Heddes *et al. Journal of Big Data* (2024) 11:145
https://doi.org/10.1186/s40537-024-01010-8

Journal of Big Data

RESEARCH Open Access

Hyperdimensional computing: a framework for stochastic computation and symbolic AI

Mike Heddes[1*], Igor Nunes[1], Tony Givargis[1], Alexandru Nicolau[1] and Alex Veidenbaum[1]

*Correspondence:
mheddes@uci.edu

[1] Department of Computer Science, University of California, Irvine, Irvine, CA 92617, USA

Abstract

Hyperdimensional Computing (HDC), also known as Vector Symbolic Architectures (VSA), is a neuro-inspired computing framework that exploits high-dimensional random vector spaces. HDC uses extremely parallelizable arithmetic to provide computational solutions that balance accuracy, efficiency and robustness. The majority of current HDC research focuses on the learning capabilities of these high-dimensional spaces. However, a tangential research direction investigates the properties of these high-dimensional spaces more generally as a probabilistic model for computation. In this manuscript, we provide an approachable, yet thorough, survey of the components of HDC. To highlight the dual use of HDC, we provide an in-depth analysis of two vastly different applications. The first uses HDC in a learning setting to classify graphs. Graphs are among the most important forms of information representation, and graph learning in IoT and sensor networks introduces challenges because of the limited compute capabilities. Compared to the state-of-the-art Graph Neural Networks, our proposed method achieves comparable accuracy, while training and inference times are on average 14.6× and 2.0× faster, respectively. Secondly, we analyse a dynamic hash table that uses a novel hypervector type called circular-hypervectors to map requests to a dynamic set of resources. The proposed hyperdimensional hashing method has the efficiency to be deployed in large systems. Moreover, our approach remains unaffected by a realistic level of memory errors which causes significant mismatches for existing methods.

Keywords: Hyperdimensional computing, Vector symbolic architectures, Basis hypervectors, Graph classification, Dynamic hash table

Introduction

The origins of Hyperdimensional Computing (HDC) [46], also known as Vector Symbolic Architectures (VSA) [22], dates back to the 80's. Notable early work include Hinton [36] and Kanerva [45] with their development of the theory of distributed representations and distributed memory, respectively. The field has gained momentum more recently and is receiving increasing interest from the broader artificial intelligence (AI) community. Current applications of HDC range from natural language processing [91], gesture and voice recognition [41, 89], to implementing finite-state automata [81, 118].

The theory of HDC/VSA emerged from comparative studies of computing in animal brains and computer logic circuits [46]. HDC can represent a "concept space" by

exploiting the geometry and algebra of high-dimensional random vector spaces. Intuitively, it enables *metaphorical* and *by analogy* reasoning like it is done in the animal brain [47]. The central observation is that large circuits are fundamental to the brain's computation. HDC incorporates this notion by computing with distributed representations [26] in high dimensions, called *hypervectors*. The dimensionality of these hypervectors is commonly in the order of thousands or tens of thousands.

Such hyperspaces (short for hyper-dimensional spaces) allow replicating certain rich brain properties that are otherwise difficult to reproduce on computers. See for instance the "what's the Dollar of Mexico?" example by Kanerva [47]. Hypervectors typically represent information holographically, meaning that each of the thousands of dimensions contains the same amount of information, ensuring inherent robustness [46, 113]. Many variations of the hyperspace have been proposed, which we refer to as *HDC models*, ranging from binary till complex numbers [94], although they are all largely similar at a conceptual level.

In addition to representation, a crucial part of a computer system is information manipulation, or arithmetic. The arithmetic in HDC is based on well-defined operations between hypervectors, such as addition (bundling), multiplication (binding) and permutation. The implementation of these operations differs between HDC models but they achieve the same abstract result, that is: superposition, association, and ordering of information, respectively. Another important function is information comparison, which in HDC usually consists of measuring the similarity between hypervectors using the dot product or the cosine similarity. Both manipulation and comparison of information are based on dimension-independent operations, providing an opportunity for massive parallelism [66, 90].

Computational efficiency is one of the core motivations aimed at since the conception of HDC and it is envisioned for and expected to reach full potential in specialized hardware [46]. Recent changes in computing demands have motivated the emergence of new hardware models, called *neuromorphic* computing paradigms (also known as: *brain-like*, *unconventional* and *natural* computing) [43]. These emerging platforms are conceived to operate with massive parallelism, at the nanoscale, with ultra-low voltage and unreliable components, resulting in a stochastic execution environment [51, 55, 105]. Because of the characteristics of HDC, such as robustness to noise and the inherent randomness, it has been proposed as an abstraction layer that enables the design of algorithms for these platforms [55].

In addition to the aforementioned parallelizability of HDC, optimizations such as in-memory processing promise to further increase the computational efficiency of HDC [40]. Schmuck et al. [95] apply a series of hardware techniques to optimize HDC, such as on-the-fly *rematerialization* of hypervectors and special memory architectures, to improve chip area and throughput at the same time. Particularly important to substantiate the claims we make in this paper about efficiency (see "Method" section), they demonstrate an FPGA implementation that uses deep adder trees to perform inference in a single clock-cycle.

The aforementioned efficiency and robustness of HDC together with its ability to solve cognitive tasks have motivated many machine learning (ML) and AI applications of HDC. We will discuss important examples of these applications in "Applications in

symbolic AI" section. The algorithms used in these applications can often be separated into three stages: encoding, training, and inference. Throughout this paper, we present each of these stages in detail and through examples. Of all these stages, encoding is presumably the most important. The encoding stage is application specific and serves to map objects in the input space $\mathcal{X}$ to hypervectors in $\mathcal{H}$. In a learning setting, the training phase aggregates hypervectors into prototypes to learn a model. Inferences can then be made using the generated class representations by comparing their similarity with an encoded test sample.

Besides cognitive tasks, the properties of HDC have also been a motivation to apply it to more general computation problems, which we will refer to as *stochastic computation*. Research in this domain seeks to create solutions to classical problems, such as hashing [32] or graph isomorphism [23] by exploiting the characteristics of HDC. These applications encompass both deterministic computation in the presence of noise and probabilistic algorithms that search for approximate solutions. Important applications are discussed in "Applications in stochastic computation" section.

In this paper we begin by presenting a detailed description of the components of HDC, accompanied by a survey of the literature relevant to each of these parts. We also present the concepts in a didactic way through examples, with the aim of serving as an introductory tutorial to the research field. Then, we present the range of possible applications of HDC, mainly highlighting its use both in machine learning applications and as a more general stochastic computation framework. To this end, we also present in detail two application examples: first, in "GraphHD" section, we describe GraphHD, a graph classification method; then, in "Hyperdimensional hashing" section, we introduce HD hashing, a dynamic hashing algorithm. Finally, we conclude the paper by presenting and discussing empirical results of these two methods to show how, in fact, approaches based on HDC have the potential to bring efficient and robust alternative solutions to relevant problems.

Problem definition

There is a trend among computing hardware towards highly parallel and stochastic platforms to satisfy new demands in computing [51, 105]. Compared to current hardware, these emerging platforms such as neuromorphic processors and in-memory computing architectures, consume only a fraction of the energy. Given the diversity of these emerging hardware platforms, the need for an abstraction layer that allows describing algorithms for such hardware emerges. This abstraction layer needs to be expressive such that it can be used to solve a wide range of complex problems. Moreover, to meet the requirements of these emerging demands, it needs to execute efficiently while being robust to the inherent noise of the stochastic computing hardware. HDC has been proposed as this abstraction layer, serving as the interface between application design and these emerging hardware platforms [55].

Among the most important of the growing demands that the emerging hardware is targeting are those related to machine learning. In the ever more important field, achieving the state-of-the-art often involves training ever larger models to improve the accuracy. It is important to note that with this expanding relevance, the need for learning capability has spread to embedded devices as well. This context presents obstacles to

Table 1 Example of three data records

Record	Fruit (F)	Weight (W)	Season (S)
r_1	Apple	80	Fall
r_2	Lemon	60	Winter
r_3	Mango	180	Summer

Table 2 Overview of HDC models

Model	Hyperspace	Bundling	Binding	Unbinding
BSC	Binary	Majority	Exclusive or	Exclusive or
MAP	Bipolar	Addition	Multiplication	Multiplication
HRR	Real unitary	Addition	Circ. convolution	Circ. correlation
FHRR	Complex unitary	Addition	Multiplication	Conj. multiplication

many conventional solutions, due to memory, timing, and power constraints [7, 10]. Moreover, their black-box approach to intelligence makes it particularly challenging to understand why and how the complex behavior formed and in which settings it will fail [5].

Despite the breath of interesting results presented in the HDC literature on the aforementioned topics, there are still many open questions. We refer the reader to the works by Hassan et al. [31] and Kleyko et al. [56] for a discussion on open problems and challenges related to HDC.

Existing solutions

In this section we present a detailed and didactic description of the elements and concepts that constitute HDC/VSA. The definitions will be exemplified using the fictitious data records in Table 1. These concepts form the basis for every application of HDC/VSA. This section is concluded with examples of applications in the domains of cognitive and stochastic computation using HDC/VSA to provide real examples of the usage of the concepts described throughout this section and to illustrate the distinction between the machine learning, and the more general computation side of HDC.

To facilitate understanding, we will present all the concepts for a specific HDC model called *Binary Spatter Codes* (BSC) [48], where the hyperspace consists of binary vectors. In Table 2 we list some other existing models in the literature, along with their differences; these include: Multiply-Add-Permute (MAP) [21], Holographic Reduced Representations (HRR) [84] and Fourier HRR (FHRR) [85]. For a more detailed description of these and many other existing models, the reader is referred to the surveys by Kleyko et al. [57] and Schlegel et al. [94]. Note that the concepts presented in this Section are applicable to all HDC/VSA models simply by using the appropriate model representations and operations.

The mapping of data to the high-dimensional space is the first step in HDC and this process corresponds to *encoding*. The process is governed by a function $\phi : \mathcal{X} \to \mathcal{H}$, that maps input arguments (e.g., graphs, text or images) in an input space $\mathcal{X}$ to the d-dimensional space $\mathcal{H}$. In the context of learning, encoding is the HDC counterpart to the

feature extraction process in classical learning methods. Thus, the main intuitive principle that governs the encoding is that inputs that are similar in the original space should be mapped to similar hypervectors.

Similarity metrics

Similarity between hypervectors is generally measured using the cosine similarity δ_c, or the normalized dot product δ_d. In a binary setting the inverse normalized Hamming distance δ_h is also used. The definitions provided in Eq. 1 show that these similarity metrics are very related. The inverse normalized Hamming distance δ_h for binary hypervectors is in fact equivalent to a shifted and scaled normalized dot product for bipolar hypervectors. There is a subtle distinction between the cosine similarity and the normalized dot product: the normalized dot product can be used to retrieve frequency information from a hypervector, for example, "*how many times is* $\mathbf{x}$ *in* $\mathbf{y}$?" The cosine similarity, in contrast, normalizes over the magnitude of the vectors and therefore loses the frequency information. However, it can still be used to answer containment questions, e.g., "*is* $\mathbf{x}$ *in* $\mathbf{y}$?" The most appropriate similarity metric is thus application specific and throughout this paper we will refer to a generic similarity metric δ or specify explicitly which similarity is used when necessary.

$$\delta_c(\mathbf{a},\mathbf{b}) = \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}, \quad \delta_d(\mathbf{a},\mathbf{b}) = \frac{\mathbf{a}\cdot\mathbf{b}}{d}, \quad \delta_h(\mathbf{a},\mathbf{b}) = \frac{1}{d}\sum_{i=1}^{d}\mathbf{1}(\mathrm{a}_i = \mathrm{b}_i) \tag{1}$$

Basis-hypervectors

Basis-hypervectors, also called *seed* or *atomic* hypervectors, are sets of hypervectors that represent, i.e., encode, the smallest units of meaningful information in the hyperspace. Their generation is typically the first stage in the encoding process. In Table 1, the atomic information are the possible fruit types, fruit weights, and seasons of the year. Since the nature of these information sets is different, so are the basis-hypervector sets used to encode them. The basis hypervectors remain fixed throughout computation and each data sample is encoded by combining and manipulating them using the addition (bundling), multiplication (binding) and permutation operations. The different sets of basis-hypervectors are illustrated in Fig. 1 and described below.

