CONTRIBUTED PAPER

Vector Symbolic Architectures as a Computing Framework for Emerging Hardware

This article reviews recent progress in the development of the computing framework referred to as vector symbolic architectures, or hyperdimensional computing.

By Denis Kleyko, *Member IEEE*, Mike Davies, *Member IEEE*, Edward Paxon Frady, Pentti Kanerva, Spencer J. Kent, Bruno A. Olshausen, Evgeny Osipov, Jan M. Rabaey, *Life Fellow IEEE*, Dmitri A. Rachkovskij, Abbas Rahimi, and Friedrich T. Sommer

Manuscript received 11 April 2022; revised 30 June 2022; accepted 2 September 2022. Date of current version 17 October 2022. The work of Denis Kleyko was supported in part by the European Union's Horizon 2020 Research and Innovation Programme under Marie Skłodowska-Curie Grant Agreement 839179, in part by the Defense Advanced Research Projects Agency's (DARPA) VIP (Super-HD Project) and AIE (HyDDENN Project) Programs, in part by the Air Force Office of Scientific Research (AFOSR) under Grant FA9550-19-1-0241, and in part by the Intel's THWAI Program. The work of Pentti Kanerva was supported in part by the DARPA's VIP (Super-HD Project) and AIE (HyDDENN Project) Programs and in part by AFOSR under Grant FA9550-19-1-0241. The work of Bruno A. Olshausen was supported in part by the DARPA's VIP (Super-HD Project) and AIE (HyDDENN Project) Programs, in part by AFOSR under Grant FA9550-19-1-0241, and in part by the Intel's THWAI Program. The work of Jan M. Rabaey was supported in part by the DARPA's VIP (Super-HD Project) and AIE (HyDDENN Project) Programs. The work of Dmitri A. Rachkovskij was supported in part by the National Academy of Sciences of Ukraine under Grant 0120U000122, Grant 0121U000016, Grant 0122U002151, and Grant 0117U002286; in part by the Ministry of Education and Science of Ukraine under Grant 0121U000228 and Grant 0122U000818; and in part by the Swedish Foundation for Strategic Research (SSF) under Grant UKR22-0024. The work of Friedrich T. Sommer was supported in part by the Intel's THWAI Program, in part by NIH under Grant R01-EB026955, and in part by NSF under Grant IIS-1718991. *(Corresponding author: Denis Kleyko.)*

Denis Kleyko is with the Redwood Center for Theoretical Neuroscience, University of California at Berkeley, Berkeley, CA 94720 USA, and also with the Intelligent Systems Laboratory, Research Institutes of Sweden, 16440 Kista, Sweden (e-mail: denis.kleyko@ri.se).

Mike Davies and **Edward Paxon Frady** are with the Neuromorphic Computing Laboratory, Intel Labs, Santa Clara, CA 95054 USA (e-mail: mike.davies@intel.com; e.paxon.frady@intel.com).

Pentti Kanerva, **Spencer J. Kent**, and **Bruno A. Olshausen** are with the Redwood Center for Theoretical Neuroscience, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: pkanerva@berkeley.edu; spencer.kent@berkeley.edu; baolshausen@berkeley.edu).

Evgeny Osipov is with the Department of Computer Science Electrical and Space Engineering, Luleå University of Technology, 97187 Luleå, Sweden (e-mail: evgeny.osipov@ltu.se).

Jan M. Rabaey is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: jan_rabaey@berkeley.edu).

Dmitri A. Rachkovskij is with the International Research and Training Center for Information Technologies and Systems, 03680 Kyiv, Ukraine, and also with the Department of Computer Science Electrical and Space Engineering, Luleå University of Technology, 97187 Luleå, Sweden (e-mail: dar@infrm.kiev.ua).

Abbas Rahimi is with IBM Research–Zurich, 8803 Rüschlikon, Switzerland (e-mail: abr@zurich.ibm.com).

Friedrich T. Sommer is with the Redwood Center for Theoretical Neuroscience, University of California at Berkeley, Berkeley, CA 94720 USA, and also with the Neuromorphic Computing Laboratory, Intel Labs, Santa Clara, CA 95054 USA (e-mail: fsommer@berkeley.edu).

Digital Object Identifier 10.1109/JPROC.2022.3209104

ABSTRACT | This article reviews recent progress in the development of the computing framework *vector symbolic architectures* (VSA) (also known as hyperdimensional computing). This framework is well suited for implementation in stochastic, emerging hardware, and it naturally expresses the types of cognitive operations required for artificial intelligence (AI). We demonstrate in this article that the field-like algebraic structure of VSA offers simple but powerful operations on high-dimensional vectors that can support all data structures and manipulations relevant to modern computing. In addition, we illustrate the distinguishing feature of VSA, "computing in superposition," which sets it apart from conventional computing. It also opens the door to efficient solutions to the difficult combinatorial search problems inherent in AI applications. We sketch ways of demonstrating that VSA are computationally universal. We see them acting as a framework for computing with distributed representations that can play a role of an abstraction layer for emerging computing hardware. This article serves as a reference for computer architects by illustrating the philosophy behind VSA, techniques of distributed computing with them, and their relevance to emerging computing hardware, such as neuromorphic computing.

KEYWORDS | Computing framework; computing in superposition; data structures; distributed representations; emerging hardware; holographic reduced representation (HRR); hyperdimensional (HD) computing; Turing completeness; vector symbolic architectures (VSA).

I. Introduction

The demands of computation are changing. First, artificial intelligence (AI) and other novel applications pose a host of computing problems that require a search over an immense space of possible solutions, with many approximately correct answers, but rarely a single correct one. Second, future emerging hardware platforms, operating

at ultralow voltages to reduce energy consumption and support continued process scaling, are destined to be noisy and, hence, operate stochastically [1]. These observations expose the need for a computing framework that supports both deterministic computation in the presence of noise, as well as the approximate and parallel nature of algorithms used in AI.

By emerging hardware, we refer to the broad class of new hardware designs that are highly parallel, fabricated at ultrasmall scales, utilize novel components, and/or operate at ultralow voltages, thus consisting of unreliable, stochastic computational elements.

The conventional (à la von Neumann) computing architecture is not well adapted to these demands, as it was designed assuming precise computational elements for tasks that require exact answers. Conventional computing architectures will continue to play an important role in technology, but there is a growing amount of computational demands that are better served by new computing designs. Thus, hardware engineers have been looking at distributed and neuromorphic computing as a way of meeting these demands.

Many of the emerging computational demands come from cognitive or perceptual applications found within the realm of AI. Examples include image recognition, computer vision, and text analysis. Indeed, large-scale deep learning neural network modeling dominates discussions about modern computing technology, pushing innovations in hardware design toward parallel, distributed processing [2]. While widely used, deep learning neural networks still have limitations, such as lacking the transparency of learned representations and the difficulties in performing symbolic computations. In order to support more sophisticated symbolic computations, researchers have been embedding conventional data structures, such as graphs and key-value pairs, into neural network models [3], [4], [5]. However, it is not yet clear whether the subsymbolic pattern recognition and learning capabilities of deep neural networks can be augmented to handle the rich control flow, abstraction, symbol manipulation, and recursion of existing computing frameworks.

Work on developing emerging computing hardware is accelerating. There are many showcase demonstrations [6], [7], [8], [9], but, so far, the following holds.

1) These demonstrations have mostly lacked a unifying theoretical framework that can bring sufficient composability, explainability, and versatility.
2) Many demonstrations still depend on handcrafted elements that would be prone to errors.
3) Most of the demonstrations have been subsymbolic in nature and resort to support from the conventional computing architecture to implement the symbolic and flow control elements.

While these points are valid in general, there are some exceptions that we discuss in Section VI-B. Nevertheless, all of these demonstrate the need for a unifying computing framework that can serve as an abstraction layer between the hardware and desired functionality. Ideally, such a framework should be flexible enough to provide interfaces to emerging hardware with various features, such as stochastic components, asynchronous spiking communication, or devices with analog elements.

For the following reasons, we propose vector symbolic architectures (VSA) [10] or, synonymously, hyperdimensional (HD) computing [11], such as a computing framework. First, HD/VSA can represent and manipulate both symbolic and numerical data structures with distributed vector representations to solve, e.g., cognitive [12], [13], [14] or machine learning [15] tasks. HD/VSA is a suitable framework for integration with neural network computations for solving problems in AI. It extends beyond typical AI tasks as an approach capable of performing symbolic manipulations with distributed representations. Second, the design of HD/VSA, which was inspired by the brain, lends itself to implementation in emerging computing technologies [16] because it is highly robust to individual device variations. Third, HD/VSA is a framework with two interfaces: one toward computations and algorithms and one toward implementation and representations (cf. Fig. 1). There are different HD/VSA models that all offer the same operation primitives but differ slightly in terms of their implementation of these primitives. For example, there are HD/VSA models that compute with binary, bipolar, continuous real, and continuous complex vectors. Thus, the HD/VSA concept has the flexibility to connect to a multitude of different hardware types, such as analog in-memory computing architectures [16] for binary-valued HD/VSA models or spiking neuron architectures [17], [18] for complex-valued ones.

HD/VSA is a relatively new concept. The key idea goes back to the 1990s, but computers of the day were not ready to process large numbers of high-dimensional vectors in parallel. Now hardware with such capabilities is emerging, and, thus, the HD/VSA framework deserves to be looked into anew, not to substitute conventional computing, but to complement it in specific challenging domains. For example, human and animal-like perception and learning have eluded our attempts to be programmed into computers. HD/VSA is a strong candidate for such tasks because of its suitability for both statistical learning and symbolic reasoning.

This article provides three main contributions. First, we review the principles of HD/VSA and how they provide a generic computing framework for implementing the primitives of conventional data structures and deterministic algorithms. Second, we highlight the pros and cons of a nontraditional mode of computing in HD/VSA, "computing in superposition," which can leverage distributed representations and parallelism for efficiently solving computationally hard problems. Finally, we present two proposals (see Appendix A) that show the universality of HD/VSA by using them to represent systems known to be Turing complete.

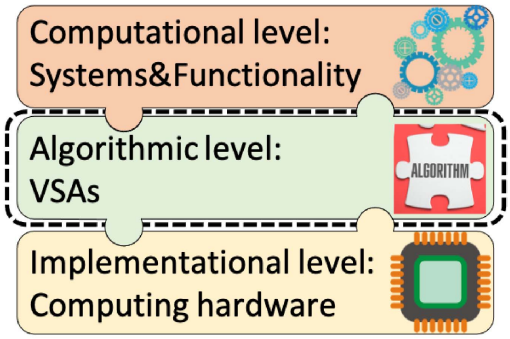


Fig. 1. *Place of HD/VSA within Marr's levels of analysis [19]. The focus of this article is marked by the dashed rectangle. We explain how HD/VSA provides primitives to formalize algorithms in ways that seamlessly connect to the computational and implementational levels in the computing hierarchy.*

A. Guide to This Article

This article is written with both newcomers to HD/VSA and seasoned readers in mind. Section II provides some motivation for using HD/VSA in the context of emerging computing hardware. This section sets up the context for this article. Section III offers a deep dive into the fundamentals of HD/VSA, recommended primarily to readers not yet familiar with the framework. Section IV explains different aspects of computing with HD/VSA, including a "cookbook" for the representation primitives for numerous data structures (see Section IV-A) and introducing an idea of computing in superposition and its existing applications (see Section IV-B). Current hardware realizations of HD/VSA models are considered in Section V. Section VI provides the discussion. Finally, Appendix A describes proposals for implementing two Turing complete systems with HD/VSA.

II. MOTIVATION

The exponential growth of big data and AI applications exposes fundamental limitations of the conventional computing framework. One problem is that energy efficiency is stagnating [20]—-training and fine-tuning a neural network for a natural language processing application consume energy and computational resources equivalent to several hundred thousand U.S. dollars [21] or more [22]. Conventional computing hardware is also highly susceptible to errors, and energy is often "wasted" attempting to maintain low error rates.

Data-intensive applications illustrate the scale of the problem and make energy efficiency the grand challenge of computer engineering. To solve this challenge, alternative hardware is required that can work with imprecise and unreliable computational elements [1]. Operating at ultralow voltages with stochastic devices that are prone to errors has the potential to greatly increase computing power and efficiency. For example, the recent advances in materials science and device manufacturing make it possible to design computing hardware that accommodates computational principles of biological brains or exploits the physical properties of the substrate material. For certain classes of problems, computing hardware, such as neuromorphic processors [23], [24], [25] and in-memory computing architectures [16], consumes only a fraction of the energy compared to current technology. For certain tasks, existing neuromorphic platforms can be 1000 times more energy efficient [24] than the conventional ones.

There is currently a focus on implementing AI capabilities in emerging computing hardware [25], with the aim of providing an energy-efficient implementation of a selected class of AI functionalities (mainly neural networks). However, we see the opportunity for a computational framework exceeding neural networks in scope, which could empower an unprecedented breakthrough in emerging computing technology. First, while neural network algorithms serve a rather small subset of computation problems extremely well, they are unable to address a large class of problems that require conventional algorithms and data structures. A computing framework with a broader application scope than neural networks could boost the adoption of emerging computing by several orders of magnitude. Second, despite many promising applications for emerging computing hardware, the programming of any new functionality is far from trivial. Emerging computing hardware currently lacks a holistic software architecture, which would streamline the development of the new functionality. Current development strategies resemble those of assembly programming, where the developer is left with the entire job—from coming up with the algorithmic idea to designing the actual machine instructions to be executed by a central processing unit. Thus, the impressive recent emerging hardware development [16], [26] needs to be complemented with the creation of computing frameworks for such hardware, which can abstract and simplify the implementation of new functionalities, including the design of programs. Last but not least, most emerging hardware differs fundamentally from traditional computer and neural network accelerator hardware in that the enabled computations are unreliable and stochastic. Thus, a computing framework is required in which error correction and error robustness are achieved.

There is ample work demonstrating that HD/VSA possesses a rich computational expressiveness from the functionality of neural networks [27], [28], [29], [30] to machine learning tasks [31], [32], [33], [34], [35] and cognitive modeling [13], [14], [36], [37], [38], [39]. Furthermore, HD/VSA can express conventional algorithms, for example, finite state automata [40], [41] and context-free grammars [42].

In this article, we explore whether HD/VSA can serve as a computing framework for taking emerging computing to the next level. We argue that HD/VSA provides a framework to formalize and modularize algorithms and, at the same time, bridge the computation and implementation

levels in Marr's framework [19] for information processing systems (see Fig. 1). Our proposal generalizes earlier suggestions to apply HD/VSA for implementing specific machine learning algorithms on emerging hardware [43], [44].

III. FUNDAMENTALS OF HD/VSA

HD/VSA [10], [11] is the term for a family of models for representing and manipulating data in a high-dimensional space. It was originally proposed in cognitive psychology and cognitive neuroscience as a connectionist model for symbolic reasoning [45]. In HD/VSA, data objects are represented by vectors of high (but fixed) dimension N, sometimes called hypervectors or HD vectors. The encoded information is distributed across all components of a hypervector. Such distributed representations [46] are distinct from localist and semilocalist representations [47], where single or subsets of components encode individual data objects.

Distributed representations are, in and of themselves, not the full story. As argued by Fodor and Pylyshyn [48], distributed representations must be productive and systematic. Productivity refers to massive expressiveness generated by simple primitives, while systematicity means that representations are sensitive to the structure of the encoded objects. These desiderata were one of the drivers for developing HD/VSA. One major advantage of HD/VSA as the algorithmic level in the Marr hierarchy (see Fig. 1) is that it embraces distributed representations, which are robust to local noise.

The idea of computing with random hypervectors as basic objects rather than Boolean or numeric scalars was developed by Kussul *et al.* [49] as part of associative-projective neural networks and independently in seminal works by Smolensky [50] on tensor product variable binding and Plate on holographic reduced representation (HRR) [51]. HD/VSA can be formulated with different types of vectors, namely, those containing real, complex, or binary entries, as well as with the multivectors from geometric algebra. These HD/VSA models come under many different names: HRR [52], [53], multiply–add–permute (MAP) [54], binary spatter codes [55], sparse binary distributed representations (SBDRs) [56], [57], sparse block codes [58], [59], matrix binding of additive terms (MBAT) [60], the geometric analog of HRR (GAHRR) [61], and so on. All of these different models have similar computational properties—see [30] and [62]. For clarity, we will use the MAP model in the remainder of this article.

A. Basic Elements of HD/VSA

1) High-Dimensional Space: HD/VSA requires a high-dimensional space. The appropriate choice of dimensionality N is somewhat dependent on the problem, but there are simple rules of thumb ($N > 1000$, for example), and the representation of particular data structures in the given problem is much more important. As mentioned above, there are HD/VSA models defined for different types of spaces (see Section V-A for more details). In this article, we will use a variation of the MAP model (MAP-I; see [62]) that operates in integer vector spaces ($\mathbb{Z}^N$). Operations and properties that have proven useful are presented in the following (Appendix B provides the summary). It is worth pointing out that the superposition and binding of hypervectors form an algebraic structure that resembles a field, and permutations extend the algebra to all finite groups up to size N.

2) Quasi-Orthogonality: HD/VSA uses random (strictly speaking, pseudorandom) vectors as a means for data representation. By using random vectors as representations, HD/VSA can exploit the concentration of measure phenomenon [63], [64], which implies that, with high probability, random vectors become almost orthogonal in high-dimensional vector spaces. This phenomenon is sometimes called progressive precision [65] or the blessing of dimensionality [64]. In the case of HD/VSA, it means that, when, e.g., two objects are represented by random vectors, with high probability, their representations will be almost orthogonal to each other. MAP uses bipolar random vectors where the ith component of a vector $\mathbf{a}$ is generated independent identically distributed (i.i.d.) random from the Bernoulli distribution: $a_i \sim 2\mathcal{B}(0.5) - 1$. In the HD/VSA literature, dissimilar representations are described by various adjectives, such as unrelated, uncorrelated, approximately, pseudo-orthogonal, or quasi-orthogonal. Unlike exact orthogonality, the dimension N is not a hard limit on the number of quasi-orthogonal vectors that one can create.

3) Similarity Measure: Processing in HD/VSA is based on the similarity between hypervectors. The common similarity measures in HD/VSA are the dot (scalar, inner) product, cosine similarity, overlap, and Hamming distance. In MAP, it is common to use either the cosine similarity or the dot product. Therefore, we will be using the dot product (denoted as $\langle \cdot, \cdot \rangle$) as the similarity measure in the following.

4) Seed Hypervectors: When designing an HD/VSA algorithm for solving a problem, it is common to define a set of the most basic concepts/symbols for the given problem and assign hypervectors to them. Such seed hypervectors are defined as representations of concepts that are irreducible. All other hypervectors occurring in the course of computation are, therefore, reducible, that is, they are composed of seed hypervectors. Here, we will focus on symbolic structures, i.e., symbols from some alphabet with size D, which are represented by i.i.d. random seed hypervectors (see Section III-A2). As mentioned above, in MAP, seed hypervectors are bipolar and so any hypervector $\mathbf{a} \in \{-1, 1\}^N$. The process of assigning seed hypervectors, usually (but not always) by i.i.d. random generation of vectors, is referred to as mapping, encoding, projection,

or embedding. We reiterate that representations in an HD/VSA algorithm need not always be quasi-orthogonal. For example, for representing real-valued variables, one might use a locality-preserving representation scheme, in which representations of similar values are systematically correlated and not quasi-orthogonal [66], [67], [68], or where the hypervectors are learned [31], [69]. Thus, one should keep in mind that i.i.d. randomness is not the only tool for designing seed representations.

5) Item Memory: Seed hypervectors are stored in the so-called item memory (or cleanup memory), a content-addressable memory that can be just a matrix, or an associative memory [70], [71], [72] that stores the hypervectors as point attractors.

B. HD/VSA Operations and Compound Representations

Seed hypervectors are the building blocks for compound HD/VSA representations, which are built from operations performed on the seed vectors. For example, a compound hypervector representing the edges of a graph (compound entity) can be constructed (see Section IV-A7) from seed hypervectors representing its nodes (basis symbols). This compositional formation of data structures in HD/VSA is akin to conventional computing and very different from the modern neural networks in which activity vectors, especially in hidden layers, often cannot be readily parsed.

Two key HD/VSA operations are dyadic vector operations between hypervectors that are referred to as *superposition* and *binding*. Like the corresponding operations between ordinary numbers, they form, together with the representation vector space, a field-like algebraic structure. Another important HD/VSA operation is the *permutation* of components within a hypervector.

The componentwise addition operation is used for bundling or superposing, and in the MAP model, it is implemented as a componentwise addition of hypervectors. The binding operation is used for variable binding. In the MAP model, the binding operation is implemented via componentwise multiplication, i.e., via the Hadamard product. The permutation operation, as its name suggests, shuffles the components of a hypervector according to a predefined permutation that can be, e.g., chosen randomly. In practice, a rotation of components, i.e., a cyclic shift of the hypervector component index, is used frequently.

In what follows, we describe each operation and its properties in more detail. It is important to stress that various HD/VSA models differ in the particular details of realizing their operations. As a consequence, the operations' properties presented below are relevant for the MAP model but are not valid for each and every HD/VSA model. For the sake of focus, we will not discuss differences between different HD/VSA models in depth here, but we encourage interested readers to consult recent studies [62], [73].

Note also that the seed hypervectors referred to in this section are pseudorandom i.i.d. Because high-dimensional representation tolerates errors, the conditions listed in the following need only be satisfied approximately or with high probability. Due to the concentration of measure phenomenon, the operations—and computations based on them—become ever more reliable, dependable, and predictable as the dimensionality N of the space increases.

1) Binding: A dyadic operation mapping two hypervectors to another hypervector. It is used to represent an object formed by the binding of two other objects. This operation is an important ingredient for forming compositional structures with distributed representations (see a discussion on its importance in the context of deep learning in [74]). Formally, for two objects a and b, represented by the hypervectors $\mathbf{a}$ and $\mathbf{b}$, the hypervector that represents the bound object (denoted by $\mathbf{m}$) is

$$\mathbf{m} = \mathbf{a} \odot \mathbf{b}. \tag{1}$$

In the MAP model, $\odot$ denotes the componentwise multiplication (Hadamard product). Multiple application of binding is denoted by $\prod$, enabling the formation of a hypervector representing the product of more than two hypervectors.

Consider the example of representing a database for trivia about countries [75]. The database record for a country contains the name, the capital, and the currency. The first step is to form hypervectors that represent key-value pairs, which can be done by binding: **country** $\odot$ **USA**, **capital** $\odot$ **Washington**, **currency** $\odot$ **USD**. To create a single hypervector that represents the entire data record for a country, we need another operation to combine the different key-value pairs (see below).

2) Superposition: A dyadic operation mapping two hypervectors to another hypervector. It is denoted with $+$ and, in the MAP model, implemented via componentwise addition, which sometimes can be thresholded to keep bipolar representations (not used in this article). The superposition operation combines several hypervectors into a single hypervector. For example, for $\mathbf{a}$ and $\mathbf{b}$, the result $\mathbf{z}$ of the superposition of their hypervectors is simply

$$\mathbf{z} = \mathbf{a} + \mathbf{b}. \tag{2}$$

The superposition of more than two hypervectors is denoted by $\sum$. Often, superposition is followed by a thresholding operation to produce a resultant hypervector that is of the same type as the seed vectors. For example, in the MAP model, the seed hypervectors are bipolar vectors, but the arithmetic sum-vector is not. Therefore, in the bipolar variant (MAP-B; see [62]), a thresholding operation, using the signs in each component, can map

the sum vector back to a bipolar hypervector. This type of thresholding is sometimes called the majority rule/sum and denoted by brackets: $[\mathbf{a}+\mathbf{b}]$. For the sake of consistency, the examples in the following use the nonthresholded sum, unless mentioned otherwise.

The nonthresholded sum has the advantage of being invertible since individual elements in the sum can be removed by subtraction (denoted as $-$) without interfering with the rest. Using the example above

$$\mathbf{a} = \mathbf{z} - \mathbf{b}. \tag{3}$$

Continuing the database example, the superposition operation can be used to create a single hypervector from hypervectors representing all key-value pairs of the record. Thus, the compound hypervector for the whole record will be formed as follows: **country** $\odot$ **USA** $+$ **capital** $\odot$ **Washington** $+$ **currency** $\odot$ **USD**.

