a long life span. Furthermore, new learning should disrupt minimally what has been learned already. Such memories are called *associative* and were a topic of early neural-net research, but they are not a part of today's deep-learning nets. In deep learning, the memory function and the forming of new representations are entangled.

Memories for high-dimensional vectors have been used in two ways in the examples above. Decoding the vector $\mathbf{s}$ of three superposed variables for the value of x in subsection "Superposing with Addition" (see Fig. 1) produces the vector $\mathbf{a}'$ that needs to be associated with its nearest neighbor among the known vectors. This is the function of the *item memory*: given a noisy vector, output its nearest neighbor among the known vectors. Using the cleaned-up **a** in further computations prevents noise from accumulating. The item memory is addressed with a bipolar vector and it outputs a bipolar vector.

The second use was to compare the profile of a test sentences to language profiles stored in memory. The profiles are (unnormalized) sum vectors with integer components, and their similarity is measured with the cosine. The profiles can be normalized to bipolar (and ultimately to binary) *after* having been accumulated, and then the memory task is identical to that of the item memory. Some information is lost to normalization [Frady et al., 2018], but the loss can be offset in part by higher vector dimensionality [Rahimi et al., 2017].

Postponing the normalization of a sum vector facilitates incremental learning—simply keep adding vectors to the sum. To make it practical, however, the range of sum-vector components needs to be limited and overflow and underflow ignored. Truncation to 8 bits per component has a minor effect on classification tasks such as language identification.

Arrays of H-dimensional vectors can be used as the memory in computer simulations. Searching through them is time-consuming, however, but the simplicity of vector algorithms can still make them practical even when simulated on standard hardware.

Associative memories at large are yet to be fully integrated into the high-dimensional computing architecture. Such memories were studied in the 1970s and 80s [Hinton & Anderson, 1981], with the cerebellum proposed as their realization in the brain [Marr, 1969; Albus, 1971; Kanerva, 1988]. Subsequent lack of interest has a plausible explanation: a versatile algebra for computing with high-dimensional vectors was unknown, and an associative memory by itself isn't particularly useful. This has changed starting with Holographic Reduced Representation in the 1990s [Plate, 1994], and memories for high-dimensional vectors are now included in our models [Karunaratne et al., 2012]. The very large circuits that the memories require, are only now becoming practical.

0.9 Outline of Systems for Autonomous Learning

Systems for computing with vectors derive their power from the remarkable properties of high-(hyper)dimensional spaces. For example, high-dimensional representation is robust and noise-tolerant in ways that human perception, learning and memory are. It also allows data structures to be encoded, manipulated, and decoded explicitly, as in traditional computing, setting it apart from artificial neural nets trained with gradient descent. It allows data to be represented and manipulated in superposition, which sets it apart from traditional computing. The resulting system of computing combines properties that traditional computing and neural nets individually lack. Its properties seem particularly appropriate for modeling of functions controlled by brains. In this section we outline a computing architecture for autonomous learning based on high-dimensional vectors.

Ideas for autonomous learning come from observing the animal world. For an animal to survive and prosper in an environment, it must recognize some situations as favorable and life-sustaining, and others as unfavorable and dangerous, and then act so as to favor the former and avoid the latter: seek reward and avoid punishment. To that effect, animals perceive the environment through a multitude of senses—sight, sound, smell, taste, temperature, pressure, acceleration, vibration, hunger, pain, proprioception—and they move about and act upon the environment by controlling their muscles. The brain's job is to coordinate it all. This can be thought of as a *classification* problem where favorable motor commands serve as the classes to which sensory states are mapped, bearing in mind that classification is a particular strength of computing with high-dimensional vectors [Ge & Parhi, 2020]. What the favorable motor commands are in any sensory state can be learned in any number of ways: by explicit design, following an example, supervised, reinforced, trial-and-error.

Coordinating a variety of sensors and actuators is a major challenge for artificial systems. Different kinds of information need to be represented in a common mathematical space that is not overly confining. As a counter-example, the number line is appropriate for representing temperature but not odor, much less combinations of the two. However, when the dimensionality of the space is high enough, all manner of things can be represented in it—i.e., *embedded*—without them unduly interfering with each other. Generic algorithms can then be used to discover relations among them and to find paths to favorable actions. That brings about the need to map raw sensory signals to high-dimensional vectors, and to map such vectors for action to signals that control actuators.