Random-hypervectors In this set, each hypervector is sampled uniformly at random from the hyperspace. Because of the high dimensionality of $\mathcal{H}$, each pair of symbols $\mathbf{r}_i$ and $\mathbf{r}_j$ in $R = \{\mathbf{r}_1, \ldots, \mathbf{r}_m\}$ is quasi-orthogonal with high probability, a phenomenon known as *concentration of measure* [45, 64]:

$$\delta(\mathbf{r}_i,\mathbf{r}_j) \approx 0 \; \forall i \neq j, \quad \mathbf{r}_i \sim \mathcal{U}(\mathcal{H}) \tag{2}$$

For this reason, it is used to map (one-to-one) categorical/symbolic data, such as the fruit types in Table 1, to the hyperspace. We denote by $\phi_R(x)$ the hypervector in R used to encode the atomic information x using random-hypervectors.

Level-hypervectors The set $L = \{\mathbf{l}_1, \ldots, \mathbf{l}_m\}$ is used to encode linearly correlated information, usually representing a subinterval of the real number line $\mathbb{R}$. Examples include: distance, time, energy and, in our Table 1 example, the weight of a fruit. Unlike

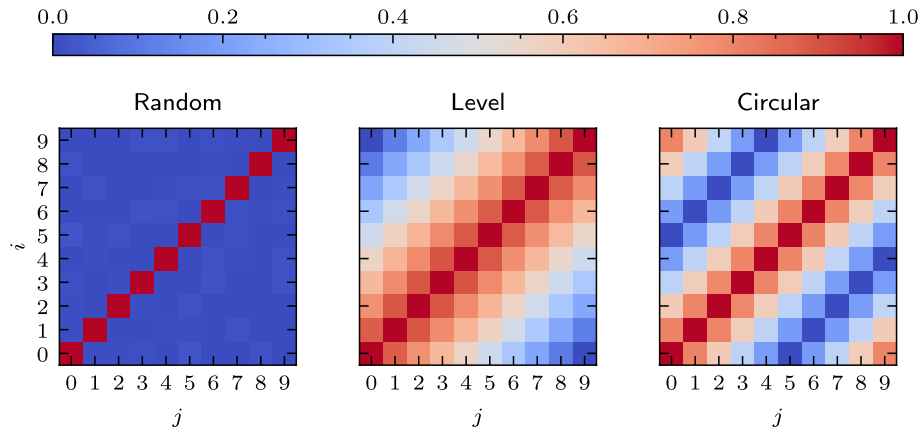

Fig. 1 Pairwise similarity of each *i*-th and *j*-th hypervector within a basis-hypervector set of size 10. Comparing between random, level and circular basis-hypervectors

random-hypervectors, these vectors are generated from an interpolation of vectors $\mathbf{l}_1$ and $\mathbf{l}_m$ which are the only two that are sampled uniformly at random from the hyperspace. Each hypervector in L is associated with a value, such that the distance of the values in $\mathbb{R}$ is proportional to the distance of their respective hypervectors. Different methods for creating a set with these characteristics were proposed independently [86, 87, 89, 101, 112]. More recently, an improved version in its representational capacity was introduced by Nunes et al. [77]. In the case of the FHRR model, exponentiation of complex components is commonly used to create level-hypervectors, in a process known as *fractional power encoding* [83]. We represent by $\phi_L(x)$ the encoding of x using level-hypervectors.

Circular-hypervectors Another important type of information, whose correlation profile is not captured by the sets above, is *circular data* [82]. This kind of data is derived from the measurement of directions, usually expressed as an angle in $\Theta = [0, 2\pi]$. It is also common to handle time measurements, like the seasons of the year in our example, as circular data. We propose the set, denoted by $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_m\}$ in "Circular hypervectors" section, which is built on the construction of level-hypervectors, but is divided into forward and backwards transformations, making the distance between the hypervectors proportional to the distance of the angles in Θ that they represent. We denote by $\phi_C(x)$ the hypervector in C used to encode the atomic information x using circular-hypervectors.

Operations

Arithmetic in HDC is based on three operations: *binding*, *bundling*, and *permuting*, outlined below. These functions are dimension-independent and always yield hypervectors in the same space as the operands, enabling the composition of operations. The implementations of these operations for the BSC model are illustrated in Fig. 2.

Binding The binding function $\otimes : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$, produces a vector dissimilar to both its operands. This operation is commonly used to "associate" information, for instance, to assign values to variables. The hypervector $\phi_R(F) \otimes \phi_R(apple)$ can be used to represent the variable-value pair *Fruit←apple* of record r_1 in Table 1,

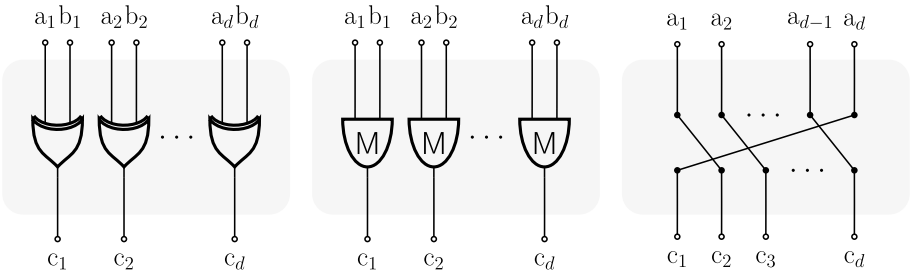

Fig. 2 *Binding*, *bundling*, and *permutation* operations illustrated on binary hypervectors **a** and **b** according to the Binary Spatter Codes HDC model. The subscript denotes the dimension index of the hypervector. The logical gates in the bundling operation are *majority gates*

where the variable name, i.e., its symbol, is encoded using random-hypervectors. In many applications it is important to define an inverse binding operation to retrieve the value of a variable from a variable-value hypervector. This operation is commonly called *unbinding* or *release*, denoted by $\oslash$, and behaves in the following way: $\mathbf{a} = (\mathbf{a} \otimes \mathbf{b}) \oslash \mathbf{b}$. Observe that in the MAP and BSC models $\otimes \equiv \oslash$, as their binding operations are self-invertible in the respective hyperspaces. Binding is also commutative and distributive over the bundling operation described below.

Bundling The bundling operation, also known as *superposition*, is used to aggregate information into a single hypervector. The function $\oplus : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$ produces a vector that is *maximally similar* to the operands. In this way, record r_1 from our example can be encoded to a single hypervector as the aggregate of variable assignments:

$$\phi(r_1) = [\phi_R(F) \otimes \phi_R(apple)] \oplus [\phi_R(W) \otimes \phi_L(60)] \oplus [\phi_R(S) \otimes \phi_C(fall)]$$

Notice that, as explained in "Basis-hypervectors" section, when encoding record r_1 above we used the appropriate basis-hypervectors to encode each of the values according to the correlation profile of their information in the input space. Also note that each variable is encoded as a different vector drawn from a random-hypervector set. There is also an inverse operation to bundling $\ominus : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$ which can be used to remove an item from a hypervector by bundling with the additive inverse. Because hypervectors are typically normalized to ensure that the resulting vector is in the same domain as the basis-hypervectors, bundling and inverse bundling become approximate operations. As an example, the BSC model uses majority voting. In that case the addition of noise in the hypervector for each bundling operation needs to be considered. This can be mitigated by using full-precision hypervectors such that the operations become exact.

Permuting The *permutation* operator is used to assign an order to hypervectors. The function $\Pi : \mathcal{H} \to \mathcal{H}$ outputs a hypervector that is dissimilar to its input. The exact input can be retrieved with the inverse operation. Cyclic shift is the most commonly used permutation and with $\Pi^i(A)$ we denote a cyclic shift of the elements of A by i coordinates. Suppose that in the example of Table 1, we want to represent not only the fruits and their characteristics, but also someone's preference order, say mango, apple then lemon. This sequence of records can be encoded as $\phi(r_3) \oplus \Pi^1\big(\phi(r_1)\big) \oplus \Pi^2\big(\phi(r_2)\big)$. In real applications, permutation has been used to represent time series and n-grams [4, 42, 73, 89].

A significant effort in HDC has been devoted to developing new and better encoding strategies [23, 31, 44, 74, 83, 102]. Encoding functions are generally application specific and are critical in successfully applying HDC to a problem. In the following section we will describe some common encoding patterns.

Encodings

Encoding patterns are used to combine multiple atomic pieces of information to encode something more complex. Examples include encoding text from characters [91, 99], graphs from vertices and edges [23, 78], time series from samples [25, 89] and images from pixel values [62, 68], among others. Below we present the most common encoding strategies.

Multisets The simplest method to combine information in hyperspace is to create a multiset. A multiset is created by bundling values together, forming a hypervector that is similar to the inputs. The equation below shows the creation of a multiset from the input hypervectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\}$:

$$\phi_{multiset}(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m) = \mathbf{v}_1 \oplus \mathbf{v}_2 \oplus \cdots \oplus \mathbf{v}_m$$

A multiset can be queried to determine whether it contains a given value by calculating the similarity of the value with the set. If the similarity is close to one, the element is in the set with high probability. Moreover, the number of occurrences of a value can be determined using the dot product. This is why the obtained hypervector can be seen as a *multiset*, rather than a *set*. For more details on the information retrieval and capacity aspects of multisets refer to Thomas et al. [107].

Hash tables To combine values with different semantic meaning it is common to introduce a key $\mathbf{k}_i$ for each value $\mathbf{v}_i$. The combined representation goes by the names of *hash table*, *record-based* encoding, or *role-filler* bindings. This encoding binds each key with its respective value hypervectors and then bundles them to create the combined representation [46, 83]:

$$\phi_{hash-table}(\mathbf{k}_1, \mathbf{v}_1, \mathbf{k}_2, \mathbf{v}_2, \ldots, \mathbf{k}_m, \mathbf{v}_m) = \bigoplus_{i=1}^{m} \mathbf{k}_i \otimes \mathbf{v}_i$$

The key hypervector can for example correspond to the position in an image or the identifier of a variable. This is the method used to illustrate the encoding of the fruits in Table 1. The value associated with a key can be efficiently queried, like a hash table in programming. This is done by unbinding (see "Operations" section) the hash table with the key and comparing the similarity of the resulting hypervector with the set of basis-hypervectors for the value. The most similar basis-hypervector is the one corresponding to the value that is the most likely to be associated with the queried key. This similarity search is equivalent to a memory lookup using associative memory, which is discussed in "Associative memory" section.

Sequences A sequence is encoded as a multiset of permuted values, where the number of permutations depends on the position where the value appears in the sequence [46, 83]:

$$\phi_{sequence}(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m) = \bigoplus_{i=1}^{m} \Pi^{m-i}(\mathbf{v}_i)$$

This *bundling-based* sequence encoding is used especially when it is important to compare the similarity between sequences. Alternatively, one can use *binding-based* sequences, where the permuted values are bound together instead of bundled. In this strategy, which has already proved useful in some applications [44, 91], even sequences that differ in only one element are completely dissimilar in hyperspace.

The bundling-based sequences can also be used to implement other common data structures such as stacks and queues [55, 83, 118]. To implement both, methods for pushing and popping values from the beginning and end of the sequence need to be defined:

$$\begin{aligned}
\text{item}(\mathbf{s}, i) &= \text{cleanup}(\Pi^{-m+i}(\mathbf{s})) \\
\text{pushend}(\mathbf{s}, \mathbf{v}) &= \Pi(\mathbf{s}) \oplus \mathbf{v} \\
\text{pushstart}(\mathbf{s}, \mathbf{v}) &= \Pi^{m}(\mathbf{v}) \oplus S \\
\text{popend}(\mathbf{s}) &= \Pi^{-1}(\mathbf{v} \ominus \text{item}(\mathbf{s}, m)) \\
\text{popstart}(\mathbf{s}) &= \mathbf{s} \ominus \Pi^{m-1}(\text{item}(\mathbf{s}, 1))
\end{aligned}$$

where $\mathbf{s}$ is a sequence of length m. The cleanup function performs an associative memory lookup of the given value, returning the original hypervector and thus removing noise introduced by the bundling of multiple values. Recall that the $\ominus$ symbol represents the inverse bundling operation.

N-grams Statistics of n-grams from textual data are often used in natural language processing [92]. In HDC, an n-gram is encoded by binding n consecutive tokens. A token $\mathbf{v}_i$ can be a word or a character, typically encoded using random-hypervectors. The entire text is then encoded as a multiset of all the n-grams [44, 91]:

$$\phi_{n-gram}(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m) = \bigoplus_{i=1}^{m-n+1} \bigotimes_{j=0}^{n-1} \Pi^{n-j-1}(\mathbf{v}_{i+j})$$

Graphs A graph $G = (V, E)$ is usually represented in the hyperspace as a multiset of its edges E. Each edge in turn, is encoded by binding the hypervectors of its endpoint vertices [23, 78]:

$$\phi_{graph}(G = (V, E)) = \bigoplus_{(\mathbf{v}_i, \mathbf{v}_j) \in E} \mathbf{v}_i \otimes \mathbf{v}_j$$

where the hypervector $\mathbf{v}_i$ is the encoded vertex at index i. In the case of directed graphs, a permutation operator is applied to the destination vertex to differentiate edges $(\mathbf{v}_i, \mathbf{v}_j)$ and $(\mathbf{v}_j, \mathbf{v}_i)$. A graph can be queried to find the neighbors of a node by binding the graph with the node hypervector (followed by an inverse permutation, for directed graphs). All the nodes that are similar to the resulting hypervector are connected to the given node with high probability.