3) Permutation: A unary operation on a hypervector that yields a hypervector. Akin to the binding operation, permutation is often used to map into an area of hypervector space that does not interfere strongly with other representations. However, unlike binding in MAP, the same permutation can be used recursively, projecting into previously unoccupied space with every iteration. Note that the number of possible permutations grows superexponentially with the dimensionality ($N!$), and permutations themselves are not elements of the space of representations. In most HD/VSA algorithms, a single one or a small set of permutations are fixed at the onset of computation. We continue with a simple example, and more examples follow in Sections IV-A5, IV-A8, and IV-A10.

Permutation can be seen as an alternative approach to binding when there is only one hypervector as the operand [54]. The permutation operation can also be used to represent sequence relations and other asymmetric relations like "part-of." For example, a fixed permutation (denoted as $\rho(\cdot)$) can be used to associate, e.g., a symbol hypervector with the position of a symbol in a sequence, resulting in a hypervector representing the symbol in that position. The single application of the permutation is

$$\mathbf{r} = \rho^1(\mathbf{a}) = \rho(\mathbf{a}). \tag{4}$$

To associate $\mathbf{a}$ with the ith position in a sequence, the permutation is applied i times. The result is the hypervector

$$\mathbf{r} = \rho^i(\mathbf{a}).$$

Note that permutation is an example of a more general unary operation, matrix-vector multiplications (see [60] for a proposal on using matrix-vector multiplications to implement the binding operation).

4) Properties of HD/VSA Operations and Their Interaction: Here, we summarize the properties of the basic HD/VSA operations and how they interact.

a) Superposition: The superposition operation has the following properties:

1) Superposition can be inverted with subtraction: $\mathbf{a}+\mathbf{b}+\mathbf{c}-\mathbf{c}=\mathbf{a}+\mathbf{b}$.
2) In contrast to the binding and permutation operations, the result of the superposition $\mathbf{z}=\mathbf{a}+\mathbf{b}$ (often called the superposition hypervector) is similar to each of its argument hypervectors, i.e., the dot product between $\mathbf{z}$ and $\mathbf{a}$ or $\mathbf{b}$ is significantly more than 0: $\langle \mathbf{z}, \mathbf{a}\rangle \approx \langle \mathbf{z}, \mathbf{b}\rangle > 0$.
3) Arguments of binding can be approximately recovered from the superposition hypervector: $\mathbf{b} \odot (\mathbf{a} \odot \mathbf{b} + \mathbf{c} \odot \mathbf{d}) \approx \mathbf{a}$.
4) Superposition is commutative: $\mathbf{a}+\mathbf{b}=\mathbf{b}+\mathbf{a}$.
5) Thresholded superposition is approximately associative: $[[\mathbf{a}+\mathbf{b}]+\mathbf{c}] \approx [\mathbf{a}+[\mathbf{b}+\mathbf{c}]]$.

Note that, if several instances of any hypervector are included (e.g., $\mathbf{z}=3\mathbf{a}+\mathbf{b}$), the resultant hypervector is more similar to the dominating hypervector than to other arguments.

b) Binding: The binding operation has the following properties:

1) Binding is commutative: $\mathbf{a} \odot \mathbf{b} = \mathbf{b} \odot \mathbf{a}$.
2) Binding distributes over superposition: $\mathbf{c} \odot (\mathbf{a}+\mathbf{b}) = (\mathbf{c} \odot \mathbf{a}) + (\mathbf{c} \odot \mathbf{b})$.
3) Binding is invertible for $\mathbf{m} = \mathbf{a} \odot \mathbf{b}$: $\mathbf{a} \odot \mathbf{m} = \mathbf{b}$. The inversion process is often called releasing or unbinding. In the case of the componentwise multiplication of bipolar vectors, the unbinding operation is performed with the same operation. Therefore, we do not introduce a separate notation for unbinding here.
4) Binding is associative: $\mathbf{c} \odot (\mathbf{a} \odot \mathbf{b}) = (\mathbf{c} \odot \mathbf{a}) \odot \mathbf{b}$.
5) The result of binding is dissimilar to each of its argument hypervectors, e.g., $\mathbf{m}$ is dissimilar to the hypervectors being bound, i.e., the dot product between $\mathbf{m}$ and $\mathbf{a}$ or $\mathbf{b}$ is approximately 0: $\langle \mathbf{m}, \mathbf{a}\rangle \approx \langle \mathbf{m}, \mathbf{b}\rangle \approx 0$.
6) Binding preserves similarity (for similar $\mathbf{a}$ and $\mathbf{a}'$): $\langle \mathbf{a} \odot \mathbf{b}, \mathbf{a}' \odot \mathbf{b}\rangle \gg 0$.
7) Binding is a "randomizing" operation (since $\langle \mathbf{a} \odot \mathbf{b}, \mathbf{a}\rangle \approx 0$) that preserves similarity (because $\langle \mathbf{a} \odot \mathbf{b}, \mathbf{c} \odot \mathbf{b}\rangle = \langle \mathbf{a}, \mathbf{c}\rangle$).

c) Permutation: The permutation operation has the following properties:

1) Permutation is invertible for $\mathbf{r}=\rho(\mathbf{a})$: $\mathbf{a}=\rho^{-1}(\mathbf{r})$.
2) In MAP, permutation distributes over both binding ($\rho(\mathbf{a} \odot \mathbf{b}) = \rho(\mathbf{a}) \odot \rho(\mathbf{b})$) and superposition ($\rho(\mathbf{a}+\mathbf{b}) = \rho(\mathbf{a}) + \rho(\mathbf{b})$).
3) Similar to the binding operation, the result $\mathbf{r}$ of a (random) permutation is dissimilar to the argument hypervector $\mathbf{a}$: $\langle \mathbf{r}, \mathbf{a}\rangle \approx 0$.
4) Permutation is a "randomizing" operation (since $\langle \rho(\mathbf{a}), \mathbf{a}\rangle \approx 0$) that preserves similarity (because $\langle \rho(\mathbf{a}), \rho(\mathbf{b})\rangle = \langle \mathbf{a}, \mathbf{b}\rangle$).

It is worth clarifying what we mean by "similarity preserving" in the case of binding and permutation versus superposition above: For binding, the similarity between

two hypervectors is the same before and after binding with a third hypervector, i.e., $\langle \mathbf{a} \odot \mathbf{b}, \mathbf{c} \odot \mathbf{b} \rangle = \langle \mathbf{a}, \mathbf{c} \rangle$, and for permutation, the similarity between the two hypervectors is also the same before and after the operation, i.e., $\langle \rho(\mathbf{a}), \rho(\mathbf{b}) \rangle = \langle \mathbf{a}, \mathbf{b} \rangle$. However, for superposition, the similarity between two hypervectors is generally lower before versus after superimposing them to a third hypervector, i.e., $\langle \mathbf{a} + \mathbf{b}, \mathbf{c} + \mathbf{b} \rangle > \langle \mathbf{a}, \mathbf{c} \rangle$, since the sum moves them in a common direction $\mathbf{b}$. On the other hand, since the superposition hypervector is similar to each of the vectors in the sum, $\langle \mathbf{a} + \mathbf{b}, \mathbf{a} \rangle \approx \langle \mathbf{a} + \mathbf{b}, \mathbf{b} \rangle > 0$, it is also sometimes referred to as "similarity preserving," in contrast to binding and permutation, which generally creates a dissimilar hypervector. One should keep this distinction in mind when referring to the similarity preserving properties of these operators.

C. Parsing Compound Representations

HD/VSA offers the possibility to encode data structures into compound hypervectors and manipulate the hypervectors with the operations described above to perform computation on the data structures. In conventional computing, data structures are always exposed, and the algorithm queries or modifies individual elements within them. In contrast, the vector operations in HD/VSA can search or transform many or all elements of a data structure in parallel, which we call "computing in superposition" (see Section IV-B). All data structures are hypervectors and can be manipulated immediately and in parallel, regardless of how complicated a structure they possess. However, this also means that the data structure of a compound hypervector is not immediately decodable from the item memory. To query element(s) of a compound hypervector, it first needs to be analyzed or "parsed." We borrow the term parsing from linguistics because the parsing of HD/VSA hypervectors is somewhat similar. To understand a sentence, one needs to divide the sentence into its parts and assign their syntactic roles, which involves comparing the parts with the stored information about their meaning and syntactic roles. Similarly, to extract the result of an HD/VSA computation, one has to parse the resultant hypervector. The parsing of HD/VSA hypervectors involves the decomposition and comparison of the resulting parts with the stored information.

Like with the sum or product of ordinary numbers, the parsing of hypervectors requires additional information, such as the operations used to form the compound representation and the set of seed vectors. Parsing a compound hypervector first entails the operation inverse that is used to encode the wanted element in the data structure. However, the result is almost always approximate because of crosstalk noise coming from all the other elements in the compound hypervector. To single out the correct result, the noisy vector has to be compared to the original seed vectors in terms of similarity. Probing is the process of retrieving the best-matching hypervector (i.e., the nearest neighbor) among the hypervectors for a given query hypervector. This is done in the item memory, which contains all the seed hypervectors. For example, consider the compound hypervector

$$\mathbf{s} = \mathbf{a} \odot \mathbf{b} + \mathbf{c} \odot \mathbf{d}.$$

In order to know which hypervector has been bound to, e.g., $\mathbf{b}$, we have to unbind (inverse binding) $\mathbf{b}$ from $\mathbf{s}$

$$\begin{aligned} \mathbf{s} \odot \mathbf{b} &= \mathbf{b} \odot (\mathbf{a} \odot \mathbf{b} + \mathbf{c} \odot \mathbf{d}) \\ &= \mathbf{a} + \mathbf{b} \odot \mathbf{c} \odot \mathbf{d} = \mathbf{a} + \text{noise} \approx \mathbf{a}. \end{aligned}$$

The resultant hypervector contains the correct answer $\mathbf{a}$ and a crosstalk noise term $\mathbf{b} \odot \mathbf{c} \odot \mathbf{d}$, which is dissimilar to any of the items in the item memory. The query hypervector $\mathbf{a} + \text{noise}$ will be highly similar to the hypervector $\mathbf{a}$ stored in the item memory, which will be successfully retrieved by the nearest neighbor search with a high probability. Thus, the probing operation removes (or cleans up) the noise and returns the correct result.

Cleanup via probing is a critical part of HD/VSA computations and has the advantage that its operation is intrinsically noise resilient, and the degree of noise robustness can be easily controlled by the dimension N. In essence, probing is a signal detection problem. The number of hypervectors that can be correctly retrieved from the superposition is called capacity. The capacity increases roughly linearly with the hypervector dimension and is quite insensitive to the details of a particular HD/VSA model. The signal detection theory for HD/VSA [30] enables one to determine the dimension of the hypervector space that is sufficient for a given computation and a given precision of the hardware.

1) Parsing Hypervectors With Multiple Bindings: In the example above, it was assumed that one argument (i.e., $\mathbf{b}$) of the key-value pair was known. This, however, is not always the case. Moreover, there exist representations where several hypervectors are being bound (e.g., $\mathbf{a} \odot \mathbf{b} \odot \mathbf{c}$). Parsing compound hypervectors with such elements is challenging due to the fact that the binding operation in the MAP model produces a hypervector dissimilar to its arguments (cf. Section III-B4.b). This means that the most obvious way to parse hypervectors of the form $\mathbf{a} \odot \mathbf{b} \odot \mathbf{c}$ is by brute force by checking all possible combinations of the arguments. The number of such combinations, however, grows exponentially with the number of arguments involved. Therefore, a mechanism called a resonator network has been proposed [76], [77], which addresses this problem by a parallel search in the space of all possible combinations.

The resonator network assumes that none of the arguments is given, but that they are contained in different item memories, which should be known to the resonator

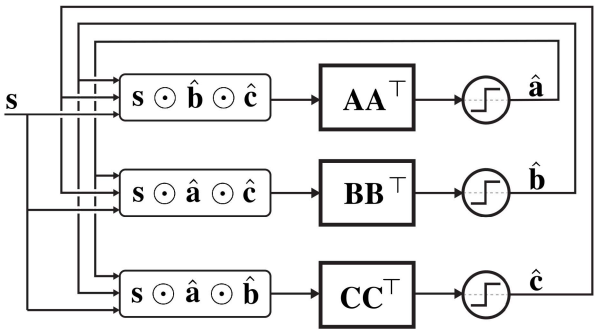


Fig. 2. *Example of a resonator network with three arguments. It is factoring a compound hypervector $\mathbf{s} = \mathbf{a} \odot \mathbf{b} \odot \mathbf{c}$; A, B, and C denote the corresponding item memories containing seed hypervectors for a, b, and c arguments, respectively.*

network. Fig. 2 illustrates an example of a resonator network for factoring the hypervector $\mathbf{s} = \mathbf{a} \odot \mathbf{b} \odot \mathbf{c}$. In a nutshell, the resonator network is a novel recurrent neural network design that uses HD/VSA principles to solve combinatorial optimization problems. As shown in the example, it factors the arguments of the input vector $\mathbf{s}$ representing the binding of several hypervectors. To do so, it uses hypervectors $\hat{\mathbf{a}}(t), \hat{\mathbf{b}}(t), \hat{\mathbf{c}}(t)$, each storing the prediction for a particular argument of the product forming $\mathbf{s}$. Each prediction communicates with the input hypervector ($\mathbf{s}$) and all other predictions using the following dynamics

$$\begin{aligned}
\hat{\mathbf{a}}(t+1) &= \text{sign}\Big(\mathbf{A}\mathbf{A}^\top(\mathbf{s} \odot \hat{\mathbf{b}}(t) \odot \hat{\mathbf{c}}(t))\Big) \\
\hat{\mathbf{b}}(t+1) &= \text{sign}\Big(\mathbf{B}\mathbf{B}^\top(\mathbf{s} \odot \hat{\mathbf{a}}(t) \odot \hat{\mathbf{c}}(t))\Big) \\
\hat{\mathbf{c}}(t+1) &= \text{sign}\Big(\mathbf{C}\mathbf{C}^\top(\mathbf{s} \odot \hat{\mathbf{a}}(t) \odot \hat{\mathbf{b}}(t))\Big)
\end{aligned} \tag{5}$$

where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ denote the corresponding item memories containing $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ arguments, respectively, and $\text{sign}(\cdot)$ denotes a step that projects the predictions back to the bipolar values. Note that the resonator network does not have to work with only bipolar hypervectors. Rather, the usage of the $\text{sign}(\cdot)$ function is determined by the fact that the seed hypervectors in the MAP model are bipolar. Thus, other types of nonlinearity functions can be used to make a resonator network compatible with the desirable format of the seed hypervectors. Note also that these item memories will contain other hypervectors as well, but hypervectors stored in $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ differ from each other. The size of each item memory depends on a task, but it will affect the performance of the resonator network as larger item memories imply a larger search space.

The key insight into the internals of the resonator network is that it iteratively tries to improve its current predictions of the arguments constituting the input hypervector $\mathbf{s}$. In essence, at time t each prediction might hold multiple weighted guesses from the corresponding item memory. The current predictions for other arguments are used to invert the input vector and infer the current argument (e.g., $\mathbf{s} \odot \hat{\mathbf{b}}(t) \odot \hat{\mathbf{c}}(t)$). The cost of using the superposition for storing the predictions is crosstalk noise. To clean up this noise, the predictions are projected back to their item memories (e.g., $\mathbf{A}^\top(\mathbf{s} \odot \hat{\mathbf{b}}(t) \odot \hat{\mathbf{c}}(t))$), which provides weights for different seed hypervectors stored in the item memory and, therefore, constrains the predictions to only to the valid entries in the item memory. These weights are then used to form a new prediction, which is a weighted superposition of all seed hypervectors. Successive iterations of the process in (5) eliminate the noise as the arguments become identified and find their place in the input vector. Once the arguments are fully identified, the resonator network reaches a stable equilibrium, and the arguments can be read. For the sake of space, we do not go into the details of applying resonator networks here. Please refer to [76] for examples of factoring hypervectors of data structures with resonator networks and [77] for their comparison with other standard optimization-based methods.

D. Generality and Utility

Currently, there are several known areas where HD/VSA have been employed. Hypervectors serve as representations for cognitive architectures [37], [38]. They are used for the approximation of conventional data structures [40], [41], [78], distributed systems [79], [80], communications [81], [82], [83], for forming representations in natural language processing applications [31], [84], and robotics [85], [86], [87], [88], [89]. The fact that it is possible to map real-valued data to hypervectors allows one to apply HD/VSA in machine learning domains. Most of these works were connected to classification tasks (see a recent overview in [15]). Examples of domains that have benefited from the application of HD/VSA modeling are biomedical signal processing [34], [90], gesture recognition [33], [91], seizure onset detection and localization [92], physical activity recognition [93], and fault isolation [94]. However, HD/VSA modeling can also be useful for very generic classification tasks [29], [95]. The common feature of these works is a simple training process, which does not require the use of iterative optimization methods, and transparently learns with few training examples.

IV. COMPUTING WITH HD/VSA

A. Computational Primitives Formalized in HD/VSA

In Section III, we have introduced the basic elements of HD/VSA. To provide the algorithmic level in the Marr computing hierarchy in Fig. 1, one needs to combine elements of HD/VSA into primitives of HD/VSA computing, i.e., something akin to design patterns in software engineering. For instance, a set of HD/VSA templates has been proposed for tasks in the domain of personalized

devices covering different multivariate modalities, such as electromyography, electroencephalography, or electrocorticography [34]. Here, we summarize the best practices for representing well-known data structures with HD/VSA—this section can be thought of as a "HD/VSA cookbook." All examples in this section are available in an interactive Jupyter Notebook.[1] After providing some basic rules for representing data structures with HD/VSA, we present a collection of primitives from prior work that has been done along these lines. We do not go into an advanced topic of how distributed representations of data structures can be used to construct or learn single-shot transformations between data structures that share symbols. It is, however, worth noting that this property differentiates distributed representations from conventional data structure manipulations, and the interested readers are referred to, e.g., [96] and [97] for more details. A well-known example of this property has been presented in [98] where a mapping between the "mother-of" relation to the "parent-of" relation was constructed with simple vector operations and using only a few examples. It was shown later in [39] that such a mapping can be used to easily form associations between observed structures and decisions caused by these structures.

It is worth noting that, in this article, we do not cover the representation of real-valued data (see [66], [99], [100], [101], and [102]) or solving machine learning problems (see [15]) as it has been covered elsewhere and is outside the immediate scope of this article.

1) Rules of Thumb: We should point out that the HD/VSA implementations that we describe are not the only possibilities, and other solutions may be possible/desirable in a particular design context. The solutions provided are, however, the most common/obvious choices, based on several "rules of thumb."

1) Superposition is used to combine individual elements of a data structure into a set.
2) Binding is used to make associations between elements, e.g., key-value pairs.
3) Permutation is used for tagging data elements to put them into a sequential order, such as in time series.
4) Permutation is used for protection from the self-inverse property of the binding operation since the hypervector will not cancel out when bound with its permuted version.

We will follow these rules most of the time when forming hypervectors for different data structures.

2) Sets: A set (denoted as S) represents a group of elements, for example, $S = \{a, b, c, d, e\}$. In order to map a set to a hypervector, two steps are required. First, an item memory storing random hypervectors for each element of a set is initialized. We will use bold font in notations of hypervectors (e.g., $\mathbf{a}$ for "a"), but a more general notation is via the mapping function $\phi(i) \mapsto \mathbf{i}, i \in S$. Second, a single hypervector (denoted as $\mathbf{s}$) is formed that represents the set as the superposition of hypervectors for the set's elements, e.g., for the set above

$$\mathbf{s} = \mathbf{a} + \mathbf{b} + \mathbf{c} + \mathbf{d} + \mathbf{e}$$
$$\mathbf{s} = \sum_{i \in S} \phi(i). \tag{6}$$

The hypervector $\mathbf{s}$ is a distributed representation of the set S. This mapping preserves the overlap between elements of the sets. For example, set membership can be tested by calculating the similarity between $\mathbf{s}$ and the hypervector corresponding to the element of interest. If the similarity score is higher than that expected between two random hypervectors, then most likely the element is present in the set. This mapping is very similar to a Bloom filter [103] (in particular, to counting Bloom filter [104]), which is a well-known randomized data structure for approximate membership query in a set. Bloom filters have been recently shown to be a subclass of HD/VSA [78], where the superposition operation is implemented via OR, and seed hypervectors are sparse, as in the SBDRs [56] model. While conceptually representation of sets via distributed representations is a simple idea, it is very influential as it has been applied in myriads of engineering problems (see a survey in [105]).

Note that the limitation of the described mapping of sets is that it does not have a simple and exact way of obtaining distributed representations of the intersection or union of two sets. The exact results can, obviously, be obtained by first parsing distributed representations of the corresponding sets, reconstructing the symbolic versions, computing the union or intersection in the symbolic domain, and, finally, forming the distributed representation of the result. There are, however, simple approximations of the operations that require fewer interactions with the symbolic domain. Both approximations are obtained by the superposition operation on the corresponding set's hypervectors (e.g., $\mathbf{s}_1$ and $\mathbf{s}_2$)

$$\mathbf{s} = \mathbf{s}_1 + \mathbf{s}_2.$$

The difference is in the way the parsing of the result in $\mathbf{s}$ is done. In order to parse the intersection of two sets, only the elements with the largest dot products should be retrieved. Thus, if the result of the intersection is stored in I, which is initially empty ($I = \emptyset$), then, for element i with the corresponding entry $\mathbf{H}_i$ in the item memory

$$I = \begin{cases} I \bigcup \{i\}, & \text{if } \mathbf{H}_i \mathbf{s} \geq \Theta_i \\ I \bigcup \{\emptyset\}, & \text{otherwise} \end{cases}$$

where Θ_i denotes the corresponding threshold.

To retrieve the union ($U = \emptyset$ at the start), the elements with the dot products sufficiently different from the noise

[1] https://github.com/denkle/HDC-VSA_cookbook_tutorial

level should be considered

$$U = \begin{cases} U \bigcup \{i\}, & \text{if } \mathbf{H}_i \mathbf{s} \geq \Theta_n \\ U \bigcup \{\emptyset\}, & \text{otherwise} \end{cases}$$

where Θ_n denotes the noise level threshold. Thus, the subtlety of the intersection is that elements present in both sets will have higher similarity then the ones present in only one of the sets (see Section III-B2). This property of the superposition operation is in fact used in Section IV-A3 for representing multisets.

3) Multisets/Histograms/Frequency Distributions: Let us consider how to form a single hypervector of a multiset or a frequency distribution in the form of counts of the occurrences of various elements in some source. The mapping is essentially the same as in the case of sets in Section IV-A2 with the only difference that a hypervector of an element can be present in the result of the superposition operation several times. For example, given $S = (a, a, a, b, b, c)$, the hypervector representing the frequency of elements is formed as

$$\begin{aligned} \mathbf{s} &= \mathbf{a} + \mathbf{a} + \mathbf{a} + \mathbf{b} + \mathbf{b} + \mathbf{c} \\ &= 3\mathbf{a} + 2\mathbf{b} + \mathbf{c}. \end{aligned}$$

Thus, the number of times a hypervector is present in the superposition determines the frequency of the corresponding element in the sequence. Using $\mathbf{s}$, it is possible to estimate either the frequency of an individual element or compare it to the frequency distribution of another sequence. Both operations require calculating the similarity between $\mathbf{s}$ and the corresponding hypervector.

Usually, $\mathbf{s}$ is used as an approximate representation of the exact counters of a histogram. Fig. 3 demonstrates the Pearson correlation coefficient between the histogram and its approximate version retrieved from a compound hypervector $\mathbf{s}$ where the approximate version was obtained as the dot product between $\mathbf{s}$ and symbols' seed hypervectors. The simulations were done for different sizes of the histogram and varying the dimensionality of hypervectors. The results are characteristic for HD/VSA—the quality of approximation improved with the increased dimensionality of hypervectors.

This mapping shall be seen as a particular instance of a count-min sketch [106] that is a randomized data structure for obtaining frequency distributions from sequences. The count-min sketch is used in a plethora of applications where data are of streaming nature (see some examples in [106]). In Section IV-A6, we will also see that the representation of multisets is an essential primitive for representing n-gram statistics that, in turn, is used for solving classification tasks (see [107], [108], and [109]). The limitation of the presented mapping is that, due to the usage of bipolar hypervectors, the resultant representation could both overcount and undercount the frequency. This limitation is partially addressed by the standard count-min sketch that could only overcount the frequency.