Each sensory system responds to the environment in its own peculiar way, and so the designing of the interfaces can take considerable engineering expertise. In working with speech, for example, the power spectrum is more useful than the sound wave and is universally used, and there are simple ways to turn spectra into high-dimensional vectors. Mapping a signal onto high-dimensional vectors corresponds to and is no more difficult than designing and selecting features for traditional classification algorithms [Burrello et al., 2020; Moin et al., 2021].

Designing of the interface can also be automated, at least in part, by employing deep learning or genetic algorithms. If we can define an appropriate objective function or measure of fitness, we can let the computer search for a useful mapping of signals to high-dimensional vectors, and from the vectors to commands to actuators. For the system to remain stable, however, the mappings need to remain fixed once they have been adopted, with further learning taking place with vectors in the high-dimensional space. Traditional computing can be used to program such a system and to interface it with the environment.

0.10 Energy-Efficiency

Ideas about computing with vectors need ultimately to be built into hardware [Semiconductor Research Corporation, 2021]. The requirements are fundamentally different from those for computing with numbers. The traditional model assumes determinism—identical inputs produce identical outputs—at a considerable cost in energy. Tasks that are uniquely suited for computing with vectors involve large collections of sensors and actuators of various kind where no single measurement is critical. They matter in the aggregate. When information is distributed equally over all components of a vector, individual components need not be 100% reliable. That makes it possible to use circuits with ever smaller elements and to operate them at very low voltages. Exploiting analog properties of materials also becomes possible, with further gains in energy efficiency [Wu et al., 2018]. In contrast, the requirements of the deterministic model become hard to meet when circuits get ever larger and their elements ever smaller.

The operations on vectors offer further opportunity to reduce the demand for energy. Addition and multiplication happen coordinatewise and thus can be done in parallel, meaning that the total system can be fast without its components needing to be much faster. Furthermore, the simplicity of addition, and particularly of multiplication (e.g., XOR), makes it possible to build them into the memory, reducing the need to bus data between a central processor and the memory [Gupta, Imani & Rosing, 2018]. In traditional computing, both speed and the accessing of memory are paid for in energy.

0.11 Discussion and Future Directions

Computing with vectors has grown out of attempts to understand how brains "compute," ideas for which have come from varied directions. Early evidence was qualitative and was based on observations of behavior and thought experiments, mainly by philosophers and psychologists. After the invention of the digital computer the models became computer-like. However, their ability to explain brains has fallen far short of expectations. Meanwhile, information from neuroscience has been accumulating, starting with neuroanatomy. However, the detailed drawings of neurons and circuits by Cajal in the late 1800s and early 1900s continue to challenge our ability to explain. With advances in neurophysiology, we are able to demonstrate learning in synapses in stereotypical tasks, but the workings of entire circuits still wait to be explained.

Many things suggest that computing with vectors will help us understand brains. Dimensionality in the thousands agrees with the size of neural circuits. High-dimensional distributed representation is extremely robust, and so the failure of a single component is no more consequential than the death of a single neuron. It matters greatly that simple operations on vectors can be made into efficient algorithms for learning. Sufficiently high dimensionality allows many kinds of things to be represented in a common mathematical space without unduly interfering with each other. For example sight and sound can be both kept apart and combined. A high-capacity associative memory is an essential part of computing with vectors. That agrees with the brain's ability to learn quickly and to retain large amounts of information over long periods of time. Among the brain's circuits, the cerebellum's looks remarkably like an associative memory, and it contains over half the brain's neurons. Its importance for motor learning has been known since the 1800s, and it appears to be involved in mental functions as well. Its interpretation as an associative memory goes back half a century [Marr, 1969], and its design can instruct the engineering of high-capacity associative memories for artificial systems.

Other evidence is experiential. The human brain receives infinitely varied input through a 100 million or more sensory neurons, and from it builds a mental world of specific, nameable, repeating, more-or-less permanent colors, sounds, shapes, objects, people, events, stories, histories, and so on. The specificity can be explained by the distribution of distances—and similarity—in a high-dimensional space, and by the tendency of some operations to cluster the inputs. The representation is robust and tolerant of variation and "noise."

Computing with vectors bridges the gap between traditional computing with numbers and symbols on one hand, and artificial neural nets and deep learning on the other. We can expect it to become an established technology for machine learning within a decade, applied widely to multimodal monitoring and control. Its ability to deal with symbols and structure makes it also a candidate technology for logic-based reasoning and language.