Random projections Alternatively, an information vector $\boldsymbol{x} \in \mathbb{R}^m$ can be mapped to the hyperspace directly using a random projection $\phi_{project}(\boldsymbol{x}) = \text{sign}(\Phi \boldsymbol{x})$, where $\Phi \in \mathbb{R}^{d \times m}$ is a matrix whose rows are uniformly sampled at random from the surface of

an m-dimensional unit sphere. This encoding ensures that similarities in the input space are preserved in the hyperspace [107]. Variations of the projection encoding have been proposed with different nonlinearities and weights sampled from other distributions [12, 35, 120].

Associative memory

Since hypervectors are typically created by a random process and do not inherently represent anything, they are stored in a special type of memory during their lifetime in an application. This memory is called associative memory, and differs from normal address accessible memory by returning the value at the address that is *most similar* to the requested address. The associative memory is a core component of most HDC/VSA applications and is key in enabling the encoding methods described before. Also, classification and regression tasks in HDC use an associative memory to make predictions as will be detailed in "Classification" section.

Let the requested address be the query vector $\mathbf{q}$, and the addresses and values of the associative memory be the keys $\mathbf{K}$ and values $\mathbf{V}$ hypervectors, respectively. Formally, the returned hypervector from a memory lookup is given by:

$$\text{lookup}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \mathrm{V}_i, \quad \text{where } i = \arg\min_j |\delta(\mathrm{K}_j, \mathbf{q})| \tag{3}$$

When the keys and values are the same, the associative memory can be used as a *cleanup memory*. That is, given some noisy, approximate version of a hypervector, its exact/original representation can be retrieved with high probability using the associative memory with equal keys and values.

Because of the parallel nature of HDC operations, including the similarity metric computation, an associative memory lookup can be efficiently implemented in hardware using deep adder trees [95].

Classification

In a learning setting, the encoding methods described above are used to map the data samples to the hyperspace. These samples then need to be combined to form a model. As an illustration, consider a large dataset with fruit records like those in Table 1, where each record has a class label, e.g., the fruit taste: sweet, sour, salty, bitter, etc. If we denote the label of each record r by $\ell(r) \in \{1, \ldots, k\}$, we can use the hypervectors of each record to create a prototype hypervector $\mathbf{m}_i$ for each class $i \in \{1, \ldots, k\}$ as follows:

$$\mathbf{m}_i = \bigoplus_{j:\ell(r_j)=i} \phi(r_j)$$

Each of these hypervectors is referred to as a *class-vector* and is the vector with the smallest average distance to the hypervectors obtained by encoding the training samples of class i.

By building a class-vector for each class, we can combine them as a classification model $M = \{\mathbf{m}_i, \ldots, \mathbf{m}_k\}$ and use it to predict the class of new records (see Fig. 3). Given an unlabeled input $\hat{r} \in \mathcal{X}$ and M, we simply compare $\phi(\hat{r})$, called the *query-vector*, with each

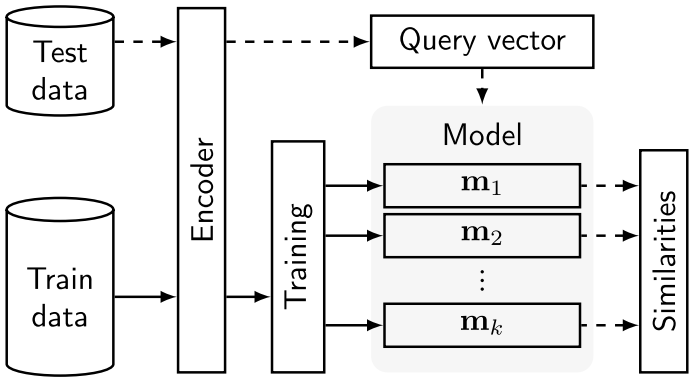

Fig. 3 Overview of the hyperdimensional computing classification framework. Solid lines indicate training steps, dashed lines indicate inference steps

class-vector and infer that the label of $\hat{r}$ is the one that corresponds to the most similar class-vector:

$$\ell^\star(\hat{r}) = \underset{i\in\{1,\dots,k\}}{\arg\max}\ \delta\big(\phi(\hat{r}), \mathbf{m}_i\big)$$

where $\ell^\star(\hat{r}) \in \{1,\dots,k\}$ is the predicted class for $\hat{r}$. It is important to emphasize that the function ϕ used in the inference stage is exactly the same as the one used for the creation of the prototypes. For an extended review on classification using HDC, we refer the reader to the works by Ge and Parhi [24], and Vergés et al. [109].

Regression

In a regression setting, the model M consists of a single hypervector $\mathbf{m}$, which memorizes training samples with their associated label. This is different from the classification setting where the class of a sample is implicitly stored as the index of the class-vector. The label of each sample in a regression setting is a real number $\ell(x) \in \mathbb{R}$. To encode a label, an invertible encoding function ϕ_ℓ, which maps real numbers to hypervectors, needs to be introduced. The invertibility property is needed to allow labels to be determined during inference. The function outputs level-hypervectors, a finite subset $L = \{\mathbf{l}_1,\dots,\mathbf{l}_k\}$ of all hypervectors in $\mathcal{H}$ whose generation is discussed in "Basis-hypervectors" section. The hypervectors in $\mathbf{l}_i$ are linearly correlated such that the closer the real numbers they represent, the more similar the hypervectors are. The regression model is then obtained as follows:

$$\mathbf{m} = \bigoplus_i \phi(x_i) \otimes \phi_\ell(\ell(x_i))$$

A prediction can be made given a trained model M and an unlabeled input $\hat{x} \in \mathcal{X}$. First the approximate label hypervector is obtained by unbinding the model with the encoded sample $\mathbf{m} \oslash \phi(\hat{x}) \approx \phi_\ell(\ell(\hat{x}))$, where $\oslash$ represents the unbinding operation. The remaining terms add noise, making the result approximately equal [46, 107]. The precise label hypervector is then the most similar label hypervector $\mathbf{l}_l$, where:

$$l = \arg\min_{i \in \{1,\ldots,k\}} \delta(\mathbf{m} \oslash \phi(\hat{x}), \mathbf{l}_i)$$

Finally, the label is obtained by decoding the label hypervector using the inverse of the label encoding function:

$$\ell^\star(\hat{x}) = \phi_\ell^{-1}(\mathbf{l}_l)$$

Applications in symbolic AI

We conclude the existing solutions with notable HDC applications in Symbolic AI or cognition tasks, and stochastic computation tasks (see "Applications in stochastic computation" section). The aim is to provide real world examples of the aforementioned concepts of HDC. Secondly, these sections highlight the dual use of HDC which we make explicit by separating the applications in two sections.

Language recognition The first application uses a dataset of sentences from 21 European languages. Rahimi et al. [91] present a method to classify the language of a given sentence. This method was among the first to show that HDC is capable of achieving comparable accuracy to established ML algorithms while being significantly more efficient and robust to memory errors. Each letter is first mapped to a random-hypervector, n-gram statistics are then gathered from the sentence in hyperspace. The similarity of the resulting hypervector is then compared against the class-hypervectors, created during training by bundling all the n-gram statistics hypervectors for each language, for each class.

EMG gesture recognition Rahimi et al. [89] propose a method for classifying hand gestures using Electromyography (EMG) signals in the setting of a smart prosthetic. The dataset provides 4 channels of EMG signal which is first spatially encoded by mapping each signal to one of 21 level-hypervectors. The channels are combined using a hash table encoding, the resulting spatial hypervector is then used to encode the temporal information using n-gram statistics over a fixed-size window.

Voice recognition Imani et al. [41] present an HDC classifier for voice signals to guess the spoken letter from an audio stream. In VoiceHD the amplitudes of the audio sample are mapped to level-hypervectors which are then combined using hash table encoding. The encoded samples are classified using the same classification framework as the one described in "Classification" section. Experiments with the addition of a neural network trained on the outputs of the HDC model shows improved accuracy. They also show that a hybrid model in combination with a neural network improves the accuracy.

Regression Hernández-Cano et al. [35] propose a method for regression. They use random projection to map the input vectors to the hyperspace and train with gradient descent. A prediction is made by calculating the dot product between the sample and model hypervectors. They also describe a multi-model setting where each model is assigned to a cluster of training samples, a prediction is then made based on the weighted sum of the models weighted by the similarity of the sample with the cluster centers.

Applications in stochastic computation

Finally, we present applications of HDC in stochastic computation settings which we categorize as the second general area of applicability of HDC.

Stack machine Yerxa et al. [118] describe a stack machine based on HDC operations by popping and pushing hypervectors to a hyperdimensional stack, defined in "Encodings" section. They also define a data structure for finite state machines and argue that these can be used together with established machine learning methods because the operations are fully differentiable.

Bloom filter Yerxa et al. [58] present a generalization of the bloom filter that enables the adjustment of its capacity. To achieve this a perfect true positive rate cannot be guaranteed but probabilistic bounds are provided. The algorithm is essentially a binarized multiset encoding of sparse Multiply-Add-Permute hypervectors. A change in capacity is obtained by modifying the threshold of the binarization process.

Service discovery By describing each service in a distributed system using HDC, Simpkin et al. [100] introduce a method for decentralized service discovery. They motivate their application of HDC by its ability to encode rich information compactly while being robust to noise. The representation of a service is created using the hash table encoding described in "Encodings" section.

Link prediction and Document Deduplication [79] introduce an estimator for set similarity metrics such as the Jaccard and Adamic-Adar indices. Their method uses the HDC multiset encoding to estimate a broader family of metrics compared to previous estimators.

Graph isomorphism Gayler and Levy [23] present a mechanism to find graph isomorphisms based on HDC. The method creates hypervectors by superposing the vertices and edges of the graph and serves as a proof-of-concept of the applicability of distributed representations in the problem.

Proposed solutions

In this section we present two very different examples of applications that can be addressed using HDC. The main objective is to illustrate that the framework is applicable both in machine learning and in other computational tasks when accuracy, robustness and efficiency are simultaneous requirements. Experimental results for both applications are presented and discussed in detail in "Elaboration" section.

The first example is GraphHD ("GraphHD" section), an algorithm for the relevant problem of classifying graphs. The proposed method creates distributed representations of graphs from the superposition of their edges. The scheme presented in "Classification" section is then used in its classification.

Before presenting the second example, in "Circular hypervectors" section we introduce *circular-hypervectors*—a new basis-hypervector set capable of encoding data with circular correlation to hyperspace (see "Basis-hypervectors" section). Finally, in "Hyperdimensional hashing" section we show how these hypervectors can be used to create a consistent hashing algorithm, used in the critical problem of load balancing that arises in cloud computing.

GraphHD

One class of learning tasks that is missing from the current body of work on HDC is graph classification. Graphs are among the most important forms of information representation, yet, to this day, HDC algorithms have not been applied to the graph learning problem in a general sense. We present GraphHD, a baseline approach for graph classification with HDC.

Motivation

Machine Learning has played an increasingly central role in academic research and industrial applications. This popularity is due in large part to the good empirical results obtained on problems in which data is captured in the Euclidean space, such as vectors of feature values, time series data or images. However, in countless real-world scenarios, in both natural and social sciences, we are often interested in representing relationships between entities. Examples range from chemical molecules [110] and bioinformatics [6], to computer vision [75] and analysis of social networks [117]. The information about such entities and the relationships between them is inherently non-Euclidean. Graphs, instead, provide a much more natural abstraction. For this reason, the challenge of developing methodologies capable of utilizing the full potential of machine learning algorithms to deal with graphs has received a lot of attention from the scientific community in recent years.

One of the first successful strategies for graph learning problems is to calculate a measure of similarity between graphs. These methods are called graph kernels [61]. The similarity measurement functions are used in conjunction with kernel machines (e.g. support vector machines) to perform cognitive tasks such as classification. A myriad of graph kernel methods have been proposed, especially in the last 15 years, which will be covered in "Background" section. While it is true that kernel methods are highly competitive graph learning approaches, especially on small graphs, considerable recent effort has focused on alternative methods [72] with better scaling and performance characteristics. In particular, kernel methods scale quadratically with respect to the size of the dataset and do not allow for online learning [52], limiting their applicability in real-time scenarios [119]. Our work introduces a new alternative approach to graph kernels.