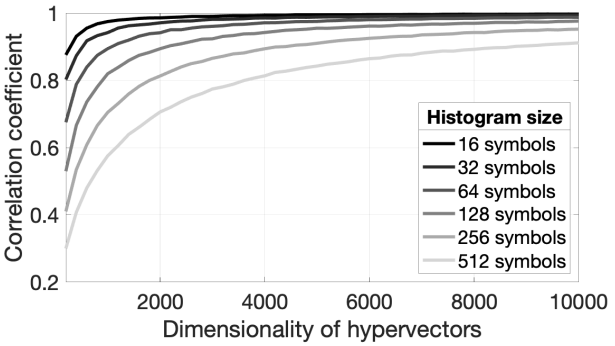


Fig. 3. ***Correlation coefficients between the exact histogram and their approximations from integer-valued $\mathbb{Z}^N$ compound hypervectors. Six different sizes of histograms were considered. The dimensionality of hypervectors varied in the range [200, 10000] with a step of 200. The values of counters were drawn from the discrete uniform distribution [0, 1023]. The reported values were averaged over 100 simulations.***

4) Cross-Product of Two Sets: A particularly interesting case is when we have hypervectors representing two different sets (e.g., $\{a, b, c, d, e\}$ and $\{x, y, z\}$). Then, a mapping based on the binding operation is used to create a hypervector corresponding to the cross-product of two sets as follows:

$$\begin{aligned} &(\mathbf{a} + \mathbf{b} + \mathbf{c} + \mathbf{d} + \mathbf{e}) \odot (\mathbf{x} + \mathbf{y} + \mathbf{z}) \\ &\quad = (\mathbf{a} \odot \mathbf{x} + \mathbf{a} \odot \mathbf{y} + \mathbf{a} \odot \mathbf{z}) + (\mathbf{b} \odot \mathbf{x} + \mathbf{b} \odot \mathbf{y} + \mathbf{b} \odot \mathbf{z}) \\ &\qquad + (\mathbf{c} \odot \mathbf{x} + \mathbf{c} \odot \mathbf{y} + \mathbf{c} \odot \mathbf{z}) + (\mathbf{d} \odot \mathbf{x} + \mathbf{d} \odot \mathbf{y} + \mathbf{d} \odot \mathbf{z}) \\ &\qquad + (\mathbf{e} \odot \mathbf{x} + \mathbf{e} \odot \mathbf{y} + \mathbf{e} \odot \mathbf{z}). \end{aligned}$$

In essence, here occurs (due to the superpositions) a simultaneous binding between all the elements in the two sets. The cross-product set, thus, consists of all possible bindings of hypervectors representing elements of the original sets (e.g., $\mathbf{a} \odot \mathbf{x}$). In the example above, when starting first with the representations of sets, only seven operations (six superpositions and one binding) were necessary to form the representation. The brute force way of forming the cross-product set hypervector would require 29 operations (14 superpositions and 15 bindings). It is clear that this shortcut works due to the fact that the binding operation distributes over the superposition operation (see Section III-B4.b). Note that, using the tensor product variable binding [50] model, the outer product of vector representations of the two sets will be a tensor with the number of dimensions determined by the number of sets in the cross-product. In contrast, the HD/VSA representation of a cross-product is given by a hypervector of the same dimension as the individual set hypervectors. Note also that, while it is simple to form a hypervector

corresponding to the cross-product of two sets with the binding operation, computing the cross-product in the symbolic domain might still require lower computational costs as it does not require high-dimensional representations. Another potential issue of such a representation is the required dimensionality of hypervectors for the situation when all the elements of the cross-product should be retrievable from the distributed representation. In this case, the dimensionality of hypervectors should be proportional to the product of the sets' cardinalities, so even moderately sized sets require a large number of components in hypervectors to provide high accuracy in retrieving individual elements of their cross-product from the corresponding hypervector.

5) Sequences: A sequence is an ordered set of elements. For example, the set from the previous section is now a sequence (a, b, c, d, e), which is not the same as, e.g., (b, a, c, d, e) since the order of elements is different. Note that a finite sequence with k elements is called k-tuple, with an ordered pair being the special case for $k = 2$.

It is clear that plain superposition of hypervectors works for representing sets but not for sequences, as the sequential order would be lost. Many authors have proposed the following idea to represent sequences with permutation, e.g., in [11], [30], [44], [110], [111], and [112]. Before combining the hypervectors of sequence elements, the order i of each element is associated by applying some specific permutation $k - i$ times to its hypervector (e.g., $\rho^2(\mathbf{c})$). The advantage of this recursive encoding of sequences is that extending a sequence can be done by permuting $\mathbf{s}$ and superimposing or binding it (see below) with the next hypervector in the sequence, hence incurring a fixed computational cost per symbol. The last step is to combine the sequence elements into a single hypervector representing the whole sequence.

There are two common ways to combine sequence elements. The first way is to use the superposition operation similar to the case of sets. For the sequence above, the resultant hypervector is

$$\mathbf{s} = \rho^4(\mathbf{a}) + \rho^3(\mathbf{b}) + \rho^2(\mathbf{c}) + \rho^1(\mathbf{d}) + \rho^0(\mathbf{e}).$$

In general, a given sequence S of length k is represented as

$$\mathbf{s} = \sum_{i=1}^{k} \rho^{k-i}(\phi(S_i)) \tag{7}$$

where S_i is the ith element of sequence S. The advantage of the mapping with the superposition operation is that it is possible to estimate the similarity of two sequences by measuring the similarity of their hypervectors. Here, the similarity of sequences is defined by the number of the same elements in the same sequential positions, where the sequences are aligned by their last elements. Evidently, this definition does not take into account the same elements in different positions in contrast to, e.g., an edit distance of sequences [113]. Note that the edit distance can be approximated by vectors of n-gram frequencies and their randomized versions akin to hypervectors (see [114] and [115]).

Another advantage of sequence representation with superposition is that it allows easily probing of the distributed representation $\mathbf{s}$. For example, one can check which element is in position i by applying inverse permutation i times to the resultant hypervector. Note that the permutation of a sequence representation is a general method for shifting an entire sequence by a single operation. It produces a shifted sequence where the ith element is now at the first position, and thus, it can be used to probe the hypervector of element i from the sequence representation. For example, when inverting position 3 in $\mathbf{s}$

$$\begin{aligned}\rho^{-2}(\mathbf{s}) &= \rho^2(\mathbf{a}) + \rho^1(\mathbf{b}) + \rho^0(\mathbf{c}) + \rho^{-1}(\mathbf{d}) + \rho^{-2}(\mathbf{e}) \\ &= \mathbf{c} + \text{noise} \approx \mathbf{c}.\end{aligned}$$

Probing $\rho^{-2}(\mathbf{s})$ with the item memory containing hypervectors of all sequence elements will return $\mathbf{c}$ as the best match (with high probability).

The second way of forming the representation of a sequence involves binding of the permuted hypervectors, e.g., the sequence above is represented as (denoted by $\mathbf{p}$)

$$\mathbf{p} = \rho^4(\mathbf{a}) \odot \rho^3(\mathbf{b}) \odot \rho^2(\mathbf{c}) \odot \rho^1(\mathbf{d}) \odot \rho^0(\mathbf{e}).$$

In general, a given sequence S of length k is represented as

$$\mathbf{p} = \prod_{i=1}^{k} \rho^{k-i}(\phi(S_i)). \tag{8}$$

The advantage of this sequence representation is that it allows forming unique hypervectors even for sequences that differ in only one position. Section IV-A6 provides a concrete example of a task where this advantage is important.

Both mappings allow the replacement of an element at position i in the sequence if the current element at the ith position is known. When the superposition operation is used, the replacement requires subtraction of the permuted hypervector of the current element followed by the superposition of the permuted hypervector of the new element. For example, replacing "d" with "z" in position 4 is done as follows:

$$\mathbf{s} - \rho^1(\mathbf{d}) + \rho^1(\mathbf{z}) = \rho^4(\mathbf{a}) + \rho^3(\mathbf{b}) + \rho^2(\mathbf{c}) + \rho^1(\mathbf{z}) + \rho^0(\mathbf{e}).$$

When the binding operation is used in the mapping, replacement requires the application of the unbinding

operation between the permuted hypervector of the current element and **s**, followed by binding with the permuted hypervector of the new element. For the example above

$$\mathbf{s} \odot \rho^1(\mathbf{d}) \odot \rho^1(\mathbf{z}) = \rho^4(\mathbf{a}) \odot \rho^3(\mathbf{b}) \odot \rho^2(\mathbf{c}) \odot \rho^1(\mathbf{z}) \odot \rho^0(\mathbf{e}).$$

Another feature of both sequence mappings is that the permutation operation distributes over both binding and superposition operations. This means that, in both mappings, the whole sequence can be shifted relative to the initial position by applying the permutation operation required number of times. For example, when applying the permutation operation three times to **s** for (a, b, c, d, e), we obtain

$$\rho^3(\mathbf{s}) = \rho^7(\mathbf{a}) + \rho^6(\mathbf{b}) + \rho^5(\mathbf{c}) + \rho^4(\mathbf{d}) + \rho^3(\mathbf{e}).$$

Thus, $\rho^3(\mathbf{s})$ is the shifted version of the original sequence. This feature can be used for sequence concatenation. For example, to concatenate (a, b, c, d, e) and (x, y, z), we can use already calculated **s** for (a, b, c, d, e) as follows:

$$\begin{aligned}&\rho^3(\mathbf{s}) + \rho^2(\mathbf{x}) + \rho^1(\mathbf{y}) + \rho^0(\mathbf{z})\\ &\quad = \rho^7(\mathbf{a}) + \rho^6(\mathbf{b})\\ &\qquad + \rho^5(\mathbf{c}) + \rho^4(\mathbf{d}) + \rho^3(\mathbf{e}) + \rho^2(\mathbf{x}) + \rho^1(\mathbf{y}) + \rho^0(\mathbf{z}).\end{aligned}$$

This feature was applied in [116] for searching for the best alignment (shift) of two sequences that results in the maximum number of coinciding elements. Other examples of using distributed representation of sequences include modeling human perception of word similarity [115], [117], [118], [119], modeling human working memory [120], [121], [122], [123], [124], [125], DNA string matching [126], and spell checking [118], [127].

An evident limitation of the above mappings is that, due to the usage of a random permutation $\rho()$, elements of the sequence in the nearby positions are dissimilar (even if the elements are the same). A possible way to handle this limitation is by using locality-preserving representations to encode positions; see some proposals in [117], [118], [119], and [128]. Generally, for a given problem, it might be useful to consider alternative representations that bind element and position hypervectors. Another limitation is that the representations of the element's order here used hypervector transformation by the permutation corresponding to its absolute position in a sequence. Thus, the resultant hypervector does not reflect the information about, e.g., successor/predecessor information. Some ways of using relative positions when representing sequences in HD/VSA are investigated in [115].

6) n-Gram Statistics: The n-gram statistics of a sequence S is the histogram of all length n substrings occurring in the sequence. The mapping of n-gram statistics to a single hypervector was presented in, e.g., [84], and includes two steps using the primitives above: first, forming hypervectors of n-grams, and second, forming a hypervector of the frequency distribution. The hypervectors of n-grams are formed as in Section IV-A5 using the chain of binding operations, i.e., each n-gram is treated as an n-tuple. The hypervectors of n-grams and their counters are then used to form a single hypervector for the frequency distribution as in Section IV-A3. Thus, in essence, this is a frequency distribution with compound symbols.

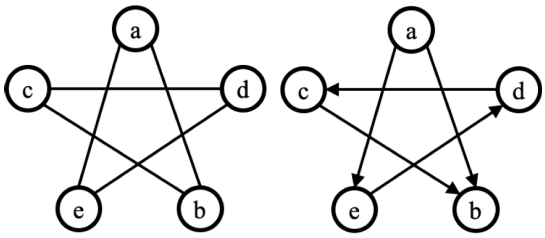


Fig. 4. *Example of undirected and directed graphs with five nodes. In the case of the undirected graph, each node has two edges.*

The advantage of this mapping is that, in order to create a representation for any n-gram, we only need to use a single item memory and several simple operations where the number of operations is proportional to n. In other words, with a fixed amount of resources, the appropriate use of operations allows forming a combinatorially large number of new representations.

The mapping, obviously, inherits the limitations of its intermediate steps. That is, due to the usage of the chain of binding operations (see Section IV-A5), similar n-grams are going to be mapped to dissimilar hypervectors (assuming that all n-grams are assigned with random seed hypervectors). Due to the representation of the frequency distribution (see Section IV-A3), the retrieved values of individual n-grams can be either overcount or undercount.

This mapping has been found useful in several applications: in language identification [84], news article classification [129], and biosignal processing [34] that leveraged its hardware-friendliness [130]. Distributed representations were also used to untie the dimensionality of the hypervector representing n-grams statistics from the possible number of n-grams, which grows exponentially with n and would dictate the size of a localist representation of the n-grams statistics. The same property was also leveraged for constructing more compact neural networks using the distributed representation of n-grams statistics as their input [108], [131], [132].

7) Graphs: A graph (denoted as G) consists of vertices and edges. Edges can either be undirected or directed. Fig. 4 presents examples of both directed and undirected graphs. Following earlier work on graph representations with hypervectors, e.g., in [56], [133], and [134], we consider the following very simple mapping of graphs into hypervectors [133]. A random hypervector is assigned to each vertex of the graph; according to Fig. 4, vertex

hypervectors are denoted by letters (i.e., **a** for vertex "a" and so on). An edge is represented via the binding operation applied to the hypervectors of the connected vertices; for instance, the edge between vertices "a" and "b" is represented as $\mathbf{a} \odot \mathbf{b}$. The whole graph G is represented simply as the superposition of hypervectors representing all edges in the graph, e.g., the undirected graph in Fig. 4 is

$$\mathbf{g} = \mathbf{a} \odot \mathbf{b} + \mathbf{a} \odot \mathbf{e} + \mathbf{b} \odot \mathbf{c} + \mathbf{c} \odot \mathbf{d} + \mathbf{d} \odot \mathbf{e}.$$

Note that, if an edge is represented as the result of the binding of two hypervectors for vertices, it has no information about the direction of the edge, and therefore, the representation above will not work for directed graphs. The direction of an edge can be added applying a permutation to the hypervector of the incidental node; the directed edge from the vertex "a" to "b" in Fig. 4 is represented as $\mathbf{a} \odot \rho(\mathbf{b})$. Note that this is just the mapping of an ordered pair (two-tuple in this case) based on the binding described in Section IV-A5. Thus, the directed graph in Fig. 4 is represented by the hypervector

$$\mathbf{g} = \mathbf{a} \odot \rho(\mathbf{b}) + \mathbf{a} \odot \rho(\mathbf{e}) + \mathbf{c} \odot \rho(\mathbf{b}) + \mathbf{d} \odot \rho(\mathbf{c}) + \mathbf{e} \odot \rho(\mathbf{d}).$$

The described graph representations **g** can be queried for the presence of a particular edge. For graphs that have the same vertex hypervectors, the inner product is a measure of the number of overlapping edges. When it comes to the usage of the described mappings, Gayler and Levy [133] propose an HD-/VSA-based algorithm for graph matching. For two graphs for which the correspondence between their vertices is unknown, graph matching finds the best match between the vertices so that the graph similarity can be assessed. In [135], a similar mapping is applied to the task of inferring missing links of knowledge graphs. The mapping can also be extended to the case when some of its parts are learned from the training data; in [136], representations of knowledge graphs are constructed with hypervectors of nodes and relations that are learned from data.

The described mappings have a number of limitations. First, they do not work for sparse graphs in which vertices can be entirely isolated because those vertices are not represented at all. One way to address it is by also superimposing to **g** the hypervectors representing the vertices or to keep a separate hypervector with the superposition of all the vertices. Another limitation is that one could come up with operations that cannot be done directly on the representation in **g**. One example of such an operation is the computation of composite edges in a directed graph (see details in [137]).

8) Binary Trees: A binary tree is a well-known data structure where each node has at most two children: the left child and the right child. Fig. 5 depicts an example

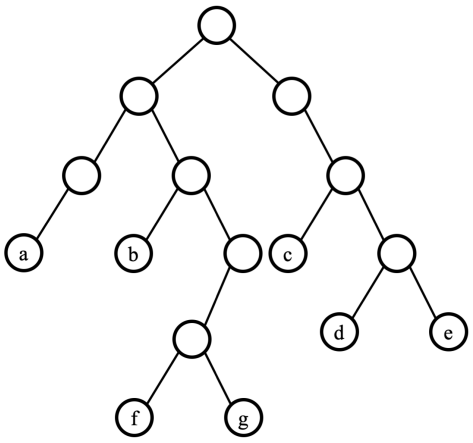


Fig. 5. *Example of a binary tree from [76] where the leaves are different symbols from the alphabet.*

of a binary tree, which will be used to demonstrate the mapping of such a data structure into a single hypervector. We describe a mapping process [76] that involves all three basic HD/VSA operations and two item memories. One item memory stores two random hypervectors corresponding to roles for the left child (denoted as **l**) and the right child (denoted as **r**). Another item memory stores random hypervectors corresponding to symbols of the alphabet, which are associated with the leaves. The example below uses letters so these hypervectors are denoted correspondingly (i.e., **a** for "a" and so on).

The permutation operation is used to create a unique hypervector corresponding to the association of the left or right child with its level in the tree. For example, the left child at the second level is represented as $\rho^2(\mathbf{l})$. In general, the level of the node equals the number of times the permutation operation is applied to its role hypervector.

The chain of the binding operations is used to create a hypervector corresponding to the trace from the tree root to a certain leaf associated with the leaf's symbol. For instance, to reach the leaf "a," it is necessary to traverse three left children. In terms of HD/VSA, this trace will be represented as follows: $\mathbf{a} \odot \mathbf{l} \odot \rho(\mathbf{l}) \odot \rho^2(\mathbf{l})$. In such a way, traces of all leaves can be represented.

Finally, the superposition operation is used to combine hypervectors of individual traces in order to create a single hypervector (denoted as **t**) corresponding to the whole binary tree. Combining all steps together, the single hypervector for the tree depicted in Fig. 5 will then look like

$$\begin{aligned}\mathbf{t} = {} & \mathbf{a} \odot \mathbf{l} \odot \rho(\mathbf{l}) \odot \rho^2(\mathbf{l}) \\ & + \mathbf{b} \odot \mathbf{l} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{l}) \\ & + \mathbf{c} \odot \mathbf{r} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{l}) \\ & + \mathbf{d} \odot \mathbf{r} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{r}) \odot \rho^3(\mathbf{l}) \\ & + \mathbf{e} \odot \mathbf{r} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{r}) \odot \rho^3(\mathbf{r}) \\ & + \mathbf{f} \odot \mathbf{l} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{r}) \odot \rho^3(\mathbf{l}) \odot \rho^4(\mathbf{l}) \\ & + \mathbf{g} \odot \mathbf{l} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{r}) \odot \rho^3(\mathbf{l}) \odot \rho^4(\mathbf{r}).\end{aligned}$$

Thus, the information about the tree in Fig. 5 is stored in a distributed way in the compound hypervector $\mathbf{t}$, which, in turn, can be queried with HD/VSA operations. For example, given a trace of children, we can extract the symbol associated with the leaf at this trace. Assume that the trace is right-right-left; then, its hypervector is $\mathbf{r} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{l})$. This hypervector can be unbound from $\mathbf{t}$ as

$$\mathbf{t} \odot (\mathbf{r} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{l})) = \mathbf{c} + \mathtt{noise}.$$

The result is $\mathbf{c} + \mathtt{noise}$ because $\mathbf{r} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{l})$ cancels out itself in $\mathbf{t}$ and, thus, releases $\mathbf{c}$, which was bound with this trace. Since there were other terms in the superposition $\mathbf{t}$, they act as crosstalk noise for $\mathbf{c}$, hence denoted as $\mathtt{noise}$. Thus, when $\mathbf{c} + \mathtt{noise}$ is presented to the item memory, the item memory is expected to return $\mathbf{c}$ as the closest alternative with high probability. The inverse task of querying the trace with a given leaf symbol is more challenging because the resultant hypervector corresponds to a chain of binding operations, e.g., for $\mathbf{c}$, we get

$$\mathbf{t} \odot \mathbf{c} = \mathbf{r} \odot \rho(\mathbf{r}) \odot \rho^2(\mathbf{l}) + \mathtt{noise}.$$

In order to interpret the resultant hypervector, one has to query all hypervectors corresponding to all possible traces in a binary tree of the given depth, where the number of traces grows exponentially with the depth of the tree. This is a significant limitation of the representation. This limitation can, however, be addressed in part by the resonator network [76], [77] (see Section III-C).

We do not cover the details of factoring trees with the resonator network here, but the interested readers are referred to [76, Sec. 4.1]. It should, of course, be noted that resonator networks are not limitless in their capabilities, since as reported in [77], for the fixed dimensionality of hypervectors their capacity decreases with the increased number of factors (i.e., tree depth in this case). Nevertheless, they still seem to be the best alternative to tackle the problem (cf. [77, Fig. 3])—their search space scales quadratically with N.

The presented mapping is, of course, not the only possible way to represent binary trees. For example, in [44], it was proposed to use two different random permutations for representing nested structures. This mechanism can be applied to trees as well by using these different random permutations instead of $\mathbf{l}$ and $\mathbf{r}$.

Last but not least, note that the mapping for binary trees can be easily generalized to trees with nodes having more than two children by superimposing additional role hypervectors in the item memory. Also, filler hypervectors for the leaves do not have to be seed hypervectors—they could represent any compound structure.

9) Stacks: A stack is a memory in which elements are written or removed in a last-in-first-out manner. At any given moment, only the top-most element of the stack can be accessed, and elements written to the stack before are inaccessible until all later elements are removed. There are two possible operations on the stack: writing (pushing) and removing (popping) an element. The writing operation adds an element to the stack—it becomes the top-most one, while all previously written elements are "pushed down." The removing operation allows reading the top-most element of the stack. Once read, it is removed from the stack, and the remaining elements are moved up.

HD-/VSA-based representations of a stack were proposed in [41] and [138]. The representation of a stack is essentially the representation of a sequence with the addition of an operation that always moves the top-most element to the beginning of the sequence. For example, if "*d*," "*c*," and "*b*" were successively added to the stack, then the hypervector for the current state of the stack is

$$\mathbf{s} = \mathbf{b} + \rho(\mathbf{c}) + \rho^2(\mathbf{d}).$$

Thus, the pushing operation is implemented as the concatenation of two sequences (i.e., a new element to be written and the current state of the stack) using their corresponding hypervectors (see Section IV-A5). In particular, the hypervector of the newly written element is added to the permuted hypervector of the current state of the stack. For instance, writing "*a*" to the current state $\mathbf{s}$ is done as follows:

$$\mathbf{s} = \mathbf{a} + \rho(\mathbf{s}) = \mathbf{a} + \rho(\mathbf{b}) + \rho^2(\mathbf{c}) + \rho^3(\mathbf{d}).$$

The popping operation includes two steps. First, $\mathbf{s}$ is probed with the item memory of elements' hypervectors in order to get the closest match for the seed hypervector of the top-most element. Once the hypervector of the top-most element is identified (e.g., $\mathbf{a}$ in the current example), it is removed from the stack, and the hypervector representation of the stack with the remaining elements is moved back by the permutation operation

$$\begin{aligned}\rho^{-1}(\mathbf{s} - \mathbf{a}) &= \rho^{-1}(\rho(\mathbf{b}) + \rho^2(\mathbf{c}) + \rho^3(\mathbf{d})) \\ &= \mathbf{b} + \rho(\mathbf{c}) + \rho^2(\mathbf{d}).\end{aligned}$$

When it comes to the limitations of this representation, there are several things to keep in mind. First, the popping operation will not work well if the hypervector representing the stack is normalized after each writing operation, so the operations described above assume that $\mathbf{s}$ is not normalized. Second, the size of the stack that can be retrieved reliably from $\mathbf{s}$ depends on the dimensionality of $\mathbf{s}$. Third, if the alphabet of symbols that can be stored in the stack is large, then the probing process for the popping operation might be a computationally demanding step. Fourth, if the

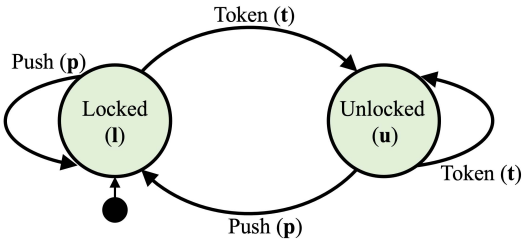


Fig. 6. *Example of a state diagram of a finite-state automaton modeling the control logic of a turnstile. It has two states. The start state is depicted by the arrow pointing from the black circle.*

stack is going to store compound hypervectors, then the popping operation would be more complicated as it either would require the item memory storing all compound hypervectors (this option quickly expands the item memory) or would need to incorporate a retrieval procedure assuming the knowledge of the structure of the compound hypervectors so that they could be parsed.