Wide-ranging exploration and large-scale experiments are needed meanwhile. New algorithms are easily simulated on standard hardware. Not necessarily in the scale as ultimately desired, but the underlying math makes it possible to see whether an algorithm scales. Semantic vectors provide an example. When made with Latent Semantic Analysis [Landauer & Dumais, 1997], which uses singular-value decomposition, runtime grows with the square of the number of documents, whereas Random Indexing [Kanerva, Kristoferson & Holst, 2000] achieves comparable results in linear time.

Algorithm development needs to proceed on two fronts: *high-dimensional core* and *interfaces*. The core algorithms are generic and are what we think of as computing with vectors—and what this chapter is about. The algorithms at the core integrate input from a multitude of sensors and generate vectors that control the system's output. They also account for learning. The interfaces are specific to sensor and motor modalities and they translate between low-dimensional signal spaces and the high-dimensional representation space as discussed in the section on autonomous learning. The interface in the language-identifying experiment (example 3 on Sequences with Permutation) is extremely simple because languages are already encoded with letters, and representing letters with high-dimensional random seed vectors was all that is needed. Designing an interface usually requires more domain knowledge but is not necessarily difficult.

Against this backdrop, we can try to see what lies ahead. We are far from understanding how brains compute, or from building artificial systems that behave like systems controlled by brains. That would be a monumental achievement, both technological and as a source of insight into human and animal minds. Computing with high-dimensional vectors seems like a necessary step in that direction, even if only a single step of how many, we do not know.

The disparity between brains and our models can give us a clue. If we were given 100 billion neuronlike circuit elements with 100 trillion points of contact between them, akin to synapses, we would not know how to connect them into a system that works. Neither does exact copying of neural circuits yield the understanding we need, unfortunately. But the *large numbers* surely are meaningful and need to be included in our theories and models—and in brain-inspired computing.

Massive feedback is an essential feature throughout the brain, and only rarely is its role apparent. Our theories and models need to come to terms with massive feedback. Activation of neurons in the brain is *sparse* whereas the models discussed in this chapter compute with dense vectors. Brains *learn continuously* with *minuscule energy* compared to neural-net models trained on computer clusters. The fluidity of *human language* will challenge our modeling for years to come.

Much work is needed to fully develop the idea of computing with vectors, but even incremental advances can lead to significant insight and applications. For example, adaptive robotics is likely to benefit early on, where sensor fusion and *sensor–motor integration* are absolute requirements [Räsänen & Saarinen, 2016; Mitrokhin et al., 2019; Neubert, Schubert & Protzel, 2019]. Taking our cues from the animal world, every motor action includes a proprioceptive component that reports on the execution of the action and its outcome. High-dimensional space is natural for dealing with the feedback and incorporating it into future action. From the (nervous)system's point of view, proprioception is just another set of sensory signals, to be integrated with the rest.

Signals of every kind need to be studied from the point of view of mapping them into high-dimensional vectors for further processing. Here again we can look to nature for clues. For example, the cochlea of the inner ear breaks sound into its frequency components before passing the signal on to the brain—it is a Fourier analyzer. No doubt the frequency spectrum tells us more about the sound source than the raw sound wave. The design of an *associative memory* for perhaps thousands-to-millions of vectors is a major engineering challenge. The size and structure of the cerebellum can provide ideas for meeting it.

The statistical nature of the operations means that computing with vectors will not replace traditional computing with numbers. Instead, it allows new algorithms that benefit from high dimensionality, for example by making it possible to learn continuously from streaming data.

Computing with vectors can benefit from hardware trends to the fullest. Because the representation is extremely redundant, circuits need not be 100% reliable. The manufacture of very large circuits that operate with very little energy becomes possible, and it will be possible to compute using the analog properties of materials.

Computing with vectors has been demonstrated here with bipolar vectors, or equivalently with dense binary vectors. To extend it to vectors of other kind, and to other mathematical objects, we need to identify operations on the objects that form a useful computational algebra, and that are also suited for realization in a physical medium. Today that medium is overwhelmingly silicon because of its success at meeting the needs of digital logic. However, the discovery of new materials and physical phenomena will widen our choices and offer new opportunities. Thinking of computing in terms of an algebra of operations on elements of a mathematical space will help us to recognize opportunities as they arise and to develop them to practical systems for computing.

Resources. At the time of this writing there are several websites tracking progress in computing with high-dimensional vectors.

- **Vector Symbolic Architectures aka Hyperdimensional Computing**

  https://www.hd-computing.com/home

- **Collection of Hyperdimensional Computing Projects**

  https://github.com/HyperdimensionalComputing/collection

- **Online Speakers' Corner on Vector Symbolic Architectures and Hyperdimensional Computing**

  https://sites.google.com/ltu.se/vsaonline/home

Computing with vectors of different kind in various contexts are summarized in several papers [Neubert, Schubert & Protzel, 2019; Ge & Parhi, 2020; Hassan et al., 2021; Kleyko et al., 2021].