Another popular alternative, motivated by the notorious accomplishments of deep neural networks, are graph neural networks (GNNs). GNNs are models that extend regular neural network operations, such as pooling and convolution, to handle graphs. Despite the functional accuracy achieved by GNNs, the high computational and energy cost of deep learning approaches make them difficult, or prohibitive, to be applied in real-world situations, such as those encountered in IoT and embedded applications [54, 63]. The demand for alternatives is clear given the growing number of graph learning applications in resource-constrained scenarios. Examples range from IoT malware detection [1] to air pollution monitoring sensor networks [17].

The previously introduced characteristics of Hyperdimensional Computing have already shown their merit on several problems (see "Existing solutions" section). However, despite the previously stated importance of graph learning applications, to the best of our knowledge, HDC algorithms have never been applied to such tasks. It is based

on this motivation that we propose GraphHD, the first of its kind baseline approach for graph learning with HDC. GraphHD focuses on providing an efficient, robust and scalable alternative to current state-of-the-art graph learning algorithms.

Background

With the growing number of applications for machine learning with graphs, several approaches have been proposed in recent years. Popular ones include those based on *graph kernels* and *graph neural networks*. Examples of graph kernel approaches are those based on spectral properties [59], random walks [49] and matching of node embeddings [76]. Some very prominent graph kernels are based on the well-known heuristic for graph isomorphism, i.e., the Weisfeiler-Leman (WL) algorithm [60, 98]. For a detailed and recent review of graph kernels, see Kriege et al. [61].

More recently, numerous attempts to adapt neural networks to deal with graphs have come to be known as graph neural networks (GNNs) [114]. The initial concept is due to Gori et al. [29], further elaborated by Scarselli et al. [93]. Despite the recent and extensive exploration of GNNs, classical graph kernels are still very competitive in terms of accuracy [72] and especially in efficiency (as indicated by our results in "Elaboration" section). Xu et al. [115] show that GNNs are *at most* as powerful as the WL test in distinguishing graph structures.

While these existing approaches have a well established track record in the field of graph similarity analysis, our HDC approach is a novel attempt at solving graph learning tasks.

Notations and problem formulation Let $G = (V, E)$ denote a graph with vertex set V and edge set E with $n = |V|$, $m = |E|$. The class of a specific graph G is denoted by $\ell(G)$. The graph classification problem is defined as follows: given a set of graphs $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$ and a training subset $\mathcal{G}_L \subset \mathcal{G}$ for which the labels are known, create a model capable of predicting the unknown labels for the graphs in $\mathcal{G} \setminus \mathcal{G}_L$.

An important thing to mention is that, since GraphHD was thought of as a baseline method for graph learning with HDC, we assume that graphs do not have any other information such as labels or attributes. Although some datasets contain this type of information, we decided to keep GraphHD as uniform and generic as possible in the present work. The use of this information in ad-hoc applications can be advantageous and shall be investigated in future work.

Method

Overview As described in "Existing solutions" section, the first and most important question to be addressed when applying HDC to a domain is: *how to encode the input data?* We want to define a function $\phi_G : \mathcal{G} \to \mathcal{H}$, capable of mapping graphs in the input set to d-dimensional hypervectors. Illustrated in Fig. 4, the overall strategy of GraphHD's encoding is to map each element that composes the graph, i.e. its vertices and edges, individually to a hypervector and then combine the information using the bundling operation.

Encoding GraphHD starts by encoding the vertices of the graph and those hypervectors are then used as input to the edge encoder. We will first describe the process $\phi_v : V(G) \to \mathcal{H}$ used to encode each element in the set of vertices of a graph G, denoted

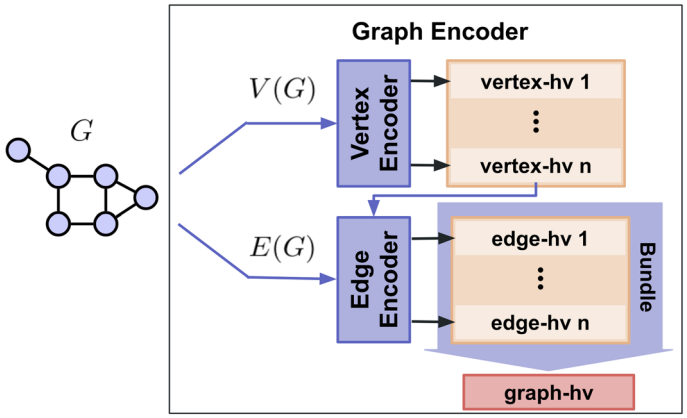

Fig. 4 GraphHD encoding

by $V(G)$. As presented in our example from "Existing solutions" section, in the existing encoding examples, used for non-graph data, the process usually starts by assigning basis-hypervectors to each possible information. For example, one for each letter to encode text.

Based on the existing encoding strategies, one could think of starting to encode graphs by assigning independent random hypervectors to each vertex. However, note that in these other problems, there is a relationship between the symbols that is consistent across different samples of the dataset. For example, the symbol "A" in a text represents equivalent information in another text, which makes it reasonable to encode both using the same hypervector. However, since we only look at the structure of the graphs, there is no such trivial correspondence between vertices of different graphs.

To address this issue, the vertex encoding process needs to start by extracting an identifier for the vertices based only on the topology of the graph. For this purpose, we propose the use of the *PageRank* centrality metric [8]. Initially developed by Google to rank web pages in the web graph, the PageRank algorithm receives a graph as input and returns, for each vertex $v_i \in V$, a value $c(v_i) \in [0, 1]$ that measures its "importance" in the graph. The metric is well established and has been widely applied to different problems beyond the web [28]. As rests evident from its initial application, PageRank can be implemented in a very efficient and scalable manner, which matches the purpose of GraphHD.

From this ranking induced by the PageRank centrality of the vertices, it is possible to establish a meaningful connection between vertices in different graphs. Therefore, GraphHD uses the centrality rank of the vertex as its identifier (or symbol). Accordingly, vertices of different graphs, but with the same centrality rank, are encoded to the same random hypervector from the basis set.

After creating the hypervectors for each vertex, GraphHD makes use of these representations to also encode each edge $(v_i, v_j) \in E(G)$. The edge encoding function ϕ_e is defined as follows:

$$\phi_e((v_i, v_j)) = \phi_v(v_i) \otimes \phi_v(v_j)$$

Recall that $\otimes$ represents the binding operation in HDC, which is the standard operation to represent an association between a pair of hypervectors, similar to the role of an edge in a graph. The result of the binding operation is a third vector, statistically quasi-orthogonal to the operand vectors, which we name *edge-hypervectors*.

Training Based on the encoding functions presented, GraphHD training is described in Algorithm 1. For each class we generate a set H_ℓ of hypervectors. Each hypervector included in H_ℓ (line 12 in Algorithm 1) is what we call a *graph-hypervector*. For each graph G, the corresponding graph-hypervector is created with bundle(H_G), which bundles all the edge-hypervectors contained in the set H_G. Note that vertex encoding is used as an intermediate edge encoding step as defined above.

Algorithm 1 GraphHD training procedure

1 GraphHD_Training $(\mathcal{G}_L)$;
Input : A training set of graphs $\mathcal{G}_L$ with their respective labels and a set of random vertex-hypervectors H_v.
Output: A trained HDC model M consisting of the class vectors $\{\mathbf{m}_1, \ldots, \mathbf{m}_k\}$
2 **for** *each class label* $i \in \{1, \ldots, k\}$ **do**
3 $\quad H_\ell \leftarrow \varnothing$
4 $\quad$ **for** *each graph* $G \in \mathcal{G}_L : \ell(G) = i$ **do**
5 $\quad\quad H_G \leftarrow \varnothing$
6 $\quad\quad$ **for** *each edge* $e \in E(G)$ **do**
7 $\quad\quad\quad H_G \leftarrow H_G \cup \phi_e(e)$
8 $\quad\quad H_\ell \leftarrow H_\ell \cup \text{bundle}(H_G)$
9 $\quad \mathbf{m}_i \leftarrow \text{bundle}(H_\ell)$
10 **return** $\{\mathbf{m}_1, \ldots, \mathbf{m}_k\}$

Circular hypervectors

We now introduce a new type of basis-hypervector set called *circular-hypervectors*. These vectors allow the encoding of circular data, an important type of information never before addressed in HDC. This new set of hypervectors is also the core component of our second example: the dynamic hash table proposed in "Hyperdimensional hashing" section.

Motivation

As described in "Basis-hypervectors" section, symbols and real numbers can be represented in the hyperspace with random and level-hypervector basis sets, respectively. However, not every type of data falls into these two categories. Consider, for instance, angular data in $\Theta = [0, 2\pi]$. The distance $\rho \in [0, 1]$ between two angles α and β in Θ is defined as [67]:

$$\rho(\alpha, \beta) = \frac{1}{2}(1 - \cos(\alpha - \beta)) \tag{4}$$

If we use level-hypervectors to encode the Θ-interval, the distances between the hypervectors would not be proportional to the distance between the angles. Notice that the

endpoints of an interval represented with level-hypervectors are completely dissimilar, while a circle has no endpoints.

Angles are widely used to represent information in meteorology [37], ecology [53, 108], medicine [2, 19, 20], astronomy [9, 70] and engineering [11]. Moreover, many natural and social phenomena have circular-linear correlation on some time scale. Consider for example the seasonal temperature variations over a year or the behavior of fish with respect to the tides in a day. In these cases, it makes sense to represent the time intervals (e.g., Jan 1st–Dec 31st) as cyclic intervals [53, 67].

Given the multitude of applications using circular data, unsurprisingly there has been great scientific effort to adapt statistical and learning methodologies to handle it appropriately [65]. This gave rise to a branch of statistical methodology known as *directional statistics* [69, 82]. Despite all this effort, to the best of our knowledge, our work is the first to address the adaptation in the context of HDC.

Background

Level-hypervectors are created by quantizing an interval to m levels and assigning a hypervector to each. The similarity between hypervectors is proportional to the distance between the intervals. For the level-hypervectors proposed by Nunes et al. [77] this correlation is achieved by assigning random d-dimensional hypervectors to the endpoints of the interval, intermediate intervals are then obtained by interpolating the endpoint hypervectors $\mathbf{l}_1$ and $\mathbf{l}_m$. To perform the interpolation a single random d-dimensional vector $\mathbf{f}$ is sampled with elements $\mathrm{f}_i \sim \mathcal{U}(0,1)$. This random vector determines from which endpoint each element of each intermediate hypervector is taken as shown in Eq. 5, where the indicator function $\mathbf{1}(\cdot)$ is applied element-wise.

$$\mathbf{l}_l = \mathbf{1}(\mathbf{f} < \tau)\mathbf{l}_1 + \mathbf{1}(\mathbf{f} \geq \tau)\mathbf{l}_m \, \forall l \in \{2, \ldots, m-1\} \quad \text{where } \tau = \frac{m-l}{m-1} \tag{5}$$

Rachkovskiy et al. [88] discuss a method to encode cyclic quantities by interpolating multiple random hypervectors. Their method starts by sampling at least three random hypervectors that represent points on a circle. They then use a technique to create level-hypervectors to interpolate the random hypervectors. This, however, does not result in a set of hypervectors whose similarities is proportional to the distance between angles as defined in Eq. 4 because at least a third of the circle is orthogonal to any point. Alternatively, when using the FHRR model, the phase distribution of the complex elements can be constructed such that fractional binding, the process of encoding real numbers with FHRR by exponentiating the complex elements, results in periodic hypervectors [18].

Method

Circular-hypervectors are an extension to level-hypervectors that eliminate the discontinuity in similarity between the last and first interval, as visualized in the similarity profiles in Fig. 1. By removing the discontinuity, the hypervectors become a set with circular correlation.