The main foreseen application of the presented representation is within some control structures as a part of HD/VSA systems. For example, it was used in [41] in a proposal for implementing stack machines and in [138] as a part of HD/VSA implementation of a general-purpose left-corner parsing with simple grammars.

10) Finite-State Automata: A deterministic finite-state automaton is an abstract computational model; it is specified by defining a finite set of states, a finite set of allowed input symbols, a transition function, the start state, and a finite set of accepting states. The automaton is always in one of its possible states. The current state can change in response to an input. The current state and input symbol together uniquely determine the next state of the automaton. Changing from one state to another is called a transition. The transition function defines all transitions in the automaton.

Fig. 6 presents an intuitive example of a finite-state automaton, the control logic of a turnstile. The set of states is {"Locked," "Unlocked"}, and the set of input symbols is {"Push," "Token"}. The transition function can be easily derived from the state diagram in Fig. 6.

HD-/VSA-based implementations of finite-state automata were proposed in [40] and [41]. Similar to binary trees, the mapping involves all three HD/VSA operations and requires two item memories. One item memory stores seed hypervectors corresponding to the set of states (denoted as $\mathbf{l}$ for "Locked" and $\mathbf{u}$ for "Unlocked"). Another item memory stores seed hypervectors corresponding to the set of input symbols (denoted as $\mathbf{p}$ for "Push" and $\mathbf{t}$ for "Token"). The hypervectors from the item memories are used to form a single hypervector (denoted as $\mathbf{a}$), which represents the transition function. Note that the state diagram of a finite-state automaton is essentially a directed graph in which each edge has an input symbol associated with it. Therefore, the mapping for the transition function is very similar to the mapping of the directed graph in Section IV-A7. The only difference is that the binding of the hypervectors for the vertices (i.e., states) involves, as an additional factor, the hypervector for the input symbol, which causes the transition. For example, the transition from "Locked" state to "Unlocked" state, contingent on receiving "Token," is represented as

$$\mathbf{t} \odot \mathbf{l} \odot \rho(\mathbf{u}).$$

Given the distributed representations of all transitions of the automaton, the transition function $\mathbf{a}$ of the automaton is represented by the superposition of the individual transitions

$$\mathbf{a} = \mathbf{p} \odot \mathbf{l} \odot \rho(\mathbf{l}) + \mathbf{t} \odot \mathbf{l} \odot \rho(\mathbf{u}) + \mathbf{p} \odot \mathbf{u} \odot \rho(\mathbf{l}) + \mathbf{t} \odot \mathbf{u} \odot \rho(\mathbf{g}).$$

In order to calculate the next state, we query $\mathbf{a}$ with the binding of the hypervectors of the current state and input symbol followed by the inverse permutation operation applied to the result. Calculated in this way, the result is the noisy version of the hypervector representing the next state. For example, if the current state is $\mathbf{l}$ and the input symbol is $\mathbf{p}$, then we have

$$\rho^{-1}(\mathbf{a} \odot \mathbf{p} \odot \mathbf{l}) = \mathbf{l} + \mathtt{noise}.$$

As usual, this hypervector should be passed to the item memory in order to retrieve the noiseless seed hypervector $\mathbf{l}$.

The same mapping can be used to create a hypervector representing a nondeterministic finite-state automaton [139]. The main difference between deterministic finite-state automata is that the nondeterministic finite-state automaton can reside simultaneously in several of its states. The transitions do not have to be uniquely determined by their current state and input symbol, i.e., there can be several valid transitions from a given current state and input symbol. The nondeterministic finite-state automaton can assume a so-called generalized state, defined as a set of the automaton's states that are simultaneously active. The generalized state corresponds to a hypervector representing the set of the currently active states with (6). Similar to the deterministic finite-state automata, the hypervector for the generalized state is used to query the automaton to get a hypervector for the next generalized state. This corresponds to parallel execution of the automaton from all currently active states. It should also be noted that, in the case of the nondeterministic finite-state automaton, due to the potential presence of several active states, the cleanup procedure (see Section III-C) has to search for several nearest neighbors. Please see Section IV-B2 for an example of such a procedure.

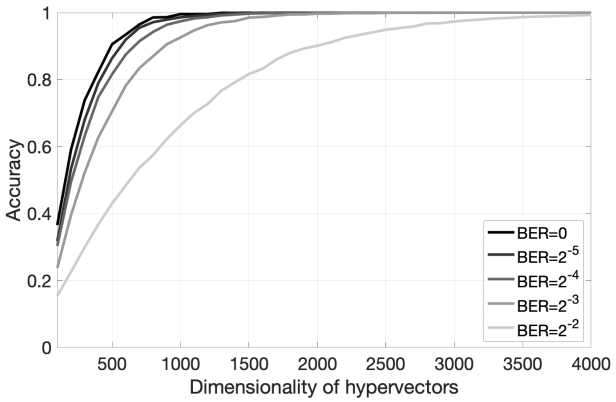


Fig. 7. *Average accuracy of the recall of the next state of the automaton from a, which was bipolarized, against the dimensionality of hypervectors ($N \in [100, 4000]$ with a step of 100). The results were obtained from over 50 random initializations of the item memories. For each initialization, 1000 transitions (chosen randomly) were performed. For each transition function, noise added to a was also generated at random. BERs were in the range of 0.0312–0.2500; BER is defined as the percentage of bits (here, dimensions) that have errors relative to the total number of bits.*

In Section IV-B2, we will see an example of how to compute with hypervectors representing automata, but the most obvious application of the presented representation is to execute the automaton in the presence of noise in hypervectors. Fig. 7 presents the accuracy of the correct recall of the next state from a bipolarized hypervector representing an automaton with 22 states and 29 symbols. The figure shows how the accuracy changed with the dimensionality of hypervectors for different values of noise in **a**. As expected, we see that, for every amount of noise, there is eventually a dimensionality that allows a perfect recall—an elegant property that can be simply leveraged for executing a deterministic behavior in a very stochastic environment.

While, currently, there are not many HD/VSA applications that use finite-state automata (but we will see one in Section IV-B2), there is potential in such a mapping as it naturally allows using HD/VSA as a medium for executing programs that can be formalized via automata. Moreover, the primitives for stacks and finite-state automata can be combined to create richer computational models, such as deterministic pushdown automata or stack machines; see [41] for a sketch of a stack machine operating with hypervectors. An alternative representation for pushdown automata and context-free grammars has been recently presented in [42].

Finally, it should be noted that the presented mapping is designed for executing an automaton; however, it is limited in the sense that it cannot be used directly to modify it or to perform composition operations (e.g., combining it with another automaton).

11) Deeper Hierarchies: Finally, it is important to touch upon constructing data structures encoding deep hierarchies. In Sections IV-A2–IV-A10, we concentrated mainly on data structures with a single-level hierarchy. In fact, this is what most of the current studies in the area used. Therefore, we will not go into technical details of existing proposals. HD/VSA, however, is well-suited for representing many levels of hierarchy, and the representation of hierarchical data structures was a part of the original motivation right from the start (see [51]). The representation of binary trees in Section IV-A8 can already be seen as a hierarchy since a tree has several levels and the representation should be able to discriminate between different levels. In the presented mapping, this was done using powers of permutation to protect different levels of hierarchy. This can be done in some other ways by, e.g., assigning special role hypervectors for each level. Currently, the usage of representations for hierarchies in HD/VSA is relatively uncommon. We mainly attribute this fact to the nature of applications that are being explored, rather than to the capabilities of HD/VSA. The use cases, which relied on the representation of the hierarchical representations, are the representation of analogical episodes [36], [53], distributed orchestration of workflows [79], and representation of hierarchies in WordNet concepts [140]. It has also been argued that the representation of hierarchical data structures via HD/VSA is an important feature for modular learning where modules at different levels of hierarchy can communicate with such representations [37], [141]. Finally, there is a recent proposal that suggests that the JSON format with several levels of hierarchy can be represented in hypervectors [142].

B. Computing in Superposition With HD/VSA

1) Simple Examples of Computing in Superposition: A well-known data structure—Bloom filter [103]—is the simplest case of computing in superposition. The Bloom filter is a sketch as a fixed-size memory footprint is used to represent a set of elements. A Bloom filter encodes a set as a superposition of its elements' sparse binary vectors, which, in essence, corresponds in HD/VSA to a compound hypervector representing sets. Thus, the Bloom filter directly corresponds to the primitive for representing sets, as described in Section IV-A2. With Bloom filters, the algorithm for searching an element in a set is a single operation of comparing the similarity of the distributed representation of the query element to the Bloom filter instance. In other words, all elements of the set are tested in one shot, i.e., the search is performed as a computation in superposition. It enables solving the approximate membership query task instantaneously. This illustrates a simple instance of computing in superposition. Bloom filters are highly specialized for one particular task. In contrast, HD/VSA constitutes a broad framework for computing in superposition, containing Bloom filters as a subclass [78]. We have already seen other examples in Section IV-A for computing in superposition with HD/VSA, such as the primitives for recursive construction of sequence representations [see (7) and (8)] and, in Section IV-A4, the

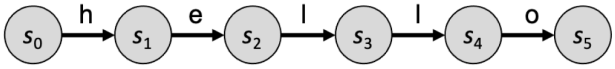


Fig. 8. *Automaton for the base string "hello."*

forming of a representation for the cross-product of two sets via a single binding operation. In these examples, the distributivity of HD/VSA operations (see Section III-B4) played an important role.

2) Computing in Superposition for Substring Search: Finding a substring within a larger string is a standard computer science problem with numerous algorithms (e.g., [143], [144], and [145]) that have a linear complexity on the total length of the base and the query strings. Recently, an algorithm based on nondeterministic finite-state automata was formulated with HD/VSA [146]. It nicely demonstrates how HD/VSA can solve computer science problems, so we briefly explain it here.

Each position of a symbol in the base string is modeled as a unique state of the nondeterministic finite-state automaton $S = \{s_0, s_1, s_2, \ldots, s_n\}$. For example, the string "hello" generates an automaton with six states: s_0 through s_5. The transitions between states are defined by the base string's (denoted B) symbols b_i from $B = \{b_1, b_2, \ldots, b_n\}$. Fig. 8 illustrates the automaton for the string "hello." The nondeterministic finite-state automaton is then defined by tuple $< S, s_0, B, T >$, where s_0 is the start state of the automaton and T is the set of transition tuples of the form $t_i = < s_{i-1}, b_i, s_i >$, where s_{i-1} and s_i are the start and end states of a particular transition caused by symbol b_i. The elements of sets B and S are represented by i.i.d. random hypervectors (denoted in bold). The hypervector $\boldsymbol{\beta}$ of the automaton for the base string is constructed as (cf. Section IV-A10)

$$\boldsymbol{\beta} = \sum_{i=1}^{|B|} \mathbf{s}_{i-1} \odot \mathbf{b}_i \odot \rho^1(\mathbf{s}_i). \tag{9}$$

Thus, $\boldsymbol{\beta}$ is the superposition of all the automaton's transitions caused by sequential input of symbols of the base string. Note that this representation corresponds to the primitive for the finite-state automata, as described in Section IV-A10.

The algorithm for finding whether a query string $Q = \{q_1, \ldots, q_l\}$ is a part of the base string B is a sequential retrieval of the next state of automaton $\boldsymbol{\beta}$ for each symbol of the query string q_j. In terms of hypervectors, this is

$$\mathbf{p}_j = \rho^{-1}(\mathbf{p}_{j-1} \odot \boldsymbol{\beta} \odot \mathbf{q}_j) \tag{10}$$

where $\mathbf{p}_j$ denotes the hypervector that includes the hypervector(s) of the next generalized automaton state (given symbol q_j), as well as crosstalk noise. Equation (10) is also a primitive from Section IV-A10. Note, however, that the generalized state may include one or several states s_i. The set of valid (i.e., permitted) previous generalized states is initialized as $\mathbf{p}_0 = \sum_{s_i \in S} \mathbf{s}_i$, which is a superposition of all the states of the base string. Since the operation in (10) is performed on the superimposed set of all states, it is qualified as computing in superposition.

While the algorithm presented in [146] works in principle (confirmed experimentally but not reported here), the required dimensionality of hypervectors grows extremely fast with the length of strings since every step of (10) introduces additional crosstalk noise to $\mathbf{p}_j$. Crosstalk noise can be reduced by a cleanup procedure on $\mathbf{p}_j$ after every execution of (10)

$$\mathbf{p}_j = \mathbf{S}\mathbf{S}^\top \mathbf{p}_j \tag{11}$$

where $\mathbf{S} \in [N, n+1]$ denotes the item memory storing hypervectors for the unique states of the base string, $S = \{s_0, s_1, s_2, \ldots, s_n\}$. This primitive uses the idea of projecting predictions back onto the item memory, and it was introduced in Section III-C as a part of the resonator network [see (5)].

We simulated the modified algorithm for searching a fixed length query substring (30 symbols) in the base string of four different lengths (see Fig. 9). The average accuracy in 30 simulation runs is plotted against the varying dimensionality of hypervectors. In every simulation run, 100 different random base strings were used. In approximately half of the searches, the query substring was present in the base string, so a single simulation run determines the accuracy of correctly detecting when a substring is present and when it is not (thus, the accuracy of a random guess is 0.5). With increasing dimensionality of hypervectors, the

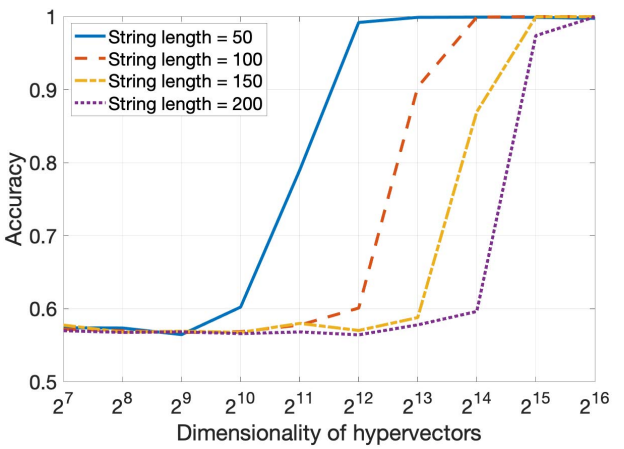


Fig. 9. *Search of a substring in superposition with HD/VSA using the modified algorithm from [146]. The length of a substring was fixed to 30. The reported values were averaged over 30 simulations.*

accuracy of detecting a substring increases and eventually approaches 1. For longer base strings, it would require larger dimensions of the hypervectors to achieve high accuracy. Nevertheless, it scales much better than the original algorithm for which we were not able to simulate large enough dimensionalities that would provide reasonable accuracy.

The substring search provides lessons for computing in superposition with HD/VSA. Both algorithms use it; the original one requires a large dimensionality to reduce crosstalk sufficiently, while the modified one includes an extra cleanup step to reduce the required dimensionality significantly—but it also increases the algorithmic complexity. In particular, the asymptotic computational complexity of the query algorithm in HD/VSA operations is $O(|Q|)$ for the original algorithm versus ($O(|Q||B|)$ for the modified algorithm. However, in terms of hypervector dimensionality, the original algorithm required much more space than the modified algorithm. Another consequence of long hypervectors required by the original algorithm is that, despite not requiring an extra cleanup step (11), the total number of operations would be higher due to much shorter hypervectors used by the modified algorithm. Moreover, with appropriate implementation of the HD/VSA algorithm on parallel hardware, the cleanup step in (11) can be parallelized[2] using, e.g., in-memory computing architectures with massive phase-change memory devices [147]. When executed on such hardware, the time complexity of the modified algorithm also becomes $O(|Q|)$.[3] Thus, computing in superposition in HD/VSA is natural but can require very high dimensionality for managing crosstalk. Steps to manage the crosstalk can be added to the algorithm at no compute time costs if the algorithm is properly mapped on parallel hardware (see [126] for the acceleration of DNA string matching with HD/VSA).

Last, it is important to note that we do not claim that the substring search will be a practically useful application of computing in superposition since its computational complexity exceeds that of the conventional algorithms optimized for the problem. However, we think that this example has a didactic value as it clearly demonstrates how the primitives for representing data structures from Section IV-A can be connected to a well-known computer science problem. Thus, it serves as an important illustration of the lines along which one should think to utilize computing in superposition. In the following, we elaborate on more practical (but not always explicit) contemporary examples of using computing in superposition.

3) Applications of Computing in Superposition: In the long term, we anticipate the resonator networks [76], [77] (see Section III-C) to become a pivotal mechanism in many solutions based on computing in superposition since they use the idea of removing crosstalk noise from the predictions represented in the superposition. In particular, we believe that this idea would be important to efficiently solve nontrivial combinatorial search problems. There are already a couple of proposals for, e.g., scene decomposition [148] and prime factorization [149], but they are yet to be demonstrated at scale.

In a short term, there is another practical direction for the application of computing in superposition that is already being used to tackle a large problem—enhancement of capabilities of machine learning algorithms (often neural networks).[4] In the following, we briefly explain the role computing in superposition plays in approaches proposed within this direction since, in our opinion, it is a unifying theme that will, hopefully, inspire more approaches for machine learning algorithms enhancement.

A recent connection, introduced in [102] and [150], between a method for representation of numeric data as hypervectors [51], [67], [68] and kernel methods allowed representing functions as compound hypervectors of weighted sets (see Section IV-A2). This finding, in turn, allowed a one-shot evaluation of kernel machines since the whole model can now be stored in the superposition as a compound hypervector. The one-shot evaluation principle was demonstrated on probability density estimation [102], [150], [151], kernel regression [102], [150], Gaussian processes-based mutual information exploration [152], and rules search in superposition [153]. The distributed representations of numeric data can also be very useful even without formal links to the kernel methods. They can be used to store in superposition multiple locations of interest on a 2-D grid that has been shown to be important for, e.g., implementing agent's memory for cognitive maps [154], navigation in 2-D environments [67], [154], and reasoning on 2-D images [67], [148], [155].

When it comes to approaches for augmenting neural networks, in [156], the weights of multiple deep neural networks trained to solve different tasks were stored jointly in superposition using a single compound hypervector. This approach addressed the so-called "catastrophic forgetting" phenomenon by using a unique random permutation assigned to each task that allows networks to co-exist in the compound hypervector without much interference. These permutations were used as keys to extract the corresponding network's weights from the superposition hypervector. A big leap of such an approach is that new networks can be added gradually into the superposition hypervector without significant degradation of the performance of the previously included networks.

Another approach combining computing in superposition and neural networks was presented in [157]. There,

[2]For the sake of fairness, it should be noted that the conventional substring search algorithms could also be parallelized.

[3]Of course, the size of the chip places limitations on the dimensionality of hypervectors and the number of hypervectors in the item memory.

[4]We additionally review some of these works in the context of connections to hardware realizations of HD/VSA in Section V-B2.

activations of the network's layers from a single data sample were used in place of value hypervectors. They were bound to the corresponding random key hypervectors, and all hypervectors of the key-value pairs were aggregated in a single compound hypervector. Since the compound hypervector simultaneously keeps all the activations, calculating the similarity between two such hypervector corresponds to an aggregate similarity score between two data samples. This property was leveraged successfully to detect out-of-distribution data. In a similar way, in [158] and [159], activations of multiple neural network-based image descriptors were combined together into a compound hypervector simultaneously representing the aggregated descriptor. Such hypervectors allowed an efficient image retrieval for visual place recognition tasks. A different combination of a neural network and a compound hypervector of the key-value pairs was reported in [160], where the compound hypervector was used to simultaneously represent the output of a neural network when solving multilabel classification tasks.

From the descriptions above, one can notice a striking pattern—most of the approaches relied on the primitive for representing sets, in general, and sets of key-value pairs, in particular. This is likely because the latter is a simple yet nontrivial data structure. We, thus, anticipate that more new approaches can be conceived by expanding to more sophisticated data structures.

V. HARDWARE REALIZATIONS OF HD/VSA

A. HD/VSA Models for Different Types of Hardware

The computational primitives of HD/VSA connect the algorithmic level of Marr's computing hierarchy (see Fig. 1) to the computational level. At the same time, an HD/VSA placed at the algorithmic level also interfaces with the implementation level. While the computational primitives are generic across different HD/VSA models, the model choice can become critical when it comes to interfacing with a particular physical substrate.[5] This suggests a general design pattern when designing an HD/VSA system to be implemented on emerging hardware: the desired computation is formalized in terms of the generic HD/VSA computational primitives, and then, the specific HD/VSA variant best suited for this emerging hardware is used in implementing these primitives. Here, we describe some of the existing HD/VSA models and examples of how they can be implemented in different hardware. Different HD/VSA models can be distinguished in terms of the properties of seed hypervectors and corresponding algebraic operations.

1) Dense Binary Vectors: The binary spatter codes [55] model uses dense binary vectors. Superposition is done by the componentwise majority rule followed by tie-breaking, and binding is by the componentwise XOR. Due to its discrete nature, binary spatter code is highly suitable for conventional digital application-specific integrated circuits (ASICs). The first ASIC design [130] was made in 65-nm CMOS for language recognition, followed by more programmable designs in 28 nm [161] and 22 nm [162]. It has been also mapped on a 28-nm FD-SOI silicon prototype with four programmable OpenRISC cores operating in a near-threshold regime (0.7–0.5 V) [163]. Overall, in the binary spatter codes model, the hypervectors are stationary and robust, and related binary operations are local and simple. This provides a natural fit for implementing the model on non-von Neumann architectures (a.k.a. in-memory computing) using emerging technologies, such as carbon nanotube FETs and resistive RAM [26], [164], [165], and phase-change memory [16], [147]. Specifically, Karunaratne *et al.* [16] describe how to organize computational memories for storing and manipulating hypervectors, whereby the operations are implemented inside, or near, computational memory elements.

2) Integer Vectors: The MAP model [54], the HD/VSA model that we have used in the examples so far as the default, employs bipolar ($+1$s and -1s) hypervectors, componentwise multiplication, and superposition with possible thresholding. The MAP model will usually suit the same technologies as binary spatter codes. For example, it was recently implemented on an FPGA for hand gesture recognition [166].

3) Real-Valued Vectors: The HRR model [52] was originally done with N-dimensional real-valued hypervectors whose components are i.i.d. normal with zero mean and $1/N$ variance. Superposition is done by the normalized vector sum, and binding is done by circular convolution. It has been shown how to map real-valued hypervectors onto spiking neurons using the principles from the neural engineering framework [167] with the help of spike-rate coding. For example, the Spaun cognitive architecture [38] has been implemented in such a way. Most of the studies were done using simulations in Nengo [168], which is a Python-based package for simulating large-scale spiking neural networks. Nevertheless, Nengo has compilers for popular neuromorphic platforms, such as SpiNNaker and Loihi; therefore, it is straightforward to deploy a model built in Nengo on the neuromorphic platforms.

4) Complex Vectors: In the Fourier HRRs [53], vector components are random phasors, superposition is by componentwise complex addition followed by normalization, and binding is by componentwise complex multiplication

[5]It should be noted that there exist subtleties when it comes to computational primitives of different HD/VSA models (see [62] for a discussion). Thus, strictly speaking, the model choice may not be only influenced by a physical substrate but also by the nature of the task at the computational level. To put it simply, not all HD/VSA models are interchangeable. This is not entirely unexpected since, if a framework can provide tight matches between computation and hardware to enable efficiency, the separation between abstraction and physical realization cannot be perfect. Thus, for the sake of narration in this section, we focus on the availability of an efficient mapping between some physical substrate and some HD/VSA model.

(addition of phasors) [53]. This HD/VSA model should be suited for implementations on coupled oscillator hardware [169]; however, we are not aware of any concrete hardware realizations as of yet. Another alternative is mapping complex HD/VSA to the neuromorphic hardware [24] by representing phasors with spike times [170]. This implementation is particularly interesting because the neuromorphic hardware scales up more easily than the current approaches to coupled oscillator hardware. However, no neuromorphic implementation of a full complex HD/VSA has been reported to date.