Acknowledgments. This chapter was written with the support of Defense Advanced Research Projects Agency (DARPA DSO program on Virtual Intelligence Processing, grant #HR00111990072) and Air Force Office of Scientific Research (AFOSR program on Cognitive and Computational Neuroscience, award number FA9550-19-1-0241). Special thanks are due Professor Bruno Olshausen, Director of UC Berkeley's Redwood Center for Theoretical Neuroscience, for fostering an environment supportive of essential research.

Bibliography

Aerts, D., Czachor, M. and De Moor, B. (2009). Geometric analogue of holographic reduced representation. *Journal of Mathematical Psychology* 53(5):389–398. doi: 10.1016/j.jmp.2009.02.005

Albus, J. S. (1971). A theory of cerebellar functions. *Mathematical Biosciences* 10:25–61.

Burrello, A., Schindler, K., Benini, L. and Rahimi, A. (2020). Hyperdimensional computing with local binary patterns: One-shot learning of seizure onset and identification of ictogenic brain regions using short-time iEEG recordings. *IEEE Transactions on Biomedical Engineering* 67(2):601–613.

Eliasmith, C. (2013). *How to Build a Brain: A Neural Architecture for Biological Cognition.* Oxford University Press.

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

Frady, E. P., Kent, S. J., Olshausen, B. A. and 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.

Gallant, S. I. and Okaywe, T. W. (2013). Representing objects, relations, and sequences. *Neural Computation* 25:2038–2078.

Gayler, R. (1998). Multiplicative binding, representation operators, and analogy. In K. Holyoak, D. Gentner and B. Kokinov (eds.), *Advances in Analogy Research: Integration of Theory and Data from the Cognitive, Computational, and Neural Sciences*, p. 405. Sofia, Bulgaria: New Bulgarian University Press.

Gayler, R. W. (2003). Vector Symbolic Architectures answer Jackendoff's challenges for cognitive neuroscience. In Peter Slezak (ed.), *ICCS/ASCS International Conference on Cognitive Science*, pp. 133–138. Sydney, Australia: University of New South Wales.

Gayler, R. W. and Levy, S. D. (2009). A distributed basis for analogical mapping. In B. Kokinov and K. Holyoak (eds.) *New Frontiers in Analogy Research, Proceedings of the Second International Analogy Conference*, pp. 165–174. Sofia: New Bulgarian University Press.

Ge, L. and Parhi, K. K. (2020). Classification using hyperdimensional computing: A review. *IEEE Circuits and Systems Magazine* 20(2):30–47.

Gupta, S., Imani, M. and Rosing, T. (2018). FELIX: Fast and energy-efficient logic in memory. *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, doi: 10.1145/3240765.3240811

Hassan, E., Halawani, Y., Mohammad, B. and Saleh, H. (2021). Hyper-dimensional computing challenges and opportunities for AI applications. *IEEE Access*, 15 pp. doi: 10.1109/ACCESS.2021.3059762

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

Hinton, G. E. and Anderson, J. A., eds. (1981) *Parallel Models of Associative Memory*, (updated edition 1989). Hillsdale, N.J.: Lawrence Erlbaum.

Joshi, A., Halseth, J. T. and Kanerva, P. (2017). Language geometry using random indexing. In J. A. de Barros, B. Coecke and E. Pothos (eds.), *Quantum Interaction: 10th International Conference*, QI 2016, San Francisco, CA, July 20–22, 2016, Revised Selected Papers, pp. 265–274. Cham, Switzerland: Springer International Publishing.

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

Kanerva, P. (1996). Binary spatter-coding of ordered K-tuples. In C. von der Malsburg, W. von Seelen, J. C.Vorbruggen, B. Sendhoff (eds.) *Artificial Neural Networks – ICANN 96 Proceedings* (Lecture Notes in Computer Science, vol. 1112), pp. 869–873. Berlin: Springer.

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

Kanerva, P., Kristoferson, J. and Holst, A. (2000). Random indexing of text samples for Latent Semantic Analysis. In L.R. Gleitman and A. K. Josh (eds.), *Proc. 22nd Annual Meeting of the Cognitive Science Society* (CogSci'00, Philadelphia), p. 1036. Mahwah, New Jersey: Erlbaum.