The creation of circular-hypervectors, shown in Fig. 5, is divided into two phases, one for each half of the circle. The first half is simply a set of $m/2 + 1$ level-hypervectors:

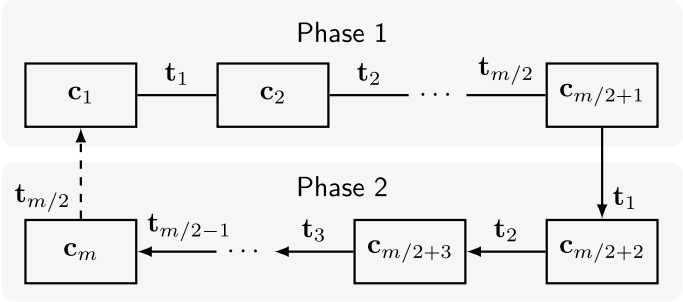

Fig. 5 Illustration of the process to create circular-hypervectors. Phase 1 shows the level-hypervectors and the transformations between them. Phase 2 shows the use of transformations for the second half of the circle. The dashed transformation is shown to complete the circle but is not needed since $\mathbf{c}_1$ is already known

$$\mathbf{c}_l = \mathbf{l}_l \text{ for } l \in \{1, \ldots, m/2 + 1\}$$

where $\mathbf{c}_1$ and $\mathbf{c}_{m/2+1}$ are quasi-orthogonal, ensuring that the opposite side of the circle is completely dissimilar. The second half is created by applying the inverse of the transitions between the levels of the first half, in order, to the last circular-hypervector:

$$\mathbf{c}_l = \mathbf{c}_{l-1} \oslash \mathbf{t}_{l-m/2-1} \text{ for } l \in \{m/2 + 2, \ldots, m\} \tag{6}$$

where the transition $\mathbf{t}_l = \mathbf{c}_{l+1} \oslash \mathbf{c}_l$ are the flipped bits between levels l and $l + 1$ for the BSC model. The combined transitions $\{\mathbf{t}_1, \ldots, \mathbf{t}_{m/2}\}$ are equal to the transition from $\mathbf{c}_1$ to $\mathbf{c}_{m/2+1}$ such that:

$$\mathbf{c}_{m/2+1} = \mathbf{c}_1 \otimes \bigotimes_{l=1}^{m/2} \mathbf{t}_l$$

The transitions unbound to $\mathbf{c}_{l-1}$ in Eq. 6 make the new hypervector $\mathbf{c}_l$ closer to $\mathbf{c}_1$. Moreover, the transitions $\{\mathbf{t}_1, \ldots, \mathbf{t}_{m/2}\}$ occur in any half of the circle, ensuring that the hypervector at the opposite side from any point is always quasi-orthogonal to it. For ease of understanding, we assume that m is even. To generate a set of circular-hypervectors with odd cardinality, simply generate $2m$ and return just $\{\mathbf{c}_1, \mathbf{c}_3, \mathbf{c}_5, \ldots, \mathbf{c}_{2\,m}\}$.

Hyperdimensional hashing

Most cloud services and distributed applications rely on efficient and robust hashing algorithms that allow dynamic scaling of the hash table. Examples include AWS, Google Cloud and BitTorrent. Consistent and rendezvous hashing are famous algorithms that minimize key remapping as the hash table resizes. While memory errors in large-scale cloud deployments are common, neither algorithm offers both efficiency and robustness. In this context, in our second example we design an HDC method for this problem, seeking to exploit the fact that the limitations of existing methods coincide exactly with the problems that HDC is intended to solve. Remember that one of the main goals here is to show that HDC is applicable both to learning problems (as in "GraphHD" section) and to other computational problems like hashing.

Our proposed method is called Hyperdimensional (HD) hashing and uses the circular-hypervectors presented above. HD hashing leverages the previously mentioned

advantages of HDC such as the efficiency of comparing hypervectors and the robustness in representing information. We show that it has the efficiency to be deployed in large systems. Moreover, a realistic level of memory errors causes more than 20% mismatches for consistent hashing while HD hashing remains unaffected.

Motivation

An important problem in many cloud services and distributed network applications is the process of mapping requests to available resources. Example systems include: load balancing in cloud data centers, web caching, peer-to-peer (P2P) services, and distributed databases. Difficulty arises in such highly dynamic systems because resources join and leave the cluster at any time, due for example to cloud elasticity [3], server failures, or availability of peers in a P2P network. It is often desirable to distribute requests evenly among resources and to minimize the number of redistributed requests when a resource joins or leaves. A non-uniform mapping results in overloading of resources and critical failure points.

The simplest hash table solves the mapping problem using modular hashing. Despite having a great lookup time complexity of $\mathcal{O}(1)$, a change in table size (number of available resources) requires virtually all requests to be redistributed due to the modulo operation (more details in "Background" section). *Consistent hashing* [50] and *rendezvous hashing* [106] are alternative hashing algorithms that minimize redistribution when the hash table is resized. They prevent resource overloading at the cost of increased lookup time, $\mathcal{O}(\log n)$ and $\mathcal{O}(n)$ respectively.

However, we show that when considered in a dynamic environment subject to errors and failures, referred to as noise, the performance of consistent hashing and rendezvous hashing in minimizing the number of redistributed requests degrades. Noise can be introduced in many aspects of a system. We focus on memory errors which can for instance be caused by soft errors in the form of single event upsets (SEU), multi-cell upsets (MCUs) or hard errors [38, 103]. MCUs, or burst errors, occur during a single event and are becoming more common as the feature size decreases. For 22 nm technology MCUs are estimated to be 45% of all SEUs [39]. Moreover, analysis of memory failures in Google's data centers revealed that each year a third of the machines experiences a memory error [96]. More robust hashing alternatives make it possible for cloud providers to perform fewer memory swaps, reducing operation cost.

Fueled by the demonstrated properties of HDC and the aforementioned limitations of current hashing algorithms, we propose *Hyperdimensional (HD) hashing*, a new HDC-based dynamic hashing algorithm. HD hashing scales similarly to consistent hashing while proving to be much more efficient than rendezvous hashing. HDC's highly parallelizable operations have been exploited in recent research, showing that special hardware can make HD hashing far superior in efficiency (more details in "Existing solutions" section). Moreover, we show that our algorithm is significantly more robust against noise. With 512 servers and a 10-bit MCU, HD hashing is unaffected while rendezvous and consistent hashing mismatch 4% and 12% of requests, respectively. With MCUs becoming more common this poses a risk for critical failures.

Background

Consistent Hashing Consistent hashing is a common way of distributing requests among a changing population of servers [71, 104]. Often times, the problem and the technique are referred to as *consistent hashing* indistinctly. The algorithm, which gave rise to Akamai [80], is used in many other real-world large scale applications such as Dynamo on Amazon Web Services [14] and Google Cloud Platform [16].

To describe consistent hashing, let $h(\cdot)$ denote a hash function that takes requests as inputs (in practice an IP address or unique identifier, for example) and $S = \{s_1, \ldots, s_n\}$ a set of servers. In modular hashing, a request r is simply assigned to s_i where $i = h(r) \bmod n$. Instead, consistent hashing maps both requests and servers uniformly to the unit interval [0, 1], which is interpreted as a circular interval. Thereafter, each request is assigned to the first server that succeeds it on the circle in clockwise order. This assignment is usually done in $\mathcal{O}(\log n)$ time using binary search.

Rendezvous Hashing The basic idea of rendezvous hashing [106], also known as highest random weight (HRW) hashing, is very simple. Given a hash function $h(\cdot)$ that takes as input a server and a request, each request r is assigned to the server s_i where:

$$i = \arg\max_{s \in S} h(s, r)$$

Each assignment is therefore done in $\mathcal{O}(n)$ time, since it is necessary to compute the hash of the request paired with each server in the system in order to compute the maximum value. In practice, Rendezvous hashing is used less often than consistent hashing, despite distributing the requests more uniformly, because of the increased time complexity.

Method

HD hashing, illustrated in Fig. 6, draws inspiration from consistent and rendezvous hashing, but seeks a solution that is both robust and efficient by translating the problem into a hyperdimensional computing task.

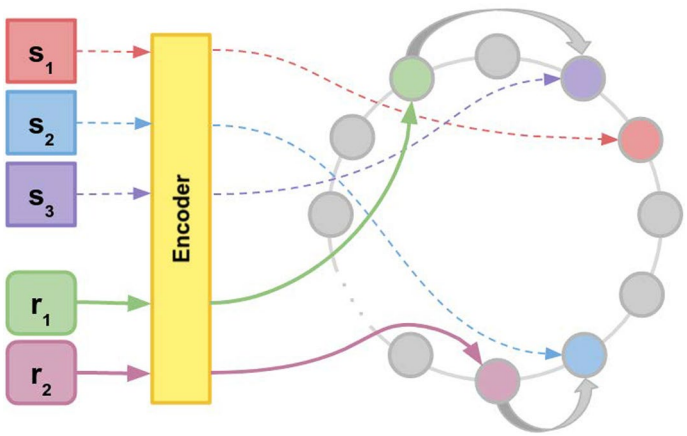

Fig. 6 Illustration of the operation of HD hashing. In this example, after encoding each of the three servers and two requests to a circular-hypervector, r_1 is assigned to server s_3, which is the server whose hyperspace representation is closest to its. Likewise, r_2 is assigned to s_2. Note that, unlike consistent hashing, the direction of rotation does not matter

Let $S = \{s_1, \ldots, s_k\}$ be a set of k servers, $R = \{r_1, \ldots, r_\ell\}$ a set of ℓ requests and $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$ a set of $n > k$ hypervectors. We also denote by $h(\cdot)$ a hash function that takes as input a server or request. The process of adding servers to the system in HD hashing is similar to consistent hashing, but instead of mapping them to a unit interval (see "Background" section), HD hashing assigns (or "*encodes*" in HDC terminology) each server to a hypervector. To distribute requests among servers, HD hashing also encodes each request. Let us represent this encoding by the function $\phi : S \cup R \to C$. Then, HD hashing encodes every server and request as follows:

$$\phi(x) \;=\; C\big[h(x) \bmod n\big] \tag{7}$$

where x is either a server or a request and $C[h(x) \bmod n]$ denotes the hypervector at position $h(x) \bmod n$ in C.

With all servers and requests encoded to the hyperspace, each request r_i is mapped to server s_j, with:

$$j = \arg\max_{s \in S} \delta\big(\phi(s), \phi(r_i)\big) \tag{8}$$

where δ is a given similarity metric between a pair of hypervectors such as inverse Hamming distance or the cosine similarity as discussed in "Similarity metrics" section. The operation above is the one mentioned in "Associative memory" section, and it is called inference using associative memory. This is exactly the operation that Schmuck et al. [95] show to be optimizable to the extreme of a single clock-cycle in special hardware. In other words, by using hardware accelerators for HDC each mapping in HD hashing could be executed in $\mathcal{O}(1)$ time.

One remaining, but crucial, question is: how do we create the set of hypervectors C? Similar to consistent hashing, we map servers and requests onto a circle. We then map the request to the server that is assigned to the nearest node on the circle according to Eq. 8. To accomplish this, we use the circular-hypervectors introduced in "Circular hypervectors" section such that the closer a node is on the circle the more similar its hypervector.

Elaboration

Here we present the empirical results for the examples of HDC applications presented in "Proposed solutions" section. The goal is to demonstrate how HDC allows creating solutions to problems, both in machine learning and other computational problems, providing a balance between accuracy, efficiency and robustness.

Experimental setup

All the experiments were implemented using Torchhd [33], a high-performance Python library for HDC, and were performed on the same machine with 20 Intel Xeon Silver 4114 CPUs at 2.20 GHz with 93 GB of RAM and 4 Nvidia TITAN Xp GPUs with 3840 cores and 12 GB running CentOS Linux.

To evaluate GraphHD, two groups of experiments were conducted. First, we adopt six graph classification datasets to evaluate the accuracy, training times, and inference times. All of these benchmarks are part of TUDataset, a collection of datasets and

standardized evaluation procedures widely used in the empirical evaluation of graph classification methods [72]. The performance of GraphHD is compared to two kernel methods and two graph neural networks. Secondly, the scaling profile of GraphHD is empirically assessed and presented in comparison to a GNN and a kernel method.

As baseline methods for comparison, two state-of-the-art GNNs and two kernel-based methods were used. The methods are the most recently published that are available for standardized evaluation in the TUDataset repository. The two selected GNN methods are GIN-ϵ [115] and GIN-ϵ-JK [116]. For both GNN methods the network architecture was fixed for all experiments at 1 layer with 32 units. We found that this is the smallest network size that matches or exceeds the accuracy of GraphHD on all datasets. We use the Adam optimizer with a learning rate scheduler starting at 0.01 with a patience parameter of 5 which decays with 0.5 till a minimum of 10^{-6}, and the batch size is 128.

For the kernel methods baseline *Weisfeiler-Lehman Subtree* (1-WL) [98] and *Weisfeiler-Lehman Optimal Assignment* (WL-OA) [60] were used. As part of the training process the C-parameter of the kernels are selected from $\{10^{-3}, 10^{-2}, \ldots, 10^{2}, 10^{3}\}$ and the number of iterations from $\{0, \ldots, 5\}$. The training hyper-parameters, except for the fixed size GNN architecture, were taken from the reference baseline experiments. GraphHD uses 10,000-dimensional bipolar hypervectors in all the experiments. We fix the number of PageRank iterations to 10 for all experiments because the accuracy of GraphHD has then plateaued. The PageRank batch size is 256.