5) Sparse Vectors: Traditional HD/VSA models use densely distributed representations. However, sparsity is an important ingredient of energy-efficient realizations in hardware. Thus, HD/VSA models that use sparse representations are important for mapping HD/VSA operations efficiently onto hardware. We are aware of two such models: SBDRs [56], [57] and sparse block codes [58], [59]. In the SBDRs model, the hypervectors are sparse patterns without any restrictions on placing the active components, while, in sparse block codes, the hypervectors are divided into blocks of the same size (denoted as K) with just one single active component per block. The SBDRs model was implemented around 1990 in specialized hardware—"associative-projective neurocomputers" [49]. This hardware was designed to operate efficiently with sparse representations [56] by using simple bitwise logical operations and a long word processor with 256 bits (later with 512 and 2048 bits, implemented by Wacom, Japan). For cleanup memory, it used Willshaw-like associative memories, following earlier ideas to implement such memory networks [171] and motivated by theoretical results suggesting high memory capacity [70], [71], [172], [173], [174], [175]. Concerning HD/VSA with sparse block codes, in particular with complex-valued sparse vectors, they seem to be the most amenable for implementations on neuromorphic and coupled oscillator hardware. Currently, there are two proposals for implementing binary sparse block codes in spiking neural network circuits [17], [18]. The proposal in [17] has been implemented on Intel's Loihi [24], while the one from [18] has not been realized in hardware yet, but it has been implemented in the Brian 2 simulator [176].

B. Mapping Algorithms to Hardware

1) Hardware Implementations of Pure HD/VSA: How do implementations of HD/VSA in existing conventional hardware produce gains over conventional machine learning methods? On a dedicated digital ASIC design, it has been demonstrated that HD-/VSA-based classification can lower the energy by about 2× compared to a k-nearest neighbors classifier for the European language recognition task [130]. By running these classifiers on the Nvidia Tegra X2 GPU, HD/VSA exhibited over 3× lower energy per prediction [161]. Considering a wide range of biomedical signal classifications, HD/VSA achieved at least the same level of accuracy compared to the baseline methods running on the conventional programmable hardware, however, at 2× lower power compared to the fixed-point SVM for EMG classification on the embedded ARM Cortex M4 [163], 2.9× lower energy compared to SVM, and over 16× compared to CNN and LSTM for iEEG classification on the Nvidia Tegra X2 [107]. More details for this benchmarking are available in [34]. Using the PageRank centrality metric, HD/VSA achieved comparable accuracy with 2× faster inference compared to the graph kernels and neural networks for graph classifications on the Intel Xeon CPUs [177]. These improvements are due to the fact that the HD-/VSA-based solutions mostly use basic bitwise operations, instead of fixed- or floating-point operations.

Another appealing property of HD-/VSA-based solutions is their great robustness, for example, they tolerate 8.8× higher probability of failures with respect to intermittent hardware errors [130] and 60× higher probability of failures with respect to permanent hardware errors [26]. This robustness makes HD/VSA ideally suited to the low signal-to-noise ratio and high variability conditions in the emerging hardware, as discussed in more detail in [43]. Among them, as a large-scale experimental demonstration [16] of HD/VSA, it was implemented on 760 000 phase-change memory devices performing analog in-memory computing with 10 000-dimensional binary hypervectors for three different classification tasks. The implementation not only achieved accuracies comparable to software implementations—despite the nonidealities in the phase-change memory devices—but also achieved over 6× end-to-end energy saving compared to an optimized digital ASIC implementation [16].

The connection of HD/VSA to spiking neuromorphic hardware is not obvious since all classical HD/VSA models used abstract connectionist representations, not spikes. However, recent work has demonstrated that representations of a complex HD/VSA model, Fourier HRRs [53], can be mapped to spike timing codes [170]. Although focused just on content-addressable memory, i.e., item memory, this work opens avenues for efficient implementations of full HD/VSA models on neuromorphic hardware [9]. Because neuromorphic hardware often optimizes spike communication for sparse network connectivity, the scaling properties of neuromorphic HD/VSA will potentially outperform other types of hardware. Furthermore, neuromorphic hardware might enable hybrid approaches by integrating HD/VSA with other computing frameworks. For instance, an event-based dynamic vision sensor (as a front-end spiking sensor) has been combined with sparse HD/VSA leading to 10× higher energy efficiency than an optimized nine-layer perceptron with comparable accuracy on an eight-core low-power digital processor [89].

The results above bring a question worth discussing—what are the common hardware primitives enabling these gains? The most common architectural primitives that are observed in the hardware implementations can, actually,

be naturally mapped to basic elements (see Section III-A) and operations (see Section III-B) of HD/VSA. For example, let us consider the implementations of the binary spatter codes model based on phase-change memory devices reported in [16] and the sparse block codes model on spiking neural network circuits described in [18]. The basic hardware primitives lying at the core of these implementations were item memory circuit (cf. [16, Fig. 1] and [18, Sect. III-A1a]), superposition operation circuit ("the complete in-memory HD system" in [16] and [18, Fig. 2]), binding operation circuit (cf. [16, Fig. 3] and [18, Fig. 4]), and circuit for probing (cf. [16, Fig. 2] and [18, Fig. 3]).

The fact that the basic HD/VSA elements and operations are the most common hardware primitives should not be surprising because, as it was demonstrated in Section IV-A, they are the key building blocks of all the computational primitives in the "HD/VSA cookbook." This implies that, given the hardware implementation of the most basic elements, it is possible to construct architectures for compositional primitives that might, e.g., combine the usage of several HD/VSA operations. This, of course, does not mean that there is no other way to approach hardware implementation of HD/VSA. In fact, there are incentives to design implementations targeting concrete compositional primitives, and they were even present in the two above works, e.g., a circuit for representing n-grams (see [16, Fig. 3]) and a circuit for representing a set of key-value pairs (see [18, Fig. 5]). The main incentive for doing so is to increase the efficiency of the implementation since it allows applying, e.g., computational reuse. A vivid example of such an approach is a circuit from [130] (cf. [130, Fig. 3]) for generating hypervectors of trigrams (see Section IV-A6) that used Barrel shifters to minimize the switching activity during the permutation operations. Note that the same circuit could have been designed using the hardware primitives for binding and permutation operations as the building blocks, but such a design would come at the price of reduced efficiency. Another common bottleneck in the hardware implementations of machine learning applications of HD/VSA is the item memory (cf. [161, Fig. 8]). The presence of this bottleneck caused researchers to consider ways of efficiently eliminating it. A prominent way to do so is the rematerialization of the item memory using inexpensive recurrent methods, as proposed in [162], [178], and [179]. This idea of rematerialization created room for trading off the system's dynamic and leakage powers and was demonstrated to increase energy efficiency in scenarios involving, e.g., biosignal processing [162], [180], [181].

In summary, we can argue that hardware implementations of HD/VSA rely on architectural primitives corresponding to the basic elements and operations of HD/VSA. However, in order to increase the efficiency, it is also common to design circuits implementing compositional computational primitives from Section IV-A.

2) HD/VSA Combined With Neural Networks: The aforementioned works have demonstrated the benefits of HD/VSA on relatively small-scaled classification tasks. In order to approach more complex tasks, a common strategy is to combine some of the basic HD/VSA primitives (discussed in Section IV-A) with neural networks. For instance, representations from pretrained neural networks have been used with the HD/VSA primitives to compactly represent a set of key-value pairs to generate image descriptors for visual place recognition [158], [159]. One step further, the deep neural networks were trained from scratch to be able to directly generate desired hypervectors that were further bound or superposed by HD/VSA operations to represent the concepts of interest [147], [153], [182]. They achieved state-of-the-art accuracy compared to the stand-alone deep learning solutions in various tasks involving images, including few-shot learning [147], continual learning [182], and visual abstract reasoning [153]. The hardware implementation of such *hybrid* architectures may vary. For instance, the associative memory for few-shot learning was implemented on the phase-change memory devices to execute searches in constant time, while the neural network was implemented externally [147]. Alternatively, the whole architecture for the visual abstract reasoning was executed on CPUs, whereby leveraging HD/VSA leads to two orders of magnitude faster execution than the functionally equivalent symbolic logical reasoning [153].

VI. DISCUSSION

HD/VSA has been criticized for lacking a structured methodology to design systems and missing well-defined design patterns [86]. Here (see Section IV-A), we compiled existing computational primitives with HD/VSA that paint a different picture. There is an HD/VSA methodology addressing a wide range of applications, but it is scattered throughout the literature. In addition to compiling existing work, we laid out design principles for building distributed representations of data structures, such as sets, sequences, trees, and key-value pairs. This demonstrates a rich algorithmic and representation-level approach that one can use as an abstraction for the next generation of computing devices.

Our compilation of varied HD/VSA primitives also suggests that, contrary to some earlier assessments (see [183] and the commentary in [184]), the repertoire potential of HD/VSA applications is extremely wide, ranging from low-level sensory processing to high-level reasoning. While we provided an extensive introduction to HD/VSA and a comprehensive collection of computational primitives and existing connections to computing hardware, it was not our goal to provide a complete overview of the area such as, e.g., a review of all existing HD/VSA models. We do however, hope that this article will motivate readers to explore the current state of the area that is covered in detail in a two-part survey covering both fundamentals [73] and applications [185]. We think that the strength

of HD/VSA benefits applications where there is a need for a computing framework constructing transparent compositional distributed representations that will allow interfacing unconventional parallel computing hardware. It is not obvious how to achieve this with, e.g., modern neural networks, though it should be noted that there is increasing empirical evidence demonstrating that certain problems benefit from hybrid approaches combining elements from HD/VSA and neural networks.

That being said, it is still important to admit the limitations and challenges of HD/VSA, and therefore, before ending this article, we would like to mention them (see Section VI-A). We conclude by discussing the role of HD/VSA as a framework for computing with emerging hardware (see Section VI-B).

A. Limitations and Open Challenges

Here, we would like to emphasize some of the limitations of HD/VSA that are directly related to the scope of this article: applications (see Section VI-A1), dimensionality of hypervectors (see Section VI-A2), and flow control (see Section VI-A3). For a broader discussion of open challenges, we kindly refer the reader to the section "Open Issues" in [185].

1) Applications: There have been numerous attempts to use HD/VSA in problems within various application domains (see [185] for detailed coverage). Some well-known examples of using HD/VSA include word embedding [186], [187] (though largely overshadowed by [188], [189]), analogical reasoning [13], [97], cognitive architectures [37], [38], and modeling [190], [191], as well as solving classification tasks [15], [34]. It must be admitted, however, that most of these use cases were limited to small scope problems; therefore, there is still a need to demonstrate how HD-/VSA-based solutions scale up to real-world computational problems and, what is also important, to identify niches where the advantages of HD/VSA are self-evident. We think that further research will eventually address this limitation as we see two recent developments in this direction. First, there is a continuing trend towards extending HD/VSA to novel domains—promising recent examples include applications in communications [83] and distributed systems [79]. Second, there is an increasing number of studies (see Section V-B2 and, e.g., [147], [153], [156], [158], [160], and [192]) that combine neural networks and HD/VSA primitives. This seems to be a promising way to scale up HD-/VSA-based solutions to real-world problems in the short-term.

2) HD/VSA Dimensionality and Working Memory: The key feature of data representation in HD/VSA is that data structures are represented by fixed-sized hypervectors, independent of the size of the data structure. This is in contrast to the localist representations of data structures, which grow linearly or even quadratically with the number of elements. On the one hand, it is a great advantage as data structures of arbitrary size and shape can be manipulated in parallel with the elementary set of HD/VSA operations. At the same time, as we have seen in Section IV-A, the dimensionality of hypervectors might easily become a limitation since, for a given dimensionality, the information content of representation, i.e., the HD/VSA capacity, limits the size of data structures that can be represented reliably [30], [193].

Conceptually, one should think of the memory in hypervectors as the working memory or working registers, holding the data relevant during an ongoing computation. In contrast, the role of long-term memory for an HD-/VSA-based system can be fulfilled by, e.g., a large capacity associative content-addressable memory that might store hypervectors of data structures [37], [194]. Currently, this idea is being investigated by the community [195].

The limitation of the working memory in HD/VSA has interesting parallels to the limitation of the human working memory. For data structures of limited size, there are guarantees for exact reconstruction [193]. However, transcending the theoretical bound for exact reconstruction, the data representation becomes lossy, with error rates being theoretically predictable [30]. HD/VSA representations of data structures in the lossy regime have been shown to reproduce some properties of the human working memory. For example, the recall of a sequence in an HD/VSA, as described in Section IV-A5, can reproduce the performance of humans remembering sequences [120], [122]. Furthermore, the modeling of memorizing sequences with HD/VSA was linked to the neuroscience literature in [125]. It is not immediately clear how this capturing of the limitations of human memory might be beneficial in engineering applications. The way biological working memory coarsens its content and gradually degrades might be an important feature of cognition whose benefits are not yet fully appreciated. However, for applications that require guarantees for exact reconstruction, the dimensionality of hypervectors needs to be specified at the design stage, which makes it a limitation for situations where the data structures to be represented can be of highly varying sizes.

3) Flow Control: HD/VSA implementations of algorithms generally rely on existing non-HD/VSA mechanisms for flow control. This is reasonable in systems where the aim is to use HD/VSA to implement conventional computing approaches. This case can be seen more as a way of extending conventional computing with HD/VSA. However, if we are modeling biological systems, we should not be using non-HD/VSA conventional computing flow control. Moreover, from the efficiency point of view, when using emerging hardware, it might not be desirable to have a conventional processing unit for flow control. For these reasons, it is important to develop methodologies for flow control that would use native HD/VSA primitives. In our opinion, this is possible. However, to date, the efforts in this direction are quite limited. There was an attempt

in [196] to define a model of a biological system with HD-/VSA-based control. Two other related efforts are [41], which presented a proposal for a stack machine and, [58], proposing a processor with instructions specified in the form of hypervectors.

B. HD/VSA as a Framework for Computing With Emerging Hardware

HD/VSA was originally proposed in cognitive neuroscience as a model for symbolic reasoning with distributed representations. More recently, it has been shown that HD/VSA can formulate subsymbolic computations, for example, in machine learning tasks.

Here, we proposed that HD/VSA provides a computing framework within the algorithmic level of Marr's framework [19] for linking abstract computation and emerging hardware levels. The algorithmic formalism of HD/VSA (with few exceptions) is the same for all of its variants. Thus, HD/VSA enables a model-independent formulation of computational primitives. At the same time, HD/VSA also provides a seamless interface between algorithms and hardware. In Section V-A, we illustrated how different HD/VSA models can connect to specific types of emerging hardware. Moreover, in Section IV-B, we demonstrated how HD/VSA can be used for computing in superposition. This feature extends HD/VSA beyond the conventional computing architectures, and we foresee that, together with algorithms that leverage computing in superposition, such as resonator networks [76], [77] (see Section III-C1), it will pave the way towards efficient solutions of nontrivial combinatorial search problems (see examples in [148] and [197]).

Another interesting aspect of computing with hypervectors is that it occupies a realm between digital and analog computing. After each computation step in a digital computer, all vector components are pulled to one of the possible digital states (bits). This individual discretization of each component avoids error accumulation. Conversely, an analog computer is supposed to implement an analog dynamical system to predict its future states. Any deviation between the dynamical system to be analyzed and its computer implementation (e.g., noise) leads to uncontrollable error accumulation in analog computers. HD/VSA operations leverage analog operations on vectors without discretization. However, discretization takes place on the entire vector level when a resultant hypervector is matched with the entries in the item memory. Thus, HD/VSA can leverage (potentially very) noisy dynamics in the high-dimensional state space of emerging hardware while still protecting against error accumulation.

Despite all the promising aspects mentioned above, the practicability of the HD/VSA computing framework for emerging computing hardware is yet to be thoroughly quantified. An important future direction is to develop a systematic methodology to quantitatively measure and compare side-by-side the efficiency of different computing frameworks on concrete hardware. In this article, we concentrated on the question of how HD/VSA enables the construction of varied algorithmic primitives and, therefore, could be a possible candidate framework in such a comparison.

Table 1 Qualitative Assessment of HD/VSA Capabilities Compared to Conventional Computing and Neural Networks

	Conventional computing/AI	Neural networks	HDC/VSA
Distributed representation	✗	✓	✓
Learning from data	✗	✓	✓
Symbolic computing with variables and binding	✓	✗	✓
Tolerance to device imperfections	✗	?	✓
Transparency	✓	✗	✓

1) Alternative Frameworks: HD/VSA constitutes a computing framework that provides an algebraic language for formulating algorithms and, at the same time, links the computation to distributed states on hardware. Table 1 compares the qualitative properties of HD/VSA as a computing framework to conventional computing and neural networks.

There is a tradeoff between how general a framework is in terms of computation and how closely it is linked to implementation. A general purpose framework typically requires a full sealing between implementation and computation, such as, for example, the conventional computing architecture. Conversely, a framework that is well matched to implementation, and can, therefore, efficiently leverage the hardware, is typically of quite special purpose. We argue that the tradeoff HD/VSA provides between generality and linking to implementation, which is ideal for emerging hardware. In particular, it seamlessly provides implementations of algorithms that leverage distributed representations and parallel operations, and can tolerate noise and imprecision [43]. Of course, HD/VSA is not the only framework candidate for emerging hardware; alternative approaches include probabilistic computing [198], sampling-based computing [199], computing by assemblies of neurons [200], and dynamic neural fields [201]. For example, in neuromorphic computing, the dynamic neural field is an alternative computing framework that could support fully symbolic operations. In fact, dynamic neural fields and HD/VSA might complement each other by combining the real-time dynamics of dynamic neural fields with the computational power and scalability of HD/VSA. The detailed comparison between these approaches and HD/VSA is, however, outside the scope of this article. Nevertheless, in our opinion, HD/VSA is the most transparent approach in structuring computation and the most general with regard to different types of hardware. In terms of formulating algorithms and computational primitives, HD/VSA offers a common language, independent of a particular HD/VSA model. For the desired computation on given hardware, one of the many existing HD/VSA models can provide the most

advantageous implementation in terms of energy and time efficiency.

There is currently a plethora of collective-state computing approaches emerging, such as compressed sensing, Bloom filters, and reservoir computing, all relying on distributed representations [169]. These approaches are rather disjoint and typically focus on special purpose computing applications. HD/VSA has been shown to be able to formalize different types of collective-state computing, including reservoir computing [28], [30], Bloom filters [78], compressed sensing [59], randomized kernels [102], [150], and extreme learning machines/random vector functional link networks [29]. Thus, we see HD/VSA as a promising candidate framework for providing a "lingua franca" for collective-state computing.

APPENDIX A ON TURING COMPLETENESS OF HD/VSA

It is practical to have a collection of primitives for common data structures. However, these primitives alone do not provide us with a quantification of the theoretical capabilities of using HD/VSA as a computing framework. Of course, it is desirable that a computing framework for emerging hardware be able to (in theory, at least) execute any algorithm. For example, in [202] that proposed a system hierarchy for neuromorphic computing, it has been emphasized that Turing completeness is an essential property for an abstraction model that is used at the algorithmic level. Therefore, in this section, we sketch ways of demonstrating that HD/VSA is computationally universal by exemplifying how they (with some assumptions) can be used to emulate systems that have already been proven to be Turing complete. While computing in superposition is likely to be the most interesting feature of operating with HD/VSA, computational universality is still a critical property to study as it characterizes the general computational power of a system. It is worth noting that, among HD/VSA researchers, there is a general agreement that HD/VSA is computationally universal, but, to the best of our knowledge, this has not been shown yet. Therefore, here, we make two proposals toward demonstrating their universality: by implementing a Turing machine and by emulating an elementary cellular automaton, which is also known to be Turing complete [203]. Note that, while these proposals might not be tight enough to be qualified as formal proof, we believe that the following directions are the most promising ways to make such proof.

A. Implementation of Turing Machines With HD/VSA

Since there are a number of small Turing machines known to be universal [204], we first focus on demonstrating how HD/VSA can be used as a part of an implementation of such a machine. In order to do so, we present how HD/VSA representations are used to map a table of behavior [204] and execute the machine.

Table 2 Table of Behavior of (2, 4) Turing Machine

	A	B
0	2 L A	3 R A
1	3 L B	2 L B
2	3 L A	0 R B
3	3 L A	1 R B

The presented implementation could be used to realize any Turing machine, but, for the sake of compactness, we exemplify the implementation with a (2, 4) Turing machine, which has two states (A and B) and four symbols (0, 1, 2, and 3). The table of the behavior of a (2, 4) Turing machine is presented in Table 2. For a given combination of the current state and the tape's content, it provides which symbol should be written to the current cell, the next state of the machine, and the direction for the head's movement.

1) HD/VSA Implementation of the Table of Behavior: We use the MAP model described above. In order to represent the table of the behavior of a Turing machine, we first create two item memories populated with random hypervectors. One item memory stores the states, e.g., in the case of a (2, 4) Turing machine, it includes only two hypervectors for states A and B (denoted as **a** and **b**), respectively. Another item memory stores hypervectors for symbols. Since the considered machine uses only four symbols, four hypervectors, **0**, **1**, **2**, and **3**, are sufficient. These item memories are used to construct a hypervector for each combination of states and symbols. The hypervector is constructed by applying the binding operation on the hypervectors for a state and a symbol.

Eight hypervectors corresponding to all possible combinations form a basis for constructing a third, heteroassociative, item memory, i.e., the memory where the address and content parts store different hypervectors. The heteroassociative item memory can implement any table of behavior by using the bound pair of state and symbol as input to the memory and issuing hypervectors, which should be used as the tape content, head's move, and next state as an output. Table 3 presents the heteroassociative item memory for the table of the behavior of (2, 4) Turing machine. Thus, three item memories constitute the static

Table 3 Heteroassociative Item Memory Implementing (2, 4) Turing Machine

Address (input)	Content (output)		
	Tape content	Next State	Head's move
$\mathbf{a} \odot \mathbf{0}$	**2**	**a**	L
$\mathbf{a} \odot \mathbf{1}$	**3**	**b**	L
$\mathbf{a} \odot \mathbf{2}$	**3**	**a**	L
$\mathbf{a} \odot \mathbf{3}$	**3**	**a**	L
$\mathbf{b} \odot \mathbf{0}$	**3**	**a**	R
$\mathbf{b} \odot \mathbf{1}$	**2**	**b**	L
$\mathbf{b} \odot \mathbf{2}$	**0**	**b**	R
$\mathbf{b} \odot \mathbf{3}$	**1**	**b**	R

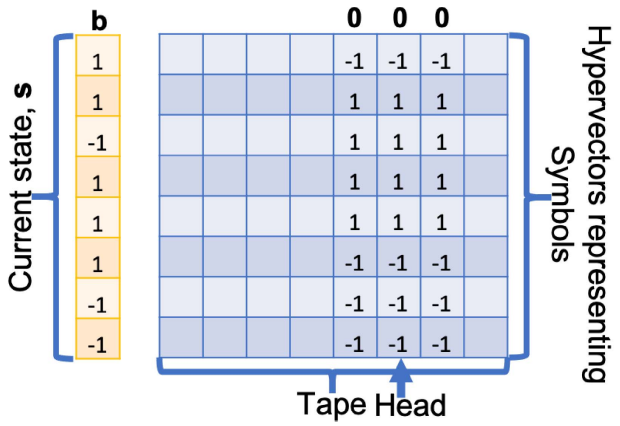


Fig. 10. *Illustration of the current state of the machine and its tape.*

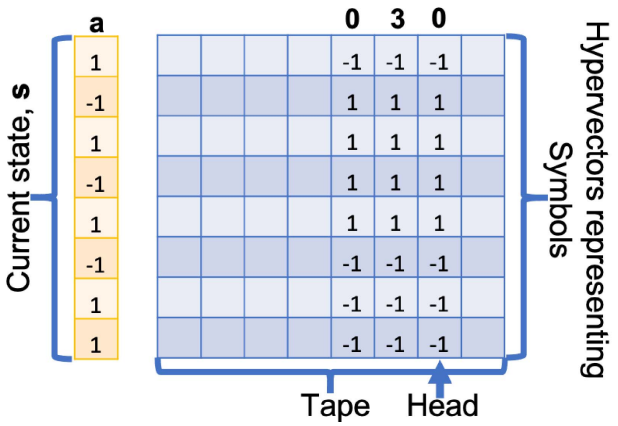


Fig. 11. *Updated state and tape of the machine after the previous state as in Fig. 10.*

part of the system, which is generated only once at the initialization. At this point, it is worth making a note that, in addition to the standard assumptions about unlimited time and memory resources, there is an extra assumption about the heteroassociative item memory. In particular, it should be guaranteed to behave correctly in the absence of external errors. Practically, it means that the address part of the heteroassociative item memory should not have repeated entries. Even for moderate dimensionality of hypervectors, the chance of such an event is low, but, if this happens, the issue is solved by the regeneration of the item memories.