Karunaratne, G., Schmuck, M., Le Gallo, M., Cherubini, G., Benini, L., Sebastian, A. and Rahimi, A. (2021). Robust high-dimensional memory-augmented neural networks. *Nature Communications* 12, Article number 2468 (2021), 12 pp. doi: 10.1038/S41467-021-22364-0

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

Kleyko, D., Davies, M., Frady, E. P., Kanerva, P., Kent, S. J., Olshausen, B. A., Osipov, E., Rabaey, J. M., Rachkovskij, D. A., Rahimi, A. and Sommer, F. T. (2021) Vector Symbolic Architectures as a computing framework for nanoscale hardware. arXiv:2106.05268v1 [cs.AR] 9 June 2021.

Landauer, T. and Dumais, S. (1997). A solution to Plato's problem: the Latent Semantic Analysis theory of acquisition, induction and representation of knowledge. *Psychological Review* 104(2):211–240.

Levy, S. D. and Gayler, R. (2008). Vector Symbolic Architectures: A new building material for artificial general intelligence. *Proceedings of the First*

Conference on Artificial General Intelligence (AGI-08). IOS Press.

Marr, D. (1969). A theory of cerebellar cortex. Journal of Physiology (London) 202:437–470.

Mitchell, M. (2019). *Artificial Intelligence: A Guide to Thinking Humans.* New York, NY: Farrar, Straus and Giroux.

Mitrokhin, A., Sutor, P., Fermüller, C. and Aloimonos, Y. (2019). Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception. *Science Robotics* 4, eaaw6736 (15 May 2019), 10 pp. doi: 10.1126/scirobotics.aaw6736

Moin, A., Zhou, A., Rahimi, A., Menon, A., Benatti, S., Alexandrov, G., Tamakloe, S., Ting, J., Yamamoto, N., Khan, Y., Burghardt, F., Benini, L., Arias, A. C. and Rabaey, J. M. (2021). A wearable biosensing system with in-sensor adaptive machine learning for hand gesture recognition. *Nature Electronics* 4(1):54–63.

Murdock, B. B. (1982). A theory for the storage and retrieval of item and associative information. *Psychological Review* 89(6):609–626. doi: 10.1037/0033-295X.89.6.609

Neubert, P., Schubert, S. and Protzel, P. (2019). An introduction to hyperdimensional computing for robotics. *Künstliche Intelligenz volume* 33:319–330. doi: 10.1007/s13218-019-00623-z

Osipov, E., Kleyko, D. and Legalov, A. (2017). Associative synthesis of finite state automata model of a controlled object with hyperdimensional computing. *IECON 2017—43rd Annual Conference of the IEEE Industrial Electronics Society*, pp 3276–3281. doi: 10.1109/IECON.2017.8216554.

Plate, T. A. (1994). Distributed Representations and Nested Compositional Structure. Doctoral dissertation, University of Toronto.

Plate, T. A. (2003). *Holographic Reduced Representation: Distributed Representation of Cognitive Structure.* Stanford, CA: CSLI Publications.

Rachkovskij, D. A. and Kussul, E. M. (2001). Binding and normalization of binary sparse distributed representations by context-dependent thinning. *Neural Computation* 13(2):411–452.

Rahimi, A., Datta, S., Kleyko, D., Frady, E. P., Olshausen, B., Kanerva, P.

and Rabaey, J. M. (2017). High-dimensional computing as a nanoscalable paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64(9):2508–2521. doi: 10.1109/tcsi.2017.2705051

Räsänen, O. J. and Saarinen, J. P. (2016). Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns. *IEEE Transactions on Neural Networks and Learning Systems* 27(9):1878–1889.

Semiconductor Research Corporation (2021). New compute trajectories for energy-efficient computing. *Decadal Plan for Semiconductors, Full Report*, pp.122–146. https://www.src.org/about/decadal-plan

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

Widdows, D. and Cohen, T. (2015). Reasoning with vectors: A continuous model for fast robust inference. *Logic Journal of the IGPL* 23(2):141–173. doi: 10.1093/jigpal/jzu028

Wu, T. F., Li, H., Huang, P.-C., Rahimi, A., Hills, G., Hodson, B., Hwang, W., Rabaey, J. M., Wong, H.-S. P., Shulaker, M. M. and Mitra, S. (2018). Hyperdimensional computing exploiting carbon nanotube FETs, resistive RAM, and their monolithic 3D integration. *IEEE Journal of Solid-State Circuits* 53(11):3183–3196.