GraphHD is expected to significantly outperform existing methods on training time. The experiment gives insight into exactly how much faster and how much accuracy is traded-off for the increase in training speed. Since GraphHD only takes the structure of the graph into account to ensure that the method is broadly applicable, we restrict the GNNs and kernel methods from utilizing the vertex and edge labels when they are available.

Datasets

The selected datasets, with the exception of DD, contain very small graphs, with an average of less than 40 vertices. The graphs in the selected datasets are also very sparse, the average fraction of connected vertices is 0.05. The dataset containing the largest graphs, DD, should give an indication on real data of how well the learning methods scale. An overview of the statistics of the datasets used is shown in Table 3. ENZIMES [6] is a dataset of protein structures, and the task is to assign each enzyme to one of 6 Enzyme Commission (EC) top-level classes. MUTAG [13] has graphs representing mutagenic aromatic and heteroaromatic nitro compounds with 7 labels.

Table 3 Statistics of graph datasets

Dataset	Graphs	Classes	Avg. vertices	Avg. edges
DD	1178	2	284.32	715.66
ENZYMES	600	6	32.63	62.14
MUTAG	188	2	17.93	19.79
NCI1	4110	2	29.87	32.3
PROTEINS	1113	2	39.06	72.82
PTC_FM	349	2	14.11	14.48

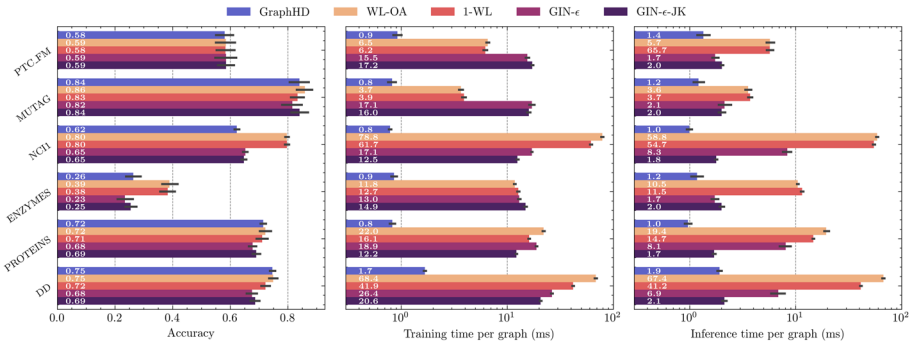

Fig. 7 Left: accuracy achieved on six datasets by GraphHD compared to that of the kernel methods, 1-WL and WL-OA, and the graph neural networks, GIN-ϵ and GIN-ϵ-JK. Middle: training time in seconds of the five learning methods for each of the six datasets. Note that the y-axis is in logarithmic scale. Right: average inference time for a graph in each of the six datasets compared across the five learning methods. Note that the y-axis is in logarithmic scale

NCI1 [110] is a set of data from the National Cancer Institute (NCI) and the task is to classify chemical compounds according to their ability to inhibit cancerous cell multiplication. In the graph datasets PROTEINS [15] and DD [15], the task is to classify whether or not the represented proteins are non-enzymes. Finally, PTC_FM [34] is a dataset from the Predictive Toxicology Challenge containing a collection of chemical compounds represented as graphs which report the carcinogenicity for rats.

For the HD Hashing experiments, we created a purpose build emulation framework to empirically verify our results. The emulator consists of two modules, a hash table and a generator. The generator emulates the requests from the outside world being sent to the hash table. The hash table module reads incoming requests from a buffer and uses a hashing algorithm to map them to an available server. Servers are added and removed using two special case requests, a `join` and `leave` request, respectively, with a unique identifier of the server. This functional emulator can be used to determine the computational efficiency of various hashing algorithms as well as their robustness to memory errors as we will describe next.

Since we do not have access to specialized HDC hardware and building the hardware is outside the scope of our work we had to implement the HDC operations using commodity hardware. To closely match the parallel nature of HDC hardware, we decided to implement HDC operations on a GPU. The GPU's communication overhead was reduced by performing mappings in batches of 256 requests. Each test was performed with different numbers of servers in the pool, going up to 2048. This scale is enough to show the results and trends of interest, but it is important to emphasize that like the other methods HD hashing can scale to much larger clusters. It can even be used hierarchically, which is a standard way to scale such hashing systems [97, 111] to handle extremely high numbers of servers.

Accuracy

We compare the accuracy and the training and inference times of GraphHD on six datasets from the TUDataset [72] collection against four methods. We use tenfold cross validation because the datasets contain relatively few graphs. We report training

and inference time per graph to normalize over varying dataset lengths. The wall-time for onefold of training is considered the training time. The inference time is set to be the testing wall-time of onefold. Measurements are averaged over 3 repetitions of tenfold cross validation. The accuracy results, shown on the left in Fig. 7, show that GraphHD has comparable accuracy to the baseline methods, except for NCI1 and ENZYMES where the kernel methods respectively do 18% and 12% better than both GraphHD and the GNN methods.

In the second accuracy experiment we tested how uniform the distribution of requests among servers is and how uniform they remain when bits of the hash values are randomly inverted in memory. For evaluation, we used the following *Pearson's chi-squared test* [30] to measure goodness of fit between our observed frequency distribution and the uniform distribution:

$$\chi^2 = \sum_{s_i \in S} \frac{\left(R(s_i) - E\right)^2}{E}$$

where $R(s_i)$ is the number of requests mapped to server s_i by the algorithm and $E = \frac{|R|}{|S|}$ is the uniformity expectation where $|R|$ and $|S|$ are the total number of requests and servers, respectively. The results, illustrated in Fig. 8, show that not only does HD hashing distribute requests more uniformly than consistent hashing in an ideal scenario, but also that the presence of bit errors worsens the uniformity of consistent hashing even more, while that of HD hashing remains intact. To make the plot more readable, we omit the rendezvous hashing result. Note, from the description of the algorithm in "Background" section, that rendezvous hashing is based only on the output of the hash function, that is, a pseudo-random number. Therefore, its assignment is perfectly (pseudo-)uniform and is not affected by bit errors. Rendezvous hashing, however, still suffers from mismatches and the method has less applicability due to its lower efficiency as illustrated in Figs. 10 and 11, respectively.

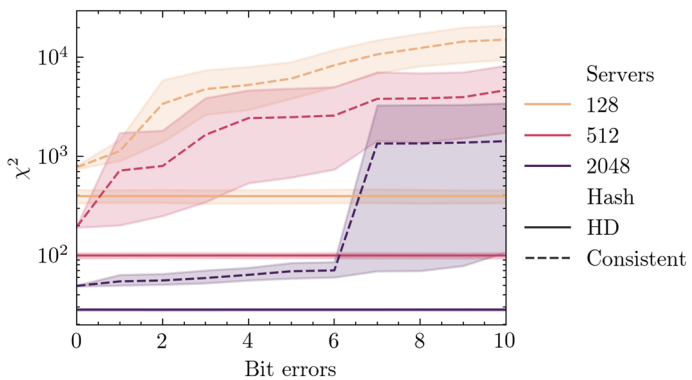

Fig. 8 The discrepancy between the distribution of requests per server obtained by each algorithm and the uniform distribution, for different numbers of servers and bit errors, measured with the Pearson's χ^2 statistical test

Efficiency

The training time results, shown in the middle of Fig. 7, confirm the notion that HDC is more computationally efficient than the other learning methods. GraphHD trains significantly faster than both the kernel and GNN methods on every dataset. On the DD dataset, which contains the largest graphs, GraphHD trains 12.1× faster than the GNNs and 24.6× faster than the fastest kernel method. Moreover, on the largest dataset, NCI1, GraphHD trains 77.1× faster than the kernel methods. Confirming that with respect to the dataset size the kernel methods have inferior scaling. The inference time of GraphHD, shown on the right in Fig. 7, is also faster than the other methods for every dataset. On the DD dataset the fastest GNN is 10.5% slower and the kernel methods are 21.7× slower.

To confirm the superior computational efficiency of HDC, the scalability experiment looks at how training time relates to the size of the graphs in the dataset. We create synthetic datasets with 2 classes evenly split over 100 graphs with varying numbers of vertices using the Erdös-Rényi random graph model [27]. The edge probability is set to 0.05, which is similar to the average connections in the real-world datasets as derived from the dataset statistics in Table 3. GraphHD is compared against one GNN and one kernel method, GIN-ϵ and WL-OA, respectively. The methods use the same hyper-parameters as described in "Experimental setup" section.

The scaling profile of GraphHD, as shown in Fig. 9, is up to an order of magnitude lower than that of the baseline methods as the graphs' sizes increase. On the largest measured graphs with 980 vertices, GraphHD is 6.2× faster than GIN-ϵ and 15.0× faster than WL-OA. The faster training and inference times allow for more graph learning applications to become feasible. These findings are consistent with those from the training times on the real-world datasets from the accuracy and training time experiment described in "Experimental setup" section. It is worth to remind the potential of HDC dedicated hardware to further improve the training and inference times of GraphHD as discussed in "Background" section.

We executed each hashing function in our emulator to empirically determine its computational efficiency. First the generator sends n `join` requests to add available servers

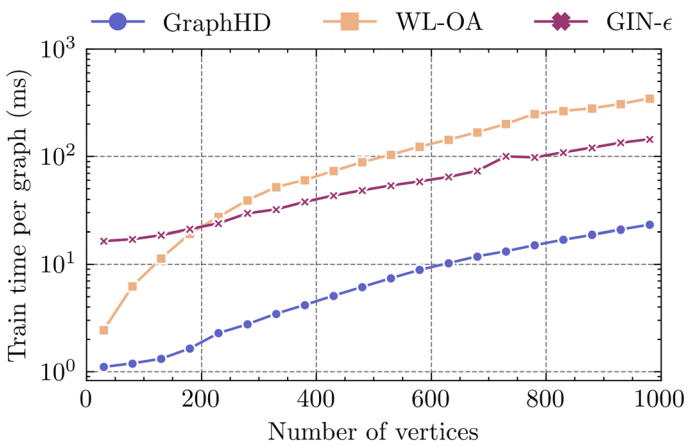

Fig. 9 Scaling profile of GraphHD compared to the graph neural network GIN-ϵ and the kernel method WL-OA. The y-axis is in logarithmic scale

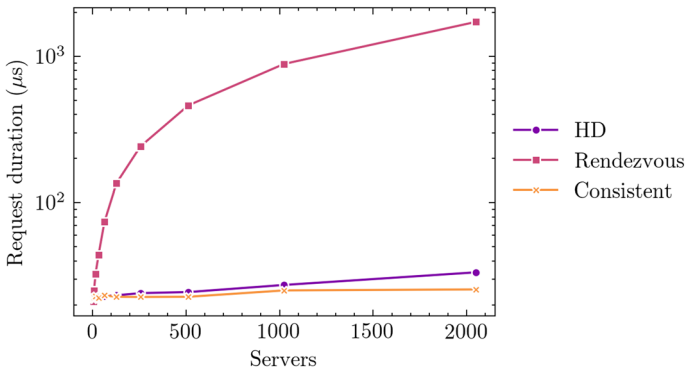

Fig. 10 Average request handling duration as the number of servers in the pool increases

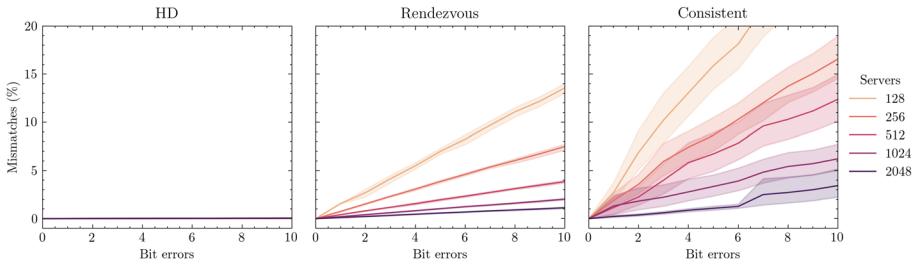

Fig. 11 Percentage of mismatched requests when a number of bit errors occur

to the hash table module. Then, the generator sends 10,000 requests and tracks the wall-time. From this we determine the average time to handle a request. For various numbers of n, ranging from 2 to 2048 in powers of 2 the results are shown in Fig. 10. The $\mathcal{O}(n)$ time complexity of rendezvous hashing is clearly evident as is the superior computational efficiency of consistent hashing with respect to rendezvous hashing. Our HDC implementation using commodity hardware has a very similar scaling profile to consistent hashing. This confirms our belief that HDC hardware can appropriately be simulated by a GPU. However, as highlighted in "Method" section, we expect the use of HDC accelerators to reduce the request handling time to a constant with the extreme of a single clock-cycle.