2) HD-/VSA-Based Tape: The other part of the system is dynamic and includes the location for storing a hypervector for the current state, the tape, and the current position of the head. Fig. 10 presents an example of the dynamic part of the system. In the case of using HD/VSA, the tape can be seen as a matrix where each column corresponds to the hypervector of a symbol. In order to make the next step, the machine has to read the hypervector of the current state (**b** in Fig. 10) and the hypervector of the symbol at the current location of the head (**0** in Fig. 10). The result of binding of these hypervectors $\mathbf{b} \odot \mathbf{0}$ is used as an input to the heteroassociative memory. The output of the memory indicates that hypervector **a** should be written to the current state; the tape's content is changed to **3**, and the head should be moved to the right of the current location. The updated state is shown in Fig. 11. In such a manner, the system could operate on the tape for the required number of computational steps. In summarizing, the proposed implementation of a Turing machine uses basic elements of HD/VSA, such as hypervectors, item memories, and the binding operation; however, it also includes few parts that go beyond HD/VSA, namely, control of head movements and unlimited memory tape.

3) Scaling HD/VSA Implementation: Since the proposed implementation of a Turing machine does not make use of the superposition operation, there is no crosstalk noise being introduced to the computations, which, in turn, means that, in the absence of external noise, the emulation behaves in a deterministic way. Thus, even tiny 3-D vectors can be used to construct the heteroassociative item memory with unique entries. Nevertheless, since one of the arguments in favor of HD/VSA is their built-in tolerance to errors, it is interesting to observe the behavior of the emulation in the presence of external noise. We performed simulations where the external noise was added to the tape by randomly flipping signs of a fraction of hypervector components. Fig. 12 presents the average dimensionality of hypervectors required to make at least 10^9 error-free updates of the emulated Turing machine

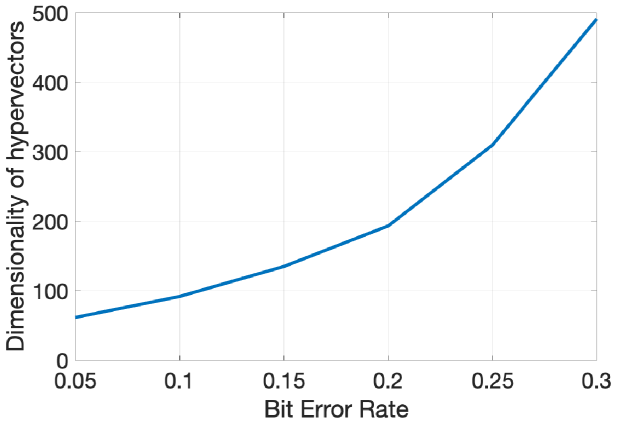


Fig. 12. *Average dimensionality of hypervectors required to make at least 10^9 error-free updates of the emulated (2, 4) Turing machine when the hypervectors representing symbols on the tape were subject to external bit flips. The BER was in the range [0.05, 0.30] with a step of 0.05. The results were computed from ten simulation runs with random initializations of hypervectors in the item memories and random bit flips added at every update of the machine.*

when the hypervectors representing symbols on the tape were subject to external bit flips. The bit error rate (BER) varied in the range [0.05, 0.30] with a step of 0.05. The starting dimensionality of hypervectors was 2^4. If the error in emulation was happening in less than 10^9 steps, then the dimensionality was increased by 10%. The results demonstrate that the proposed implementation can reliably emulate the Turing machine given adequate resources (i.e., the dimensionality of hypervectors). Naturally, in the presence of external noise, more resources are needed to obtain the error-free execution of the machine. Nevertheless, an important observation is that the implementation works with imprecise noisy representations. Moreover, the robustness of the implementation comes at no cost in terms of design, as the same algorithm is being used for any amount of noise, and the only cost to be paid is the increased size of the system.

B. Emulation of Cellular Automaton With HD/VSA

Since HD/VSA is designed to create vector representations of symbolic structures, when identifying a Turing complete system suitable for emulation with HD/VSA, it is also natural to choose a highly structured system that uses a small finite alphabet of symbols. We think that an elementary cellular automaton is one example of such a system. Since the elementary cellular automaton with rule 110 is known to be Turing complete [203], we would like to demonstrate how HD/VSA can be used in emulating this rule. In order to do so, we first revisit the elementary cellular automaton concept. Next, we present an HD/VSA algorithm for mapping and executing an elementary cellular automaton. Thus, we literally follow the roadmap from [203]: "the automaton itself is so simple that its universality gives us a new tool for proving that other systems are universal." Finally, we explore how the proposed implementation is scaling with respect to the size of the initial grid state of an elementary cellular automaton, the dimensionality of hypervectors, and the amount of noise present during the computations. The major point of the latter is that, even for a large amount of noise, the implementation can perfectly emulate the elementary cellular automaton given sufficiently large dimensionality of hypervectors, which is a nice property as robustness is achieved without modifying the design.

1) Elementary Cellular Automata: An elementary cellular automaton is a discrete computational model consisting of a 1-D grid of cells [205]. Each cell can be in one of a finite number of states (two—for the elementary automaton). States of cells evolve in discrete time steps according to a fixed rule. The state of a cell at the next computational step depends on its current state and the states of its neighbors. The computations performed by an elementary cellular automaton are local. The new state of a cell is determined by the previous states of the cell itself and its two neighboring cells (left and right). Thus, only three cells

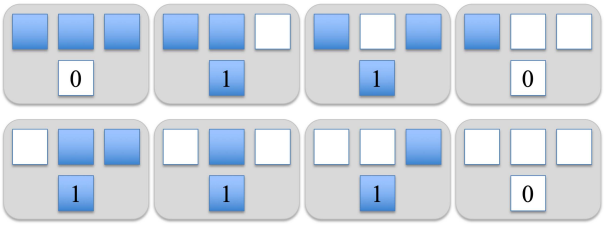


Fig. 13. *Assignment of new states for a center cell when the cellular automaton uses rule 110. A hollow cell corresponds to a zero state, while a shaded cell marks one state.*

are involved in a computation step, i.e., for binary states, there are in total $2^3 = 8$ combinations. A rule assigns states for each of the eight combinations. Fig. 13 presents all combinations and the corresponding states for rule 110.

2) HD/VSA Algorithm for Emulating an Elementary Cellular Automaton With the Rule 110: We use the MAP model described above. In order to represent an elementary cellular automaton with rule 110, we first create two item memories populated with random hypervectors. One item memory stores the finite alphabet, i.e., it includes only two hypervectors, for one and zero (denoted as $\mathbf{1}$ and $\mathbf{0}$, respectively). Another item memory stores hypervectors for positions. Since an elementary cellular automaton relies only on a cell in focus and its immediate neighbors, then three hypervectors, $\mathbf{l}$ (left), $\mathbf{c}$ (center), and $\mathbf{r}$ (right), are sufficient. These item memories are used to construct a hypervector for each combination of states in three consecutive cells. The hypervector is constructed by applying the superposition operation on the bound pairs of a positional hypervector and an alphabet hypervector. In other words, the current states in three consecutive cells are represented as a set of unordered pairs. For example, for 010, the corresponding compound hypervector is constructed as

$$\mathbf{h}_{010} = [\mathbf{l} \odot \mathbf{0} + \mathbf{c} \odot \mathbf{1} + \mathbf{r} \odot \mathbf{0}].$$

All eight compound hypervectors form a basis for constructing a heteroassociative item memory, which can implement any elementary rule by using the compound hypervectors as input to the memory, and issuing either $\mathbf{1}$ or $\mathbf{0}$ (determined by the rule) as an output. Table 4

Table 4 Heteroassociative Item Memory Implementing Rule 110

Address (input)	Content (output)
$\mathbf{h}_{111} = [\mathbf{l} \odot \mathbf{1} + \mathbf{c} \odot \mathbf{1} + \mathbf{r} \odot \mathbf{1}]$	$\mathbf{0}$
$\mathbf{h}_{110} = [\mathbf{l} \odot \mathbf{1} + \mathbf{c} \odot \mathbf{1} + \mathbf{r} \odot \mathbf{0}]$	$\mathbf{1}$
$\mathbf{h}_{101} = [\mathbf{l} \odot \mathbf{1} + \mathbf{c} \odot \mathbf{0} + \mathbf{r} \odot \mathbf{1}]$	$\mathbf{1}$
$\mathbf{h}_{100} = [\mathbf{l} \odot \mathbf{1} + \mathbf{c} \odot \mathbf{0} + \mathbf{r} \odot \mathbf{0}]$	$\mathbf{0}$
$\mathbf{h}_{011} = [\mathbf{l} \odot \mathbf{0} + \mathbf{c} \odot \mathbf{1} + \mathbf{r} \odot \mathbf{1}]$	$\mathbf{1}$
$\mathbf{h}_{010} = [\mathbf{l} \odot \mathbf{0} + \mathbf{c} \odot \mathbf{1} + \mathbf{r} \odot \mathbf{0}]$	$\mathbf{1}$
$\mathbf{h}_{001} = [\mathbf{l} \odot \mathbf{0} + \mathbf{c} \odot \mathbf{0} + \mathbf{r} \odot \mathbf{1}]$	$\mathbf{1}$
$\mathbf{h}_{000} = [\mathbf{l} \odot \mathbf{0} + \mathbf{c} \odot \mathbf{0} + \mathbf{r} \odot \mathbf{0}]$	$\mathbf{0}$

presents the heteroassociative item memory for the rule 110. Thus, three item memories constitute the static part of the system, which is generated only once at the initialization.

The other part of the system performs computations for a given initial grid state of length l at time $t = 0$. The initial grid state is mapped to a compound hypervector (denoted as $\mathbf{a}_0$). The mapping is done by applying the superposition operation on all hypervectors representing the states of cells at all positions. Position j in the grid is represented by applying the permutation operation j times to the hypervector corresponding to a state at position j. Thus, this representation corresponds to the mapping of a sequence with the superposition operation. For example, if the initial grid state is 10101, then the representation of the state at the fifth position is $\rho^5\mathbf{1}$, while the compound hypervector for the initial grid state is

$$\mathbf{a}_0 = [\rho^1\mathbf{1} + \rho^2\mathbf{0} + \rho^3\mathbf{1} + \rho^4\mathbf{0} + \rho^5\mathbf{1}].$$

Given $\mathbf{a}_0$, the next step is to compute $\mathbf{a}_1$ or, in general, $\mathbf{a}_{t+1}$ given $\mathbf{a}_t$.

First, $\mathbf{a}_{t+1}$ is initialized to be an empty hypervector. Next, for each position j ranging from 1 to l, we do the following (this step can be either serial or parallel).

1) Approximately recover the state at j and its neighbors as $\hat{\mathbf{h}} = [\mathbf{l} \odot \rho^{-(j-1)}\mathbf{a}_t + \mathbf{c} \odot \rho^{-j}\mathbf{a}_t + \mathbf{r} \odot \rho^{-(j+1)}\mathbf{a}_t]$.
2) Use $\hat{\mathbf{h}}$ as the query to the heteroassociative item memory. The memory returns the content (i.e., $\mathbf{0}$ or $\mathbf{1}$) for the address closest to $\hat{\mathbf{h}}$ in terms of dot product. The returned content is denoted as $\mathbf{v}_j$.
3) Modify $\mathbf{a}_{t+1}$ with $\mathbf{v}_j$ as $\mathbf{a}_{t+1} += \rho^j\mathbf{v}_j$.

Finally, apply the majority rule on $\mathbf{a}_{t+1}$: $\mathbf{a}_{t+1} = [\mathbf{a}_{t+1}]$, so that it becomes bipolar. In such a manner, the system could iterate through the grid for the required number of computational steps.

Last but not least, it is worth explicating that the proposed implementation assumes parts that go beyond HD/VSA. First, the full computational system has its control architecture that is responsible for initializing the grid state and running the for-loop, which can be seen as a recurrent connection, required for constructing $\mathbf{a}_{t+1}$. The second part that is assumed here to be the same as in the standard implementation of a cellular automaton is the circuit determining when to stop the computation. We have not focused on this circuit as our main goal here was to demonstrate how to evolve HD/VSA representations to perform cellular automaton computations.

3) Scaling HD/VSA Emulation: It is known that compound hypervectors can be used to retrieve their components (see Section III-C); however, there is a limit on the number of components, which can be stored in a compound hypervector without losing the ability to recover the components [30]. The rule of thumb is that, for larger hypervector dimensionalities, more components can be recovered from a compound hypervector. For the task of emulating an elementary cellular automaton, it is important that $\hat{\mathbf{h}}$ is similar enough to the correct state hypervector in the item memory. Otherwise, we will introduce errors to the computations being emulated, which is highly undesirable. When constructing $\hat{\mathbf{h}}$, the main source of noise is the crosstalk noise from other cell states stored in $\mathbf{a}_t$. Therefore, in order to avoid errors in the computations, the dimensionality of hypervectors should depend on the length of the grid: the longer the grid, the larger the dimensionality is required for robustly querying the item memory.[6]

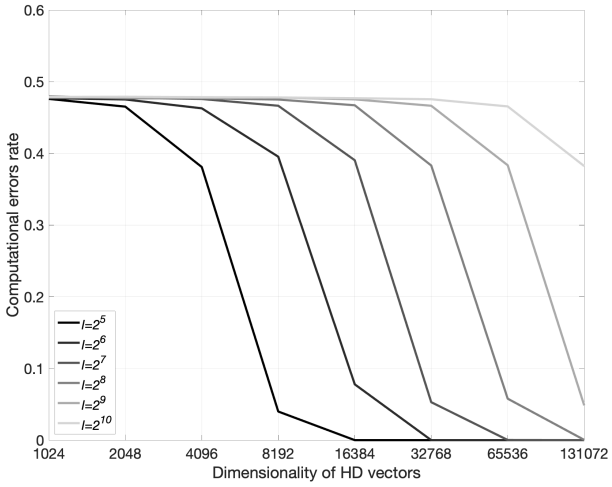


Fig. 14. ***Average error rate after 100 computational steps of the elementary cellular automaton against the dimensionality of hypervectors ($N = 2^i$, $i \in [10, 17]$) for several different lengths of the grid ($l = 2^i$, $i \in [5, 10]$). The results were computed from 100 simulation runs with random initializations of hypervectors in the item memories. The initial grid states were also randomized.***

Fig. 14 presents the empirical results for a range of l and N values. The curves depict the average error rate after 100 computational steps of the elementary cellular automaton. Note that the errors occurring at the earlier computational steps will most likely propagate to the successive steps. The length of the grid, l, varied as 2^i, $i \in [5, 10]$, while the dimensionality of hypervectors, N, varied as 2^i, $i \in [10, 17]$. Thus, the results demonstrate that HD/VSA can perfectly emulate the elementary cellular automaton with a grid of a certain length, given adequate resources (i.e., the dimensionality of hypervectors).

Note that Fig. 14 presented the results for the case when hypervectors did not include any external noise. Since one of the arguments in favor of HD/VSA is their built-in tolerance to errors, it is interesting to observe the behavior of the emulation in the presence of external noise. External noise was added by randomly flipping a fraction of components in $\mathbf{a}_t$, but it was still assumed that

[6] In principle, it should be possible to analytically find the minimal dimensionality of hypervectors for robustly emulating the grid of the given length.

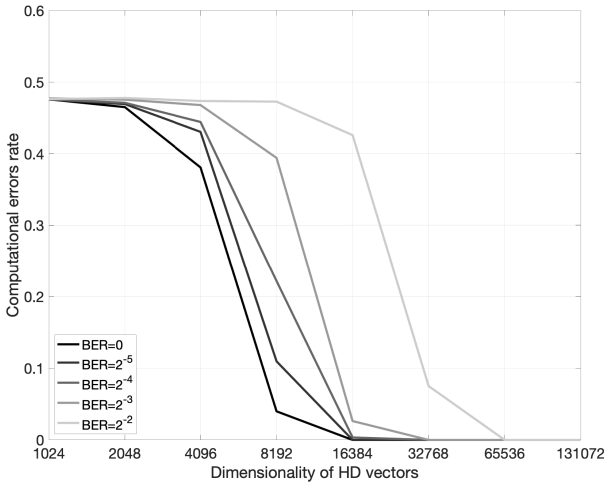


Fig. 15. *Average error rate after 100 computational steps of the elementary cellular automaton against the dimensionality of hypervectors ($N = 2^i$, $i \in [10, 17]$) for several different BERs ($p = 2^{-i}$, $i \in [2, 5]$) for the length of the grid $l = 32$. The results were computed from 100 simulation runs with random initializations of hypervectors in the item memories. The initial grid states were also randomized.*

the control architecture functions without errors. Fig. 15 presents the average error rate after 100 computational steps of the elementary cellular automaton in the presence of external noise. The BER, p, varied as 2^{-i}, $i \in [2, 5]$. The length of the grid was fixed to $l = 32$.

The results demonstrate that, naturally, in the presence of external noise, more resources are needed to obtain error-free emulation. Nevertheless, an important observation is that the HD-/VSA-based system works with imprecise noisy representations. Moreover, the robustness of the system comes at no cost in terms of design, as the same algorithm is used in both cases, and the only cost to be paid is the increased size of the system.

4) Studies Related to Computational Universality of HD/VSA: Studying the computational universality of a particular computing framework is important for understating the ultimate theoretical limitations of computing hardware using this framework. For example, Siegelmann and Sontag [206] have shown that recurrent neural networks are computationally universal; Perez *et al.* [207] have shown the universality of modern transformer and Neural GPU networks. Since HD/VSA can express some recurrent neural networks [28], studying their universality by leveraging the existing results for neural networks is a possible direction of research. We, however, followed earlier approaches that showed that neural network-like systems can implement Turing machines [208]. In Appendix A-A and Appendix A-B, we sketched how HD/VSA can be used in implementations of a small Turing machine [204] and a universal elementary cellular automaton with the rule 110 [203].

Recently, Kwisthout and Donselaar [209] emphasized the need for a formal machine model for novel neuromorphic hardware in order to develop a computational complexity theory for neuromorphic computations. This is an important direction of research for understanding the full potential of emerging hardware. They argued, however, that, in order to encompass the computational abilities of neuromorphic hardware, one will likely need to define an entirely new computing theory framework. Their study has proposed to use spiking neural networks (shown to be Turing complete [210]) because, similar to HD/VSA, they are suitable for co-located computation and memory, and massive parallelism—which is not the case for the conventional computing architecture.

In addition to the demonstration of universality, an important practical question is how a complete computational architecture should look like. This is still an open question. A proposal has been sketched in [58], which featured an HD-/VSA-based processor where both data and instructions were represented as hypervectors. There is another approach known as tensor product variable binding, which is closely related to HD/VSA. For example, tensor product variable binding can also be used to represent data structures in distributed representations [211]. The study [50] has demonstrated how to implement push, pop, and the Lisp primitives CAR and CDR with tensor product variable binding, while Dolan and Smolensky [212] have demonstrated how to implement a production system. An HD-/VSA-based model, which was positioned as a general-purpose neural controller playing a role analogous to a production system, was proposed in [196].

Another relevant result is a demonstration of the feasibility of implementing fluid construction grammars with HD/VSA [213]. Even though fluid construction grammars have not been shown to be universal, it is a powerful and interesting approach for both cognitive and evolutionary linguistics. Knight *et al.* [213] proposed a vision similar to the one presented in Fig. 1. They suggest that HD/VSA can be seen as a "virtual machine" that can have different (independent) physical implementations, such as an indirect mapping to spiking neurons [170] or direct mapping of operations with analog/digital implementations [16].

APPENDIX B
SUMMARY OF VECTOR-SYMBOLIC SPACE AND OPERATIONS

A. Key Components

This appendix presents excerpts from Section III providing a summary of HD/VSA. The key components of all HD/VSA are given as follows:

1) high-dimensional space (e.g., bipolar);
2) orthogonality;
3) similarity measure (e.g., dot product $\langle \mathbf{a}, \mathbf{b} \rangle$);
4) seed representations (e.g., random i.i.d. vectors);
5) operations on representations.

There are three key operations in HD/VSA:

1) binding (denoted as $\odot$, implemented as componentwise multiplication (Hadamard product) in the MAP model);

2) superposition (denoted as $+$, implemented as componentwise addition, enclosed in $[\ldots]$ when thresholded);
3) permutation (denoted as ρ, e.g., rotation of coordinates).

In the following, we present the properties of the implementations of these operations for the MAP HD/VSA model [54]. Here, we enumerate the properties assuming that the seed hypervectors are bipolar.

B. Properties of the Binding Operation

The binding operation has the following properties:

1) Binding is commutative: $\mathbf{a} \odot \mathbf{b} = \mathbf{b} \odot \mathbf{a}$.
2) Binding distributes over superposition: $\mathbf{c} \odot (\mathbf{a} + \mathbf{b}) = \mathbf{c} \odot \mathbf{a} + \mathbf{c} \odot \mathbf{b}$.
3) Binding is invertible: $(\mathbf{a} \odot \mathbf{b}) \odot \mathbf{b} = \mathbf{a}$ (bipolar $\mathbf{b}$ is self-inverse); the inverse operation is called releasing or unbinding.
4) Binding is associative: $(\mathbf{a} \odot \mathbf{b}) \odot \mathbf{c} = \mathbf{a} \odot (\mathbf{b} \odot \mathbf{c})$.
5) The result of binding is dissimilar to each of its argument hypervectors: $\langle (\mathbf{a} \odot \mathbf{b}), \mathbf{a} \rangle \approx \langle (\mathbf{a} \odot \mathbf{b}), \mathbf{b} \rangle \approx 0$; hence, binding is a "randomizing" operation.
6) Binding preserves similarity: $\langle (\mathbf{c} \odot \mathbf{a}), (\mathbf{c} \odot \mathbf{b}) \rangle = \langle \mathbf{a}, \mathbf{b} \rangle$.

C. Properties of the Superposition Operation

The superposition operation has the following properties:

1) Superposition is invertible: $(\mathbf{a} + \mathbf{b}) + (-\mathbf{b}) = \mathbf{a}$; for thresholded superposition: $\langle [[\mathbf{a} + \mathbf{b}] + (-\mathbf{b})], \mathbf{a} \rangle > 0$.
2) In contrast to binding and permutation operations, the result of superposition $\mathbf{z} = \mathbf{a} + \mathbf{b}$ (often called the superposition hypervector) is similar to each of its argument hypervectors, i.e., the dot product between $\mathbf{z}$ and $\mathbf{a}$ or $\mathbf{b}$ is considerably greater than 0, $\langle \mathbf{z}, \mathbf{a} \rangle \gg 0$, and $\langle \mathbf{z}, \mathbf{b} \rangle \gg 0$.
3) Superposition is commutative: $\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$.
4) Thresholded superposition is approximately associative: $[[\mathbf{a} + \mathbf{b}] + \mathbf{c}] \approx [\mathbf{a} + [\mathbf{b} + \mathbf{c}]]$.

D. Properties of the Permutation Operation

The permutation operation has the following properties:

1) Permutation is invertible: $\rho^{-1}(\rho(\mathbf{a})) = \mathbf{a}$.
2) Permutation distributes over both binding and superposition: $\rho(\mathbf{a} \odot \mathbf{b}) = \rho(\mathbf{a}) \odot \rho(\mathbf{b})$ and $\rho(\mathbf{a} + \mathbf{b}) = \rho(\mathbf{a}) + \rho(\mathbf{b})$.
3) Similar to the binding operation, a random permutation ρ results in a vector that is dissimilar to the argument hypervector: $\langle \rho(\mathbf{a}), \mathbf{a} \rangle \approx 0$; hence, permutation is a "randomizing" operation.
4) Permutation preserves similarity: $\langle \rho(\mathbf{a}), \rho(\mathbf{b}) \rangle = \langle \mathbf{a}, \mathbf{b} \rangle$. ■

Acknowledgment

The authors thank members of the Redwood Center for Theoretical Neuroscience and the Berkeley Wireless Research Center for stimulating discussions. They would also like to thank Ross W. Gayler and Sohum Datta for their in-depth comments on the early versions of this article. Finally, they would like to thank three anonymous reviewers and the editors for their insightful feedback and Linda Rudin for the careful proofreading that contributed to the final shape of this article.