Robustness

As motivated before, the other main goal of HD hashing is to be a robust alternative to consistent and rendezvous hashing. In order to assess the performance of each hashing algorithm in an environment subject to noise, two experiments were performed using the emulator's noise injection capabilities. The first and most important, whose results are in Fig. 11, shows how the ability of each technique to map keys to the correct value degrades when a certain number of bits in memory are randomly flipped. Ibe et al. [39] show that for 22 nm technology, 4-bit and 8-bit bursts occur 10% and 1% of the time, respectively. Moreover, errors within a machine are found to be strongly correlated, if a machine experienced an error it is 13–228 times more likely to experience another error in the same month [96]. To capture such features of a realistic scenario, we test each hashing technique in the range of 0 to 10 bit flips.

In our experiments, HD hashing confirmed our expectations, turning out to be far superior as none of the requests sent were matched to the wrong server. Meanwhile, in both consistent and rendezvous hashing an increasing percentage of mismatches occur, depending on the noise level.

Conclusion

We present Hyperdimensional Computing (HDC), also called Vector Symbolic Architectures (VSA), as an emerging computational framework. Having attracted increasing attention, HDC is envisioned to reach its full potential as an abstraction layer between new hardware platforms and algorithms. The model takes advantage of geometric and arithmetic properties of high-dimensional vector spaces to create solutions to problems balancing accuracy, efficiency and noise tolerance. While most of the research in the field has been focused on machine learning applications, the use of HDC in other tasks is also becoming increasingly attractive.

In this article we present a detailed introduction to HDC, aiming at both a survey and a tutorial role. In order to illustrate the generality of the framework, we present two examples of solutions based on HDC for very different problems, one related to machine learning and the other a classic algorithm with stochastic behavior.

First, we present a graph classification method called GraphHD. We show that GraphHD achieves comparable accuracy while proving to be significantly more efficient. Remarkably when the graphs increase in size, the scaling profile of GraphHD is much more favorable, opening up possibilities for graph classification on large graphs that were previously not computationally feasible. We introduced a baseline graph encoding algorithm that makes it possible to use HDC for graph learning applications. The results of GraphHD are promising and indicate the importance of continuing to investigate HDC algorithms for graph learning as a light-weight, robust and scalable alternative to deep learning, especially in resource constrained applications such as embedded devices and IoT.

In the second example, we seek to show the usefulness of HDC in a non-learning setting. We propose HD hashing, a new hashing algorithm which allows dynamic scaling of the hash table with minimal rehashing, a problem found in some of the most popular web applications. Through an emulation framework, we compare our method with consistent and rendezvous hashing and the experimental results show that HD hashing is the only approach that guarantees both efficiency and robustness. HD hashing scales similar to consistent hashing, while both are significantly more efficient than rendezvous hashing. Consistent hashing suffers from more than 20% mismatches with a realistic level of memory errors, which are common in large-scale cloud systems, while HD hashing remains unaffected. This superior level of tolerance to bit errors reduces the chance of critical failures in load balancing and web caching systems, among others.

This paper seeks to describe this emerging area of research in a didactic way, explaining each element that makes up HDC in depth and through illustrative examples. In addition, each discussion is accompanied by a review of the main papers that addressed each topic. We believe that this, along with the examples and discussion of the generality of the model for both learning and non-learning applications, makes this paper valuable material for researchers in the field or those looking to get started in it.

Acknowledgements
Not applicable

Author contributions
M.H. and I.N. wrote the main manuscript text. All authors read and approved the final manuscript.

Funding
Not applicable.

Availability of data and materials
The datasets analysed during the current study are available in the TUDatasets repository, https://chrsmrrs.github.io/datasets/docs/datasets/.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 19 April 2024 Accepted: 13 October 2024
Published online: 24 October 2024

References

1. Abusnaina A, Khormali A, Alasmary H, Park J, Anwar A, Mohaisen A. Adversarial learning attacks on graph-based IoT malware detection systems. In: 2019 IEEE 39th international conference on distributed computing systems (ICDCS). IEEE; 2019. p. 1296–305.
2. Ahmidi N, Tao L, et al. A dataset and benchmarks for segmentation and recognition of gestures in robotic surgery. Trans Biomed Eng. 2017;64:2025–41.
3. Al-Dhuraibi Y, Paraiso F, Djarallah N, Merle P. Elasticity in cloud computing: state of the art and research challenges. IEEE Trans Serv Comput. 2017;11:430–47.
4. Alonso P, Shridhar K, Kleyko D, Osipov E, Liwicki M. Hyperembed: tradeoffs between resources and performance in NLP tasks with hyperdimensional computing enabled embedding of n-gram statistics. In: 2021 international joint conference on neural networks (IJCNN). 2021; IEEE. p. 1–9.
5. Arrieta AB, Díaz-Rodríguez N, Del Ser J, Bennetot A, Tabik S, Barbado A, García S, Gil-López S, Molina D, Benjamins R, et al. Explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. Inf Fusion. 2020;58:82–115.
6. Borgwardt KM, Ong CS, Schönauer S, Vishwanathan S, Smola AJ, Kriegel HP. Protein function prediction via graph kernels. Bioinformatics. 2005;21:i47–56.
7. Branco S, Ferreira AG, Cabral J. Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: a survey. Electronics. 2019;8:1289.
8. Brin S, Page L. The anatomy of a large-scale hypertextual web search engine. Comput Netw ISDN Syst. 1998;30:107–17.
9. Cabella P, Marinucci D. Statistical challenges in the analysis of cosmic microwave background radiation. Ann Appl Stat. 2009;3:61–95.
10. Chen Y, Zheng B, Zhang Z, Wang Q, Shen C, Zhang Q. Deep learning on mobile and embedded devices: state-of-the-art, challenges, and future directions. ACM Comput Surv. 2020;53:1–37.
11. Chirikjian GS. Engineering applications of noncommutative harmonic analysis: with emphasis on rotation and motion groups. Boca Raton: CRC Press; 2000.
12. Dasgupta S, Stevens CF, Navlakha S. A neural algorithm for a fundamental computing problem. Science. 2017;358:793–6.
13. Debnath AK, Lopez de Compadre RL, Debnath G, Shusterman AJ, Hansch C. Structure–activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. J Med Chem. 1991;34:786–97.
14. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. ACM SIGOPS Oper Syst Rev. 2007;41:205–20.
15. Dobson PD, Doig AJ. Distinguishing enzyme structures from non-enzymes without alignments. J Mol Biol. 2003;330:771–83.
16. Eisenbud DE, Yi C, Contavalli C, Smith C, Kononov R, Mann-Hielscher E, Cilingiroglu A, Cheyney B, Shang W, Hosein JD. Maglev: a fast and reliable software network load balancer. In: 13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16; 2016. p. 523–35.
17. Ferrer-Cid P, Barcelo-Ordinas JM, Garcia-Vidal J. Graph learning techniques using structured data for IoT air pollution monitoring platforms. IEEE Internet Things J. 2021;8(17):13652–63.
18. Frady EP, Kleyko D, Kymn CJ, Olshausen BA, Sommer FT. Computing on functions using randomized vector representations. arXiv preprint. 2021. arXiv:2109.03429 .

19. Gao F, Chia KS, Krantz I, Nordin P, Machin D. On the application of the von Mises distribution and angular regression methods to investigate the seasonality of disease onset. Stat Med. 2006;25:1593–618.
20. Gao Y, et al. Jhu-isi gesture and skill assessment working set (jigsaws): a surgical activity dataset for human motion modeling. In: MICCAI workshop: M2cai; 2014. p. 3.
21. Gayler R. Multiplicative binding, representation operators and analogy. In: Advances in analogy research. 1998. p. 1–4.
22. Gayler RW. Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience. arXiv preprint. 2004. cs/0412059.
23. Gayler RW, Levy SD. A distributed basis for analogical mapping. In: New frontiers in analogy research; 2009.
24. Ge L, Parhi KK. Classification using hyperdimensional computing: a review. Circ Syst Mag. 2020;20:30–47.
25. Ge L, Parhi KK. Seizure detection using power spectral density via hyperdimensional computing. In: ICASSP 2021–2021 IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE; 2021. p. 7858–62.
26. van Gelder T. Distributed vs. local representation. In: MIT encyclopedia of the cognitive sciences. 1999. p. 235–7.
27. Gilbert EN. Random graphs. Ann Math Stat. 1959;30:1141–4.
28. Gleich DF. Pagerank beyond the web. SIAM Rev. 2015;57:321–63.
29. Gori M, Monfardini G, Scarselli F. A new model for learning in graph domains. In: International joint conference on neural networks (IJCNN). 2005; IEEE. p. 729–34.
30. Greenwood PE, Nikulin MS. A guide to chi-squared testing, vol. 280. New York: Wiley; 1996.
31. Hassan E, Halawani Y, Mohammad B, Saleh H. Hyper-dimensional computing challenges and opportunities for AI applications. IEEE Access. 2021;10:97651–64.
32. Heddes M, Nunes I, Givargis T, Nicolau A, Veidenbaum A. Hyperdimensional hashing: a robust and efficient dynamic hash table. In: Design automation conference (DAC). 2022.
33. Heddes M, Nunes I, Vergés P, Kleyko D, Abraham D, Givargis T, Nicolau A, Veidenbaum A. Torchhd: an open source python library to support research on hyperdimensional computing and vector symbolic architectures. J Mach Learn Res. 2023;24:1–10.
34. Helma C, King RD, Kramer S, Srinivasan A. The predictive toxicology challenge 2000–2001. Bioinformatics. 2001;17:107–8.
35. Hernández-Cano A, Zhuo C, Yin X, Imani M. Reghd: robust and efficient regression in hyper-dimensional learning system. In: Design automation conference (DAC). IEEE; 2021. p. 7–12.
36. Hinton GE. Distributed representations. 1984.
37. Holzmann H, Munk A, Suster M, Zucchini W. Hidden Markov models for circular and linear-circular time series. Environ Ecol Stat. 2006;13:325–47.
38. Hwang AA, Stefanovici IA, Schroeder B. Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design. ACM SIGPLAN Not. 2012;47:111–22.
39. Ibe E, Taniguchi H, Yahagi Y, Shimbo KI, Toba T. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. IEEE Trans Electron Devices. 2010;57:1527–38.
40. Imani M, Gupta S, Rosing T. Ultra-efficient processing in-memory for data intensive applications. In: Design automation conference (DAC). IEEE; 2017. p. 1–6.
41. Imani M, Kong D, Rahimi A, Rosing T. Voicehd: hyperdimensional computing for efficient speech recognition. In: International conference on rebooting computing (ICRC). IEEE; 2017. p. 1–8.
42. Imani M, Nassar T, Rahimi A, Rosing T. Hdna: energy-efficient dna sequencing using hyperdimensional computing. In: 2018 IEEE EMBS international conference on biomedical & health informatics (BHI). IEEE; 2018. p. 271–4.
43. Jaeger H. Towards a generalized theory comprising digital, neuromorphic and unconventional computing. Neuromorphic Comput Eng. 2021;1: 012002.
44. Joshi A, Halseth JT, Kanerva P. Language geometry using random indexing. In: International symposium on quantum interaction (QI). Springer; 2016. p. 265–74.
45. Kanerva P. Sparse distributed memory. London: MIT press; 1988.
46. Kanerva P. Hyperdimensional computing: an introduction to computing in distributed representation with high-dimensional random vectors. Cogn Comput. 2009;1:139–59.
47. Kanerva P. What we mean when we say what's the dollar of Mexico?: prototypes and mapping in concept space. In: AAAI fall symposium series; 2010.
48. Kanerva P, et al. Fully distributed representation. PAT. 1997;1:10000.
49. Kang U, Tong H, Sun J. Fast random walk graph kernel. In: SIAM international conference on data mining, SIAM; 2012. p. 828–38.
50. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the twenty-ninth annual ACM symposium on theory of computing. 1997. p. 654–63.
51. Karunaratne G, Le Gallo M, Cherubini G, Benini L, Rahimi A, Sebastian A. In-memory hyperdimensional computing. Nat Electron. 2020;3:327–37.
52. Keerthi SS, Chapelle O, DeCoste D, Bennett KP, Parrado-Hernández E. Building support vector machines with reduced classifier complexity. J Mach Learn Res. 2006;7:1493–515.
53. Kempter R, Leibold C, et al. Quantifying circular-linear associations: hippocampal phase precession. J Neurosci Methods. 2012;207:113–24.
54. Khan R, Khan SU, Zaheer R, Khan S. Future internet: the internet of things architecture, possible applications and key challenges. In: International conference on frontiers of information technology (FIT). IEEE; 2012. p. 257–60.
55. Kleyko D, Davies M, Frady EP, Kanerva P, Kent SJ, Olshausen BA, Osipov E, Rabaey JM, Rachkovskij DA, Rahimi A, et al. Vector symbolic architectures as a computing framework for emerging hardware. Proc IEEE. 2022;110:1538–71.
56. Kleyko D, Rachkovskij DA, Osipov E, Rahim A. A survey on hyperdimensional computing aka vector symbolic architectures, part II: applications, cognitive models, and challenges. arXiv preprint. 2021. arXiv:2112.15424 .