REFERENCES

[1] H. Jaeger, "Towards a generalized theory comprising digital, neuromorphic and unconventional computing," *Neuromorphic Comput. Eng.*, vol. 1, no. 1, pp. 1–38, 2021.
[2] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 1–43, Jul. 2020.
[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017, pp. 1–14.
[4] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.
[5] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[6] X. Lin *et al.*, "All-optical machine learning using diffractive deep neural networks," *Science*, vol. 361, no. 6406, pp. 1004–1008, Sep. 2018.
[7] J. Pei *et al.*, "Towards artificial general intelligence with hybrid Tianjic chip architecture," *Nature*, vol. 572, no. 7767, pp. 106–111, 2019.
[8] N. Imam and T. A. Cleland, "Rapid online learning and robust recall in a neuromorphic olfactory circuit," *Nat. Mach. Intell.*, vol. 2, no. 3, pp. 181–191, Dec. 2020.
[9] M. Davies, "Advancing neuromorphic computing with Loihi: A survey of results and outlook," *Proc. IEEE*, vol. 109, no. 5, pp. 911–934, May 2021.
[10] R. W. Gayler, "Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience," in *Proc. Joint Int. Conf. Cogn. Sci. (ICCS/ASCS)*, 2003, pp. 133–138.
[11] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cogn. Comput.*, vol. 1, no. 2, pp. 139–159, Oct. 2009.
[12] C. Eliasmith *et al.*, "A large-scale model of the functioning brain," *Science*, vol. 338, no. 6111, pp. 1202–1205, 2012.
[13] D. A. Rachkovskij and S. V. Slipchenko, "Similarity-based retrieval with structure-sensitive sparse binary distributed representations," *Comput. Intell.*, vol. 28, no. 1, pp. 106–129, 2012.
[14] B. Emruli, R. W. Gayler, and F. Sandin, "Analogical mapping and inference with binary spatter codes and sparse distributed memory," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Aug. 2013, pp. 1–8.
[15] L. Ge and K. K. Parhi, "Classification using hyperdimensional computing: A review," *IEEE Circuits Syst. Mag.*, vol. 20, no. 2, pp. 30–47, 2nd Quart., 2020.
[16] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *Nature Electron.*, vol. 3, no. 6, pp. 327–337, Jun. 2020.
[17] A. Renner, Y. Sandamirskaya, F. Sommer, and E. P. Frady, "Sparse vector binding on spiking neuromorphic hardware using synaptic delays," in *Proc. Int. Conf. Neuromorphic Syst.*, Jul. 2022, pp. 1–5.
[18] G. Bent, C. Simpkin, Y. Li, and A. Preece, "Hyperdimensional computing using time-to-spike neuromorphic circuits," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2022, pp. 1–8.
[19] D. Marr, *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. New York, NY, USA: W. H. Freeman and Company, 1982.
[20] A. S. G. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, 2015.
[21] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 3645–3650.
[22] A. Rogers. (2019). How the Transformers Broke NLP Leaderboards. [Online]. Available: https://hackingsemantics.xyz/2019/leaderboards/
[23] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.
[24] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018.
[25] E. P. Frady *et al.*, "Neuromorphic nearest neighbor search using Intel's Pohoiki springs," in *Proc. Neuro-Inspired Comput. Elements Workshop (NICE)*, Mar. 2020, pp. 1–10.
[26] H. Li *et al.*, "Hyperdimensional computing with 3D VRRAM in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *IEDM Tech. Dig.*, Dec. 2016, pp. 1–4.
[27] D. Kleyko, E. Osipov, D. D. Silva, U. Wiklund, and D. Alahakoon, "Integer self-organizing maps for digital hardware," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2019, pp. 1–8.

[28] D. Kleyko, E. P. Frady, M. Kheffache, and E. Osipov, "Integer echo state networks: Efficient reservoir computing for digital hardware," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 4, pp. 1688–1701, Apr. 2022.

[29] D. Kleyko, M. Kheffache, E. P. Frady, U. Wiklund, and E. Osipov, "Density encoding enables resource-efficient randomly connected neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 8, pp. 3777–3783, Aug. 2021.

[30] E. P. Frady, D. Kleyko, and F. T. Sommer, "A theory of sequence indexing and working memory in recurrent neural networks," *Neural Comput.*, vol. 30, no. 6, pp. 1449–1513, 2018.

[31] G. Recchia, M. Sahlgren, P. Kanerva, and M. N. Jones, "Encoding sequential information in semantic space models: Comparing holographic reduced representation and random permutation," *Comput. Intell. Neurosci.*, vol. 2015, pp. 1–18, Feb. 2015.

[32] O. J. Räsänen and J. P. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 9, pp. 1878–1889, Sep. 2016.

[33] D. Kleyko, A. Rahimi, D. A. Rachkovskij, E. Osipov, and J. M. Rabaey, "Classification and recall with binary hyperdimensional computing: Tradeoffs in choice of density and mapping characteristics," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 12, pp. 5880–5898, Dec. 2018.

[34] A. Rahimi, P. Kanerva, L. Benini, and J. M. Rabaey, "Efficient biosignal processing using hyperdimensional computing: Network templates for combined learning and classification of ExG signals," *Proc. IEEE*, vol. 107, no. 1, pp. 123–143, Jan. 2019.

[35] D. A. Rachkovskij, "Representation of spatial objects by shift-equivariant similarity-preserving hypervectors," *Neural Comput. Appl.*, pp. 1–17, Sep. 2022.

[36] D. A. Rachkovskij, "Some approaches to analogical mapping with structure sensitive distributed representations," *J. Exp. Theor. Artif. Intell.*, vol. 16, no. 3, pp. 125–145, 2004.

[37] D. A. Rachkovskij, E. M. Kussul, and T. N. Baidyk, "Building a world model with structure-sensitive sparse binary distributed representations," *Biol. Inspired Cogn. Archit.*, vol. 3, pp. 64–86, Jan. 2013.

[38] C. Eliasmith, *How to Build a Brain*. Oxford, U.K.: Oxford Univ. Press, 2013.

[39] D. Kleyko, E. Osipov, R. W. Gayler, A. I. Khan, and A. G. Dyer, "Imitation of honey bees' concept learning processes using vector symbolic architectures," *Biologically Inspired Cogn. Archit.*, vol. 14, pp. 57–72, Oct. 2015.

[40] E. Osipov, D. Kleyko, and A. Legalov, "Associative synthesis of finite state automata model of a controlled object with hyperdimensional computing," in *Proc. 43rd Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Oct. 2017, pp. 3276–3281.

[41] T. Yerxa, A. Anderson, and E. Weiss, "The hyperdimensional stack machine," in *Cognitive Computing*, 2018, pp. 1–2.

[42] P. B. Graben, M. Huber, W. Meyer, R. Römer, and M. Wolff, "Vector symbolic architectures for context-free grammars," *Cogn. Comput.*, vol. 14, no. 2, pp. 733–748, Mar. 2022.

[43] A. Rahimi *et al.*, "High-dimensional computing as a nanoscalable paradigm," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 9, pp. 2508–2521, Sep. 2017.

[44] P. Kanerva, "Computing with high-dimensional vectors," *IEEE Des. Test*, vol. 36, no. 3, pp. 7–14, Jun. 2019.

[45] T. A. Plate, "Estimating analogical similarity by dot-products of holographic reduced representations," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 1994, pp. 1109–1116.

[46] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart, "Distributed representations," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 77–109.

[47] S. J. Thorpe, "Localized versus distributed representations," in *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA, USA: MIT Press, 2003, pp. 643–646.

[48] J. A. Fodor and Z. W. Pylyshyn, "Connectionism and cognitive architecture: A critical analysis," *Cognition*, vol. 28, nos. 1–2, pp. 3–71, Mar. 1988.

[49] E. M. Kussul, D. A. Rachkovskij, and T. N. Baidyk, "On image texture recognition by associative-projective neurocomputer," in *Intelligent Engineering Systems Through Artificial Neural Networks (ANNIE)*, 1991, pp. 453–458.

[50] P. Smolensky, "Tensor product variable binding and the representation of symbolic structures in connectionist systems," *Artif. Intell.*, vol. 46, nos. 1–2, pp. 159–216, Nov. 1990.

[51] T. A. Plate, "Distributed representations and nested compositional structure," Ph.D. thesis, Graduate Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, 1994.

[52] T. A. Plate, "Holographic reduced representations," *IEEE Trans. Neural Netw.*, vol. 6, no. 3, pp. 623–641, May 1995.

[53] T. A. Plate, *Holographic Reduced Representations: Distributed Representation for Cognitive Structures*. Stanford, CA, USA: CSLI, 2003.

[54] R. W. Gayler, "Multiplicative binding, representation operators & analogy," in *Analogy Research: Integration of Theory and Data from the Cognitive, Computational, and Neural Sciences*, 1998, pp. 1–4.

[55] P. Kanerva, "Fully distributed representation," in *Proc. Real World Comput. Symp. (RWC)*, 1997, pp. 358–365.

[56] D. A. Rachkovskij and E. M. Kussul, "Binding and normalization of binary sparse distributed representations by context-dependent thinning," *Neural Comput.*, vol. 13, no. 2, pp. 411–452, 2001.

[57] D. Kleyko, E. Osipov, and D. A. Rachkovskij, "Modification of holographic graph neuron using sparse distributed representations," *Proc. Comput. Sci.*, vol. 88, pp. 39–45, Oct. 2016.

[58] M. Laiho, J. H. Poikonen, P. Kanerva, and E. Lehtonen, "High-dimensional computing with sparse vectors," in *Proc. IEEE Biomed. Circuits Syst. Conf. (BioCAS)*, Oct. 2015, pp. 1–4.

[59] E. P. Frady, D. Kleyko, and F. T. Sommer, "Variable binding for sparse distributed representations: Theory and applications," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Sep. 3, 2021, doi: 10.1109/TNNLS.2021.3105949.

[60] S. I. Gallant and T. W. Okaywe, "Representing objects, relations, and sequences," *Neural Comput.*, vol. 25, no. 8, pp. 2038–2078, 2013.

[61] D. Aerts, M. Czachor, and B. De Moor, "Geometric analogue of holographic reduced representation," *J. Math. Psychol.*, vol. 53, no. 5, pp. 389–398, 2009.

[62] K. Schlegel, P. Neubert, and P. Protzel, "A comparison of vector symbolic architectures," *Artif. Intell. Rev.*, vol. 55, no. 6, pp. 4523–4555, Aug. 2022.

[63] M. Ledoux, *The Concentration of Measure Phenomenon* (Mathematical Surveys and Monographs), no. 89. Providence, RI, USA: Amer. Math. Soc., 2001.

[64] A. N. Gorban and I. Y. Tyukin, "Blessing of dimensionality: Mathematical foundations of the statistical physics of data," *Philos. Trans. Roy. Soc. A, Math., Phys. Eng. Sci.*, vol. 376, no. 2118, pp. 1–18, 2018.

[65] A. Alaghi and J. P. Hayes, "Computing with randomness," *IEEE Spectr.*, vol. 55, no. 3, pp. 46–51, Mar. 2018.

[66] D. A. Rachkovskij, S. V. Slipchenko, E. M. Kussul, and T. N. Baidyk, "Sparse binary distributed encoding of scalars," *J. Autom. Inf. Sci.*, vol. 37, no. 6, pp. 12–23, 2005.

[67] E. Weiss, B. Cheung, and B. A. Olshausen, "A neural architecture for representing and reasoning about spatial relationships," in *Proc. ICLR Workshop*, 2016, pp. 1–4.

[68] B. Komer, T. C. Stewart, A. R. Voelker, and C. Eliasmith, "A neural representation of continuous space using fractional binding," in *Proc. Annu. Meeting Cogn. Sci. Soc. (CogSci)*, 2019, pp. 2038–2043.

[69] P. Sutor, D. Summers-Stay, and Y. Aloimonos, "A computational theory for life-long learning of semantics," in *Proc. Int. Conf. Artif. Gen. Intell. (AGI)*, 2018, pp. 217–226.

[70] A. A. Frolov, D. A. Rachkovskij, and D. Husek, "On informational characteristics of Willshaw-like auto-associative memory," *Neural Netw. World*, vol. 12, no. 2, pp. 141–157, 2002.

[71] A. A. Frolov, D. Husek, and D. A. Rachkovskij, "Time of searching for similar binary vectors in associative memory," *Cybern. Syst. Anal.*, vol. 42, no. 5, pp. 615–623, 2006.

[72] V. I. Gritsenko, D. A. Rachkovskij, A. A. Frolov, R. Gayler, D. Kleyko, and E. Osipov, "Neural distributed autoassociative memories: A survey," *Cybern. Comput. Eng.*, vol. 2, no. 188, pp. 5–35, 2017.

[73] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi, "A survey on hyperdimensional computing aka vector symbolic architectures, part I: Models and data transformations," *ACM Comput. Surv.*, pp. 1–40, May 2022.

[74] K. Greff, S. van Steenkiste, and J. Schmidhuber, "On the binding problem in artificial neural networks," 2020, *arXiv:2012.05208*.

[75] P. Kanerva, "What we mean when we say 'what's the dollar of Mexico?': Prototypes and mapping in concept space," in *Proc. AAAI Fall Symp. Ser.*, 2010, pp. 2–6.

[76] E. P. Frady, S. J. Kent, B. A. Olshausen, and F. T. Sommer, "Resonator networks, 1: An efficient solution for factoring high-dimensional, distributed representations of data structures," *Neural Comput.*, vol. 32, no. 12, pp. 2311–2331, Dec. 2020.

[77] S. J. Kent, E. P. Frady, F. T. Sommer, and B. A. Olshausen, "Resonator networks, 2: Factorization performance and capacity compared to optimization-based methods," *Neural Comput.*, vol. 32, no. 12, pp. 2332–2388, Dec. 2020.

[78] D. Kleyko, A. Rahimi, R. W. Gayler, and E. Osipov, "Autoscaling Bloom filter: Controlling trade-off between true and false positives," *Neural Comput. Appl.*, vol. 32, no. 8, pp. 3675–3684, Apr. 2020.

[79] C. Simpkin *et al.*, "Constructing distributed time-critical applications using cognitive enabled services," *Future Gener. Comput. Syst.*, vol. 100, pp. 70–85, Nov. 2019.

[80] A. Rosato, M. Panella, and D. Kleyko, "Hyperdimensional computing for efficient distributed classification with randomized neural networks," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2021, pp. 1–10.

[81] P. Jakimovski, H. R. Schmidtke, S. Sigg, L. W. F. Chaves, and M. Beigl, "Collective communication for dense sensing environments," *J. Ambient Intell. Smart Environ.*, vol. 4, no. 2, pp. 123–134, Mar. 2012.

[82] D. Kleyko, N. Lyamin, E. Osipov, and L. Riliskis, "Dependable MAC layer architecture based on holographic data representation using hyper-dimensional binary spatter codes," in *Multiple Access Communications* (Lecture Notes in Computer Science), vol. 7642. Springer, 2012, pp. 134–145.

[83] H.-S. Kim, "HDM: Hyper-dimensional modulation for robust low-power communications," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2018, pp. 1–6.

[84] A. Joshi, J. T. Halseth, and P. Kanerva, "Language geometry using random indexing," in *Proc. Int. Symp. Quantum Interact. (QI)*, 2016, pp. 265–274.

[85] S. Levy, S. Bajracharya, and R. W. Gayler, "Learning behavior hierarchies via high-dimensional sensor projection," in *Proc. 27th AAAI Conf. Artif. Intell. (AAAI)*, 2013, pp. 1–4.

[86] P. Neubert, S. Schubert, and P. Protzel, "An introduction to hyperdimensional computing for robotics," *Künstliche Intelligenz*, vol. 33, no. 4, pp. 319–330, Dec. 2019.

[87] A. Mitrokhin, P. Sutor, C. Fermüller, and Y. Aloimonos, "Learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception," *Sci. Robot.*, vol. 4, no. 30, pp. 1–10, May 2019.

[88] D. Kleyko, R. W. Gayler, and E. Osipov, "Commentaries on 'learning sensorimotor control with neuromorphic sensors: Toward hyperdimensional active perception' [science robotics vol. 4 issue 30 (2019) 1–10," 2020, *arXiv:2003.11458*.

[89] M. Hersche, E. M. Rella, A. Di Mauro, L. Benini, and A. Rahimi, "Integrating event-based dynamic vision sensors with sparse hyperdimensional computing: A low-power accelerator with online learning capability," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, Aug. 2020, pp. 169–174.

[90] D. Kleyko, E. Osipov, and U. Wiklund, "A hyperdimensional computing framework for analysis of cardiorespiratory synchronization during paced deep breathing," *IEEE Access*, vol. 7, pp. 34403–34415, 2019.

[91] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey, "Hyperdimensional biosignal processing: A case study for EMG-based hand gesture recognition," in *Proc. IEEE Int. Conf. Rebooting Comput. (ICRC)*, Oct. 2016, pp. 1–8.

[92] A. Burrello, K. Schindler, L. Benini, and A. Rahimi, "Hyperdimensional computing with local binary patterns: One-shot learning of seizure onset and identification of ictogenic brain regions using short-time iEEG recordings," *IEEE Trans. Biomed. Eng.*, vol. 67, no. 2, pp. 601–613, Feb. 2020.

[93] O. Rasanen and S. Kakouros, "Modeling dependencies in multiple parallel data streams with hyperdimensional computing," *IEEE Signal Process. Lett.*, vol. 21, no. 7, pp. 899–903, Jul. 2014.

[94] D. Kleyko, E. Osipov, N. Papakonstantinou, and V. Vyatkin, "Hyperdimensional computing in industrial systems: The use-case of distributed fault isolation in a power plant," *IEEE Access*, vol. 6, pp. 30766–30777, 2018.

[95] C. Diao, D. Kleyko, J. M. Rabaey, and B. A. Olshausen, "Generalized learning vector quantization for classification in randomized neural networks and hyperdimensional computing," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2021, pp. 1–9.

[96] J. Neumann, "Learning the systematic transformation of holographic reduced representations," *Cogn. Syst. Res.*, vol. 3, no. 2, pp. 227–235, Jun. 2002.

[97] T. A. Plate, "Structure matching and transformation with distributed representations," in *Connectionist-Symbolic Integration*, 1997, pp. 1–19.

[98] P. Kanerva, "Large patterns make great symbols: An example of learning from example," in *Proc. Int. Workshop Hybrid Neural Syst.*, in Lecture Notes in Computer Science, vol. 1778, 2000, pp. 194–203.

[99] E. Kussul, D. Rachkovskij, and D. Wunsch, "The random subspace coarse coding scheme for real-valued vectors," in *Proc. Int. Joint Conf. Neural Netw.*, vol. 1, 1999, pp. 450–455.

[100] D. A. Rachkovskij, "Formation of similarity-reflecting binary vectors with random binary projections," *Cybern. Syst. Anal.*, vol. 51, no. 2, pp. 313–323, 2015.

[101] D. Widdows and T. Cohen, "Reasoning with vectors: A continuous model for fast robust inference," *Logic J. IGPL*, vol. 23, no. 2, pp. 141–173, 2015.

[102] E. P. Frady, D. Kleyko, C. J. Kymn, B. A. Olshausen, and F. T. Sommer, "Computing on functions using randomized vector representations," 2021, *arXiv:2109.03429*.

[103] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[104] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[105] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 1, pp. 131–155, Feb. 2012.

[106] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[107] A. Burrello, L. Cavigelli, K. Schindler, L. Benini, and A. Rahimi, "Laelaps: An energy-efficient seizure detection algorithm from long-term human iEEG recordings without false alarms," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 752–757.

[108] P. Alonso, K. Shridhar, D. Kleyko, E. Osipov, and M. Liwicki, "HyperEmbed: Tradeoffs between resources and performance in NLP tasks with hyperdimensional computing enabled embedding of n-gram statistics," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2021, pp. 1–9.

[109] K. Shridhar, H. Jain, A. Agarwal, and D. Kleyko, "End to end binarized neural networks for text classification," in *Proc. Workshop Simple Efficient Natural Lang. Process.*, 2020, pp. 29–34.

[110] E. M. Kussul and T. N. Baidyk, "On information encoding in associative-projective neural networks," V. M. Glushkov Inst. Cybern., Kyiv, Ukraine, Tech. Rep., 93–3, 1993.

[111] T. A. Plate, "Networks which learn to store variable-length sequences in a fixed set of unit activations," Univ. British Columbia, Vancouver, BC, Canada, Tech. Rep., 1995, pp. 1–19.

[112] M. Sahlgren, A. Holst, and P. Kanerva, "Permutations as a means to encode order in word space," in *Proc. Annu. Meeting Cogn. Sci. Soc. (CogSci)*, 2008, pp. 1300–1305.

[113] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Sov. Phys.-Dokl.*, vol. 10, no. 8, pp. 707–710, 1966.

[114] A. M. Sokolov, "Vector representations for efficient comparison and search for similar strings," *Cybern. Syst. Anal.*, vol. 43, no. 4, pp. 484–498, Jul. 2007.

[115] T. Hannagan, E. Dupoux, and A. Christophe, "Holographic string encoding," *Cogn. Sci.*, vol. 35, no. 1, pp. 79–118, Jan. 2011.

[116] D. Kleyko and E. Osipov, "On bidirectional transitions between localist and distributed representations: The case of common substrings search using vector symbolic architecture," *Proc. Comput. Sci.*, vol. 41, pp. 104–113, Dec. 2014.

[117] T. Cohen, D. Widdows, M. Wahle, and R. W. Schvaneveldt, "Orthogonality and orthography: Introducing measured distance into semantic space," in *Proc. Int. Symp. Quantum Interact. (QI)* in Lecture Notes in Computer Science, vol. 8369, 2013, pp. 34–46.

[118] D. A. Rachkovskij, "Shift-equivariant similarity-preserving hypervector representations of sequences," 2021, *arXiv:2112.15475*.

[119] D. A. Rachkovskij and D. Kleyko, "Recursive binding for similarity-preserving hypervector representations of sequences," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2022, pp. 1–8.

[120] X. Choo and C. Eliasmith, "A spiking neuron model of serial-order recall," in *Proc. Annu. Meeting Cogn. Sci. Soc. (CogSci)*, 2010, pp. 2188–2193.

[121] P. Blouw and C. Eliasmith, "A neurally plausible encoding of word order information into a semantic vector space," in *Proc. Annu. Meeting Cogn. Sci. Soc. (CogSci)*, 2013, pp. 1905–1910.

[122] M. A. Kelly, N. Arora, R. L. West, and D. Reitter, "Holographic declarative memory: Distributional semantics as the architecture of memory," *Cogn. Sci.*, vol. 44, no. 11, pp. 1–34, Nov. 2020.

[123] J. Gosmann and C. Eliasmith, "CUE: A unified spiking neuron model of short-term and long-term memory," *Psychol. Rev.*, vol. 128, no. 1, pp. 104–124, Jan. 2021.

[124] S. Reimann, "The algebra of cognitive states: Towards modelling the serial position curve," in *Proc. Int. Conf. Cogn. Modeling (ICCM)*, 2021, pp. 1–7.

[125] R. Calmus, B. Wilson, Y. Kikuchi, and C. I. Petkov, "Structured sequence processing and combinatorial binding: Neurobiologically and computationally informed hypotheses," *Philos. Trans. Roy. Soc. B*, vol. 375, no. 1791, pp. 1–13, 2019.

[126] Y. Kim, M. Imani, N. Moshiri, and T. Rosing, "GenieHD: Efficient DNA pattern matching accelerator using hyperdimensional computing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 115–120.

[127] D. Kleyko, E. Osipov, and R. W. Gayler, "Recognizing permuted words with vector symbolic architectures: A Cambridge test for machines," *Proc. Comput. Sci.*, vol. 88, pp. 169–175, Dec. 2016.

[128] K. Schlegel, P. Neubert, and P. Protzel, "HDC-MiniROCKET: Explicit time encoding in time series classification with hyperdimensional computing," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2022, pp. 1–8.

[129] F. R. Najafabadi, A. Rahimi, P. Kanerva, and J. M. Rabaey, "Hyperdimensional computing for text classification," in *Proc. Design, Autom. Test Eur. Conf. (DATE)*, 2016, p. 1.

[130] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, Aug. 2016, pp. 64–69.

[131] D. Kleyko, E. Osipov, D. D. Silva, U. Wiklund, V. Vyatkin, and D. Alahakoon, "Distributed representation of n-gram statistics for boosting self-organizing maps with hyperdimensional computing," in *Proc. Int. Andrei Ershov Memorial Conf. Perspect. Syst. Inform.*, 2019, pp. 64–79.

[132] T. Bandaragoda, D. De Silva, D. Kleyko, E. Osipov, U. Wiklund, and D. Alahakoon, "Trajectory clustering of road traffic in urban environments using incremental machine learning in combination with hyperdimensional computing," in *Proc. IEEE Intell. Transp. Syst. Conf. (ITSC)*, Oct. 2019, pp. 1664–1670.

[133] R. W. Gayler and S. D. Levy, "A distributed basis for analogical mapping," in *Proc. New Frontiers Anal. Res., 2nd Int. Conf. Anal.*, 2009, pp. 165–174.

[134] J. K. Guo, D. Van Brackle, N. Lofaso, and M. O. Hofmann, "Vector representation for sub-graph encoding to resolve entities," *Proc. Comput. Sci.*, vol. 95, pp. 327–334, Jan. 2016.

[135] Y. Ma, M. Hildebrandt, V. Tresp, and S. Baier, "Holistic representations for memorization and inference," in *Proc. Conf. Uncertainty Artif. Intell. (UAI)*, 2018, pp. 1–11.

[136] M. Nickel, L. Rosasco, and T. Poggio, "Holographic embeddings of knowledge graphs," in *Proc. AAAI Conf. Artif. Intell.*, 2016, pp. 1955–1961.

[137] F. Qiu, "Graph embeddings via tensor products and approximately orthonormal codes," 2022, *arXiv:2208.10917*.

[138] T. C. Stewart, X. Choo, and C. Eliasmith, "Sentence processing in spiking neurons: A biologically plausible left-corner parser," in *Proc. Annu. Meeting Cogn. Sci. Soc. (CogSci)*, 2014, pp. 1533–1538.

[139] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM J. Res. Develop.*, vol. 3, no. 2, pp. 114–125, 1959.

[140] E. Crawford, M. Gingerich, and C. Eliasmith, "Biologically plausible, human-scale knowledge representation," *Cogn. Sci.*, vol. 40, no. 4, pp. 782–821, 2016.

[141] B. Ghazi, R. Panigrahy, and J. Wang, "Recursive sketches for modular deep learning," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2019, pp. 2211–2220.

[142] S. I. Gallant, "Orthogonal matrices for MBAT vector symbolic architectures, and a 'soft' VSA representation for JSON," 2022,

arXiv:2202.04771.

[143] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[144] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Develop.*, vol. 31, no. 2, pp. 249–260, Mar. 1987.

[145] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, Jul. 1977.

[146] D. V. Pashchenko, D. A. Trokoz, A. I. Martyshkin, M. P. Sinev, and B. L. Svistunov, "Search for a substring of characters using the theory of non-deterministic finite automata and vector-character architecture," *Bull. Electr. Eng. Informat.*, vol. 9, no. 3, pp. 1238–1250, Jun. 2020.

[147] G. Karunaratne *et al.*, "Robust high-dimensional memory-augmented neural networks," *Nature Commun.*, vol. 12, no. 1, pp. 1–12, Dec. 2021.

[148] E. P. Frady, S. J. Kent, P. Kanerva, B. A. Olshausen, and F. T. Sommer, "Cognitive neural systems for disentangling compositions," in *Cognitive Computing*, 2018, pp. 1–3.

[149] D. Kleyko *et al.*, "Integer factorization with compositional distributed representations," in *Proc. Neuro-Inspired Comput. Elements Conf. (NICE)*, Mar. 2022, pp. 73–80.

[150] E. P. Frady, D. Kleyko, C. J. Kymn, B. A. Olshausen, and F. T. Sommer, "Computing on functions using randomized vector representations (in brief)," in *Proc. Neuro-Inspired Comput. Elements Conf. (NICE)*, Mar. 2022, pp. 115–122.

[151] P. M. Furlong and C. Eliasmith, "Fractional binding in vector symbolic architectures as quasi-probability statements," in *Proc. Annu. Meeting Cogn. Sci. Soc. (CogSci)*, 2022, pp. 259–266.

[152] P. M. Furlong, T. C. Stewart, and C. Eliasmith, "Fractional binding in vector symbolic representations for efficient mutual information exploration," in *Proc. ICRA Workshop, Towards Curious Robots, Mod. Approaches Intrinsically-Motivated Intell. Behav.*, 2022, pp. 1–5.

[153] M. Hersche, M. Zeqiri, L. Benini, A. Sebastian, and A. Rahimi, "A neuro-vector-symbolic architecture for solving Raven's progressive matrices," 2022, *arXiv:2203.04571*.

[154] B. Komer and C. Eliasmith, "Efficient navigation using a scalable, biologically inspired spatial representation," in *Proc. Annu. Meeting Cogn. Sci. Soc. (CogSci)*, 2020, pp. 1532–1538.

[155] T. Lu, A. R. Voelker, B. Komer, and C. Eliasmith, "Representing spatial relations with fractional binding," in *Proc. Annu. Meeting Cognit. Sci. Soc. (CogSci)*, 2019, pp. 2214–2220.

[156] B. Cheung, A. Terekhov, Y. Chen, P. Agrawal, and B. Olshausen, "Superposition of many models into one," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2019, pp. 10868–10877.

[157] Z. Zou, H. Alimohamadi, F. Imani, Y. Kim, and M. Imani, "Spiking hyperdimensional network: Neuromorphic models integrated with memory-inspired framework," 2021, *arXiv:2110.00214*.

[158] P. Neubert and S. Schubert, "Hyperdimensional computing as a framework for systematic aggregation of image descriptors," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2021, pp. 16938–16947.

[159] P. Neubert, S. Schubert, K. Schlegel, and P. Protzel, "Vector semantic representations as descriptors for visual place recognition," in *Robotics: Science and Systems*, Jul. 2021, pp. 1–11.

[160] A. Ganesan *et al.*, "Learning with holographic reduced representations," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2021, pp. 1–15.

[161] S. Datta, R. A. G. Antonio, A. R. S. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric IoT," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 3, pp. 439–452, Sep. 2019.

[162] M. Eggimann, A. Rahimi, and L. Benini, "A 5 μW standard cell memory-based configurable hyperdimensional computing accelerator for always-on smart sensing," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 10, pp. 4116–4128, Oct. 2021.

[163] F. Montagna, A. Rahimi, S. Benatti, D. Rossi, and L. Benini, "PULP-HD: Accelerating brain-inspired high-dimensional computing on a parallel ultra-low power platform," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[164] T. Wu *et al.*, "Brain-inspired computing exploiting carbon nanotube FETs and resistive RAM: Hyperdimensional computing case study," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Apr. 2018, pp. 492–493.

[165] T. F. Wu *et al.*, "Hyperdimensional computing exploiting carbon nanotube FETs, resistive RAM, and their monolithic 3D integration," *IEEE J. Solid-State Circuits*, vol. 53, no. 11, pp. 3183–3196, Nov. 2018.

[166] A. Moin *et al.*, "A wearable biosensing system with in-sensor adaptive machine learning for hand gesture recognition," *Nature Electron.*, vol. 4, pp. 54–63, Jan. 2021.

[167] C. Eliasmith and C. H. Anderson, *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. Cambridge, MA, USA: MIT Press, 2003.

[168] T. Bekolay *et al.*, "Nengo: A Python tool for building large-scale functional brain models," *Frontiers Neuroinf.*, vol. 7, no. 48, pp. 1–13, Jan. 2014.

[169] G. Csaba and W. Porod, "Coupled oscillators for computing: A review and perspective," *Appl. Phys. Rev.*, vol. 7, no. 1, pp. 1–19, 2020.

[170] E. P. Frady and F. T. Sommer, "Robust computation with rhythmic spike patterns," *Proc. Nat. Acad. Sci. USA*, vol. 116, no. 36, pp. 18050–18059, Sep. 2019.

[171] G. Palm and T. Bonhoeffer, "Parallel processing for associative and neuronal networks," *Biol. Cybern.*, vol. 51, no. 3, pp. 201–204, Dec. 1984.

[172] D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins, "Non-holographic associative memory," *Nature*, vol. 222, no. 5197, pp. 960–962, Jun. 1969.

[173] G. Palm, "On associative memory," *Biol. Cybern.*, vol. 36, no. 1, pp. 19–31, 1980.

[174] G. Palm and F. T. Sommer, "Information capacity in recurrent McCulloch–Pitts networks with sparsely coded memory states," *Netw., Comput. Neural Syst.*, vol. 3, no. 2, pp. 177–186, Jan. 1992.

[175] F. T. Sommer and P. Dayan, "Bayesian retrieval in associative memories with storage errors," *IEEE Trans. Neural Netw.*, vol. 9, no. 4, pp. 705–713, Jul. 1998.

[176] M. Stimberg, R. Brette, and D. F. Goodman, "Brian 2, an intuitive and efficient neural simulator," *ELife*, vol. 8, pp. 1–41, Aug. 2019.

[177] I. Nunes, M. Heddes, T. Givargis, A. Nicolau, and A. Veidenbaum, "GraphHD: Efficient graph classification using hyperdimensional computing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2022, pp. 1485–1490.

[178] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 15, no. 4, pp. 1–25, Dec. 2019.

[179] D. Kleyko, E. P. Frady, and F. T. Sommer, "Cellular automata can reduce memory requirements of collective-state computing," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 6, pp. 2701–2713, Jun. 2022.

[180] A. Menon, D. Sun, M. Aristio, H. Liew, K. Lee, and J. M. Rabaey, "A highly energy-efficient hyperdimensional computing processor for wearable multi-modal classification," in *Proc. IEEE Biomed. Circuits Syst. Conf. (BioCAS)*, Oct. 2021, pp. 1–4.

[181] A. Menon *et al.*, "A highly energy-efficient hyperdimensional computing processor for biosignal classification," *IEEE Trans. Biomed. Circuits Syst.*, early access, Jul. 1, 2022, doi: 10.1109/TBCAS.2022.3187944.

[182] M. Hersche, G. Karunaratne, G. Cherubini, L. Benini, A. Sebastian, and A. Rahimi, "Constrained few-shot class-incremental learning," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2022, pp. 1–19.

[183] F. van der Velde and M. de Kamps, "Neural blackboard architectures of combinatorial structures in cognition," *Behav. Brain Sci.*, vol. 29, no. 1, pp. 37–70, Feb. 2006.

[184] R. W. Gayler, "Vector symbolic architectures are a viable alternative for Jackendoff's challenges," *Behav. Brain Sci.*, vol. 29, no. 1, pp. 78–79, Feb. 2006.

[185] D. Kleyko, D. A. Rachkovskij, E. Osipov, and A. Rahimi, "A survey on hyperdimensional computing aka vector symbolic architectures, part II: Applications, cognitive models, and challenges," *ACM Comput. Surv.*, pp. 1–52, Aug. 2022.

[186] P. Kanerva, J. Kristoferson, and A. Holst, "Random indexing of text samples for latent semantic analysis," in *Proc. Annu. Meeting Cognit. Sci. Soc. (CogSci)*, 2000, p. 1036.

[187] M. N. Jones and D. J. K. Mewhort, "Representing word meaning and order information in a composite holographic lexicon," *Psychol. Rev.*, vol. 114, no. 1, pp. 1–37, 2007.

[188] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2013, pp. 1–9.

[189] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1532–1543.

[190] P. Blouw, E. Solodkin, P. Thagard, and C. Eliasmith, "Concepts as semantic pointers: A framework and computational model," *Cogn. Sci.*, vol. 40, no. 5, pp. 1128–1162, Jul. 2016.

[191] M. A. Kelly, D. J. K. Mewhort, and R. L. West, "The memory tesseract: Mathematical equivalence between composite and separate storage memory models," *J. Math. Psychol.*, vol. 77, pp. 142–155, Apr. 2017.

[192] D. Kleyko, G. Karunaratne, J. M. Rabaey, A. Sebastian, and A. Rahimi, "Generalized key-value memory to flexibly adjust redundancy in memory-augmented networks," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Mar. 25, 2022, doi: 10.1109/TNNLS.2022.3159445.

[193] A. Thomas, S. Dasgupta, and T. Rosing, "A theoretical perspective on hyperdimensional computing," *J. Artif. Intell. Res.*, vol. 72, pp. 215–249, Oct. 2021.

[194] B. Emruli, F. Sandin, and J. Delsing, "Vector space architecture for emergent interoperability of systems by learning from demonstration," *Biologically Inspired Cognit. Archit.*, vol. 11, pp. 53–64, Jan. 2015.

[195] J. Steinberg and H. Sompolinsky, "Associative memory of structured knowledge," *BioRxiv*, pp. 1–27, Jan. 2022.

[196] T. C. Stewart, X. Choo, and C. Eliasmith, "Symbolic reasoning in spiking neurons: A model of the cortex/basal ganglia/thalamus loop," in *Proc. Annu. Meeting Cognit. Sci. Soc. (CogSci)*, 2010, pp. 1100–1105.

[197] A. Renner *et al.*, "Neuromorphic visual scene understanding with resonator networks," 2022, *arXiv:2208.12880*.

[198] V. K. Mansinghka, "Natively probabilistic computation," Ph.D. dissertation, Dept. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2009.

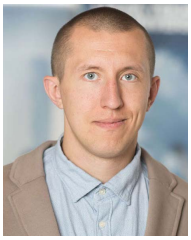
[199] G. Orbán, P. Berkes, J. Fiser, and M. Lengyel, "Neural variability and sampling-based probabilistic representations in the visual cortex," *Neuron*, vol. 92, no. 2, pp. 530–543, Oct. 2016.

[200] C. H. Papadimitriou, S. S. Vempala, D. Mitropolsky, M. Collins, and W. Maass, "Brain computation by assemblies of neurons," *Proc. Nat. Acad. Sci. USA*, vol. 117, no. 25, pp. 14464–14472, Jun. 2020.

[201] G. Schöner, J. P. Spencer, and T. D. R. Group, *Dynamic Thinking: A Primer on Dynamic Field*

Theory. New York, NY, USA: Oxford Univ. Press, 2016.
[202] Y. Zhang *et al.*, "A system hierarchy for brain-inspired computing," *Nature*, vol. 586, no. 7829, pp. 378–384, Oct. 2020.
[203] M. Cook, "Universality in elementary cellular automata," *Complex Syst.*, vol. 15, no. 1, pp. 1–40, 2004.
[204] T. Neary and D. Woods, "Small weakly universal Turing machines," in *Proc. Int. Symp. Fundamentals Comput. Theory (FCT)*, 2009, pp. 262–273.
[205] S. Wolfram, *A New Kind of Science*. Champaign, IL, USA: Wolfram Media, 2002.
[206] H. T. Siegelmann and E. D. Sontag, "Turing computability with neural nets," *Appl. Math. Lett.*, vol. 4, no. 6, pp. 77–80, 1991.
[207] J. Perez, J. Marinkovic, and P. Barcelo, "On the Turing completeness of modern neural network architectures," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2019, pp. 1–36.
[208] P. B. Graben and R. Potthast, "Implementing Turing machines in dynamic field architectures," 2012, *arXiv:1204.5462*.
[209] J. Kwisthout and N. Donselaar, "On the computational power and complexity of spiking neural networks," in *Proc. Neuro-Inspired Comput. Elements Workshop*, Mar. 2020, pp. 1–7.
[210] W. Maass, "Lower bounds for the computational power of networks of spiking neurons," *Neural Comput.*, vol. 8, no. 1, pp. 1–40, Jan. 1996.
[211] A. Demidovskij, "Encoding and decoding of recursive structures in neural-symbolic systems," *Opt. Memory Neural Netw.*, vol. 30, no. 1, pp. 37–50, Jan. 2021.
[212] C. P. Dolan and P. Smolensky, "Tensor product production system: A modular architecture and representation," *Connection Sci.*, vol. 1, no. 1, pp. 53–68, Jan. 1989.
[213] Y. Knight, M. Spranger, and L. Steels, "A vector representation of fluid construction grammar using holographic reduced representations," in *Proc. EuroAsianPacific Joint Conf. Cognit. Sci. (EAPCogSci)*, Apr. 2015, pp. 560–565.

ABOUT THE AUTHORS



Denis Kleyko (Member, IEEE) received the B.S. degree (Hons.) in telecommunication systems and the M.S. degree (Hons.) in information systems from the Siberian State University of Telecommunications and Informatics, Novosibirsk, Russia, in 2011 and 2013, respectively, and the Ph.D. degree in computer science from the Luleå University of Technology, Luleå, Sweden, in 2018.

He is currently a Postdoctoral Researcher on a joint appointment between the Redwood Center for Theoretical Neuroscience, University of California at Berkeley, Berkeley, CA, USA, and the Intelligent Systems Laboratory, Research Institutes of Sweden, Kista, Sweden. His current research interests include machine learning, reservoir computing, and vector symbolic architectures/hyperdimensional computing.

Mike Davies (Member, IEEE) received the B.S. and M.S. degrees from the California Institute of Technology (Caltech), Pasadena, CA, USA, in 1998 and 2000, respectively.

He was a Founding Employee with Fulcrum Microsystems, Calabasas, CA, USA, and the Director of its Silicon Engineering Group until Intel's acquisition of Fulcrum in 2011. He led the development of four generations of low-latency, highly integrated Ethernet switches using Fulcrum's proprietary asynchronous design methodology. Since 2014, he has been researching neuromorphic circuits, architectures, and algorithms at Intel Labs, Intel Corporation, Santa Clara, CA, USA. He is currently a Senior Principal Engineer and the Director of the Neuromorphic Computing Laboratory, Intel Corporation.

Edward Paxon Frady received the B.S. degree in computation and neural systems from the California Institute of Technology (Caltech), Pasadena, CA, USA, in 2008, and the Ph.D. degree in neuroscience from the University of California at San Diego, La Jolla, CA, USA, in 2014.

He is currently a Research Lead with the Neuromorphic Computing Laboratory, Intel Labs, Santa Clara, CA, USA. His research interests include neuromorphic engineering, vector symbolic architectures/hyperdimensional computing, and machine learning.

Pentti Kanerva received the Ph.D. degree in philosophy from Stanford University, Stanford, CA, USA, in 1984.

He was involved in designing and building computer systems for 20 years, and he has over 30 years of research into understanding brains in computing terms. He has held research positions at the NASA Ames Research Center, Mountain View, CA, USA; the Swedish Institute of Computer Science, Kista, Sweden; and the Redwood Neuroscience Institute, Berkeley, CA, USA. He is currently a Researcher with the Redwood Center for Theoretical Neuroscience, University of California at Berkeley, Berkeley. His thesis was published in the book *Sparse Distributed Memory* (MIT Press). His subsequent research includes binary spatter code, random indexing, and hyperdimensional computing.

Spencer J. Kent received the B.S. degree in electrical and computer engineering from Rice University, Houston, TX, USA, in 2015, and the Ph.D. degree in electrical engineering and computer sciences from the University of California at Berkeley, Berkeley, CA, USA, in 2020.

He is currently leading an early stage company that builds software and actuarial models for the insurance industry.

Dr. Kent was a recipient of a National Science Foundation Graduate Research Fellowship for his work on vector symbolic architectures and image compression.

Bruno A. Olshausen received the B.S. and M.S. degrees in electrical engineering from Stanford University, Palo Alto, CA, USA, in 1986 and 1987, respectively, and the Ph.D. degree in computation and neural systems from the California Institute of Technology (Caltech), Pasadena, CA, USA, in 1994.

From 1996 to 2005, he was an Assistant Professor and subsequently an Associate Professor with the Departments of Psychology and Neurobiology, Physiology, and Behavior, University of California at Davis, Davis, CA, USA. Since 2005, he has been with the University of California at Berkeley, Berkeley, CA, USA, where he is currently a Professor of neuroscience and optometry. He also serves as the Director of the Redwood Center for Theoretical Neuroscience, University of California at Berkeley, which is developing mathematical and computational models of brain function. His research aims to understand the information processing strategies employed by the brain for doing tasks, such as object recognition and scene analysis. The aim of this work is not only to advance our understanding of the brain but also to discover new algorithms for scene analysis based on how brains work.

Evgeny Osipov received the Ph.D. degree in computer science from the University of Basel, Basel, Switzerland, in 2005.

He is currently a Full Professor of dependable communication and computation systems with the Department of Computer Science and Electrical Engineering, Luleå University of Technology, Luleå, Sweden. His research interests are in novel computational models for unconventional computer architectures. His current research focuses on hyperdimensional computing and vector symbolic architectures.

Jan M. Rabaey (Life Fellow, IEEE) became the CTO of the System-Technology Co-Optimization Division, imec, Leuven, Belgium, in 2019. He has twice served as the Electrical Engineering Division Chair at the University of California at Berkeley (UC Berkeley), Berkeley, CA, USA. He is currently a Professor with the Department of Electrical Engineering and Computer Sciences (EECS), Graduate School, UC Berkeley, after being the holder of the Donald O. Pederson Distinguished Professorship at the same institute for over 30 years. He is the Founding Director of the Berkeley Wireless Research Center (BWRC) and the Berkeley Ubiquitous SwarmLab, UC Berkeley. He has been involved in a broad variety of startup ventures. He has made high-impact contributions to a number of fields, including low-power integrated circuits, advanced wireless systems, mobile devices, sensor networks, and ubiquitous computing. He is the primary author of the influential *Digital Integrated Circuits: A Design Perspective* textbook that has served to educate hundreds of thousands of students all over the world. His current interests include the conception of the next-generation distributed systems and the exploration of the interaction between the cyber and the biological worlds.

Dr. Rabaey was a recipient of numerous awards.

Dmitri A. Rachkovskij received the M.S. degree in radiophysics from Rostov State University, Rostov, Russia, in 1983, the Ph.D. degree from the V. M. Glushkov Institute of Cybernetics, Kyiv, Ukraine, in 1990, and the D.Sc. degree in artificial intelligence from the International Research and Training Center for Information Technologies and Systems (IRTC ITS), National Academy of Sciences of Ukraine and Ministry of Education and Science of Ukraine, Kyiv, in 2008.

In 1987, he joined the Cybernetics Center, and IRTC ITS in 1997, where he is currently a Chief Research Scientist. In 2019, he received the title of a Full Professor. Since 2022, he has been a Visiting Professor with the Luleå University of Technology, Luleå, Sweden. He led over 20 projects and has authored or coauthored more than 80 refereed publications, including those in high-impact journals. His research is connected with the domain of artificial intelligence and neural networks. He was also involved extensively in the areas of pattern recognition, software and hardware neurocomputers, and micromechanics. His current research interests include distributed representations, similarity search, analogical reasoning, and distributed autoassociative memories.

Abbas Rahimi received the B.S. degree in computer engineering from the University of Tehran, Tehran, Iran, in 2010, and the M.S. and Ph.D. degrees in computer science and engineering from the University of California at San Diego, La Jolla, CA, USA, in 2013 and 2015, respectively.

Since then, until 2020, he held postdoctoral research positions at the University of California at Berkeley, Berkeley, CA, USA, and ETH Zürich, Zürich, Switzerland. In 2020, he joined the IBM Research–Zürich Laboratory, Rüschlikon, Switzerland, as a Research Staff Member.

Dr. Rahimi received the 2015 Outstanding Dissertation Award in the area of "new directions in embedded system design and embedded software" from the European Design and Automation Association and the ETH Zürich Postdoctoral Fellowship in 2017. He was a co-recipient of the Best Paper Nominations at the ACM/IEEE Design Automation Conference (DAC) in 2013 and the Design, Automation Test in Europe Conference Exhibition (DATE) in 2019, the Best Paper Awards at the EAI International Conference on Bio-inspired Information and Communications Technologies (BICT) in 2017 and the IEEE Biomedical Circuits and Systems Conference (BioCAS) in 2018, and the IBM's Pat Goldberg Memorial Best Paper Award in 2020.

Friedrich T. Sommer received the Diploma degree in physics from the University of Tübingen, Tübingen, Germany, in 1987, the Ph.D. degree from the University of Düsseldorf, Düsseldorf, Germany, in 1993, and the Habilitation degree in computer science from Ulm University, Ulm, Germany, in 2002.

He is currently an Adjunct Professor with the Redwood Center for Theoretical Neuroscience, University of California at Berkeley, Berkeley, CA, USA, and a Researcher with the Neuromorphic Computing Laboratory, Intel Labs, Santa Clara, CA, USA. His research interests include neuromorphic engineering, vector symbolic architectures/hyperdimensional computing, and machine learning.