57. Kleyko D, Rachkovskij DA, Osipov E, Rahimi A. A survey on hyperdimensional computing aka vector symbolic architectures, part I: models and data transformations. arXiv preprint. 2021. arXiv:2111.06077 .
58. Kleyko D, Rahimi A, Gayler RW, Osipov E. Autoscaling bloom filter: controlling trade-off between true and false positives. Neural Comput Appl. 2020;32:3675–84.
59. Kondor R, Pan H. The multiscale Laplacian graph kernel. In: Advances in neural information processing systems (NIPS). 2016. p. 2982–90.
60. Kriege NM, Giscard PL, Wilson RC. On valid optimal assignment kernels and applications to graph classification. In: Advances in neural information processing systems (NIPS). 2016. p. 1615–23.
61. Kriege NM, Johansson FD, Morris C. A survey on graph kernels. Appl Netw Sci. 2020;5:1–42.
62. Kussul EM, Baidyk TN, Wunsch DC II, Makeyev O, Martin A. Permutation coding technique for image recognition systems. IEEE Trans Neural Netw. 2006;17:1566–79.
63. Lai L, Suda N. Enabling deep learning at the IoT edge. In: Proceedings of the international conference on computer-aided design. New York: ACM; 2018. p. 1–6. https://doi.org/10.1145/3240765.3243473.
64. Ledoux M. The concentration of measure phenomenon, vol. 89. Rhode Island: American Mathematical Soc; 2001.
65. Lee A. Circular data. Wiley Interdiscip Rev Comput Stat. 2010;2:477–86.
66. Li H, Wu TF, Rahimi A, Li KS, Rusch M, Lin CH, Hsu JL, Sabry MM, Eryilmaz SB, Sohn J, et al. Hyperdimensional computing with 3D VRRAM in-memory kernels: device-architecture co-design for energy-efficient, error-resilient language recognition. In: International electron devices meeting (IEDM). IEEE; 2016. p. 16–1.
67. Lund U. Least circular distance regression for directional data. J Appl Stat. 1999;26:723–33.
68. Manabat AX, Marcelo CR, Quinquito AL, Alvarez A. Performance analysis of hyperdimensional computing for character recognition. In: International symposium on multimedia and communication technology (ISMAC). IEEE; 2019. p. 1–5.
69. Mardia KV, Jupp PE, Mardia K. Directional statistics, vol. 2. London: Wiley Online Library; 2000.
70. Marinucci D, Peccati G. Random fields on the sphere: representation, limit theorems and cosmological applications, vol. 389. Cambridge: Cambridge University Press; 2011.
71. Mirrokni V, Thorup M, Zadimoghaddam M. Consistent hashing with bounded loads. In: Proceedings of the twenty-ninth annual ACM-SIAM symposium on discrete algorithms. SIAM; 2018. p. 587–604.
72. Morris C, Kriege NM, Bause F, Kersting K, Mutzel P, Neumann M. Tudataset: a collection of benchmark datasets for learning with graphs. In: ICML workshop on graph representation learning and beyond (GR+). 2020.
73. Najafabadi FR, Rahimi A, Kanerva P, Rabaey JM. Hyperdimensional computing for text classification. In: Design, automation test in Europe conference exhibition (DATE). 2016. p. 1–1.
74. Neubert P, Protzel P. Towards hypervector representations for learning and planning with schemas. In: Joint German/Austrian conference on artificial intelligence (Künstliche Intelligenz). Springer; 2018. p. 182–9.
75. Neumann M, Moreno P, Antanas L, Garnett R, Kersting K. Graph kernels for object category prediction in task-dependent robot grasping. In: KDD workshop on mining and learning with graphs (MGL). 2013. p. 0–6.
76. Nikolentzos G, Meladianos P, Vazirgiannis M. Matching node embeddings for graph similarity. In: Thirty-first AAAI conference on artificial intelligence. 2017.
77. Nunes I, Heddes M, Givargis T, Nicolau A. An extension to basis-hypervectors for learning from circular data in hyperdimensional computing. In: 2023 60th ACM/IEEE design automation conference (DAC). IEEE; 2023. p. 1–6.
78. Nunes I, Heddes M, Givargis T, Nicolau A, Veidenbaum A. Graphhd: efficient graph classification using hyperdimensional computing. In: Design, automation & test in Europe conference & exhibition (DATE). 2022.
79. Nunes I, Heddes M, Vergés P, Abraham D, Veidenbaum A, Nicolau A, Givargis T. Dothash: estimating set similarity metrics for link prediction and document deduplication. In: Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining. 2023. p. 1758–69.
80. Nygren E, Sitaraman RK, Sun J. The akamai network: a platform for high-performance internet applications. ACM SIGOPS Oper Syst Rev. 2010;44:2–19.
81. Osipov E, Kleyko D, Legalov A. Associative synthesis of finite state automata model of a controlled object with hyperdimensional computing. In: IECON 2017-43rd annual conference of the IEEE industrial electronics society. IEEE; 2017. p. 3276–81.
82. Pewsey A, García-Portugués E. Recent advances in directional statistics. TEST. 2021;30:1–58.
83. Plate TA. Distributed representations and nested compositional structure. Ph.D. thesis. University of Toronto; 1994.
84. Plate TA. Holographic reduced representations. IEEE Trans Neural Netw. 1995;6:623–41.
85. Plate TA. Holographic reduced representation: distributed representation for cognitive structures. Stanford: CSLI Publications; 2003.
86. Rachkovskij D, Fedoseyeva T. On audio signals recognition by multilevel neural network. In: Proceedings of the international symposium on neural networks and neural computing (NEURONET). 1990. p. 281–3.
87. Rachkovskij DA, Kussul EM. Binding and normalization of binary sparse distributed representations by context-dependent thinning. Neural Comput. 2001;13:411–52.
88. Rachkovskiy DA, Slipchenko SV, Kussul EM, Baidyk TN. Sparse binary distributed encoding of scalars. J Autom Inf Sci. 2005;37:12–23.
89. Rahimi A, Benatti S, Kanerva P, Benini L, Rabaey JM. Hyperdimensional biosignal processing: a case study for EMG-based hand gesture recognition. In: International conference on rebooting computing (ICRC). IEEE; 2016. p. 1–8.
90. Rahimi A, Datta S, Kleyko D, Frady EP, Olshausen B, Kanerva P, Rabaey JM. High-dimensional computing as a nanoscalable paradigm. Trans Circ Syst I Regular Pap. 2017;64:2508–21.
91. Rahimi A, Kanerva P, Rabaey JM. A robust and energy-efficient classifier using brain-inspired hyperdimensional computing. In: International symposium on low power electronics and design (ISLPED). 2016. p. 64–9.
92. Robertson AM, Willett P. Applications of n-grams in textual information systems. J Doc. 1998;54(1):48–67.
93. Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G. The graph neural network model. Trans Neural Netw. 2008;20:61–80.
94. Schlegel K, Neubert P, Protzel P. A comparison of vector symbolic architectures. Artif Intell Rev. 2021;55:1–33.

95. Schmuck M, Benini L, Rahimi A. Hardware optimizations of dense binary hyperdimensional computing: rematerialization of hypervectors, binarized bundling, and combinational associative memory. J Emerg Technol Comput Syst. 2019;15:1–25.
96. Schroeder B, Pinheiro E, Weber WD. Dram errors in the wild: a large-scale field study. ACM SIGMETRICS Perform Eval Rev. 2009;37:193–204.
97. Shen H, Xu CZ, Chen G. Cycloid: a constant-degree and lookup-efficient P2P overlay network. Perform Eval. 2006;63:195–216.
98. Shervashidze N, Schweitzer P, Van Leeuwen EJ, Mehlhorn K, Borgwardt KM. Weisfeiler-lehman graph kernels. J Mach Learn Res. 2011;12:2539–61.
99. Shridhar K, Jain H, Agarwal A, Kleyko D. End to end binarized neural networks for text classification. In: Proceedings of SustaiNLP: workshop on simple and efficient natural language processing. 2020. p. 29–34.
100. Simpkin C, Taylor I, Bent GA, de Mel G, Rallapalli S, Ma L, Srivatsa M. Constructing distributed time-critical applications using cognitive enabled services. Futur Gener Comput Syst. 2019;100:70–85.
101. Smith D, Stanford P. A random walk in hamming space. In: 1990 IJCNN international joint conference on neural networks. IEEE; 1990. p. 465–70.
102. Sokolov A, Rachkovskij D. Approaches to sequence similarity representation. J Inf Theor Appl. 2006;13(3):272–8.
103. Sridharan V, Liberty D. A study of dram failures in the field. In: SC'12: proceedings of the international conference on high performance computing, networking, storage and analysis. IEEE; 2012. p. 1–11.
104. Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H. Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans Netw. 2003;11:17–32.
105. Suri M. Advances in neuromorphic hardware exploiting emerging nanoscale devices. New Delhi: Springer; 2017.
106. Thaler DG, Ravishankar CV. Using name-based mappings to increase hit rates. IEEE/ACM Trans Netw. 1998;6:1–14.
107. Thomas A, Dasgupta S, Rosing T. Theoretical foundations of hyperdimensional computing. J Artif Intell Res. 2021;72:215–49.
108. Tracey J, Zhu J, Crooks K. A set of nonlinear regression models for animal movement in response to a single landscape feature. J Agric Biol Environ Stat. 2005;10:1–18.
109. Vergés P, Heddes M, Nunes I, Givargis T, Nicolau A. Classification using hyperdimensional computing: a review with comparative analysis. 2023.
110. Wale N, Karypis G. Comparison of descriptor spaces for chemical compound retrieval and classification. In: International conference on data mining (ICDM). 2006. p. 678–89. https://doi.org/10.1109/ICDM.2006.39.
111. Wang W, Ravishankar CV. Hash-based virtual hierarchies for scalable location service in mobile ad-hoc networks. Mobile Netw Appl. 2009;14:625–37.
112. Widdows D, Cohen T. Reasoning with vectors: a continuous model for fast robust inference. Logic J IGPL. 2015;23:141–73.
113. Wu TF, Li H, Huang PC, Rahimi A, Rabaey JM, Wong HSP, Shulaker MM, Mitra S. Brain-inspired computing exploiting carbon nanotube fets and resistive ram: hyperdimensional computing case study. In: International solid-state circuits conference-(ISSCC); IEEE. 2018. p. 492–4.
114. Wu Z, Pan S, Chen F, Long G, Zhang C, Philip SY. A comprehensive survey on graph neural networks. IEEE Trans Neural Netw Learn Syst. 2020;32(1):4–24.
115. Xu K, Hu W, Leskovec J, Jegelka S. How powerful are graph neural networks? In: International conference on learning representations (ICLR). 2019.
116. Xu K, Li C, Tian Y, Sonobe T, Kawarabayashi KI, Jegelka S. Representation learning on graphs with jumping knowledge networks. In: International conference on machine learning (ICML). 2018. p. 5453–62.
117. Yanardag P, Vishwanathan S. Deep graph kernels. In: International conference on knowledge discovery and data mining (SIGKDD). 2015. p. 1365–74.
118. Yerxa T, Anderson A, Weiss E. The hyperdimensional stack machine. In: Cognitive computing. 2018. p. 1–2.
119. Zhan Y, Shen D. Increasing the efficiency of support vector machine by simplifying the shape of separation hypersurface. In: International conference on computational and information science. Springer; 2004. p. 732–8.
120. Zou Z, Kim Y, Imani F, Alimohamadi H, Cammarota R, Imani M. Scalable edge-based hyperdimensional learning system with brain-like neural adaptation. In: International conference for high performance computing, storage and analysis (SC): networking. 2021. p. 1–15.

Publisher's Note