PeerJ Computer Science

Designing vector-symbolic architectures for biomedical applications: ten tips and common pitfalls

Fabio Cumbo[1], Davide Chicco[2,3], Sercan Aygun[4] and Daniel Blankenberg[1,5]

[1] Center for Computational Life Sciences, Cleveland Clinic Research, Cleveland Clinic Foundation, Cleveland, Ohio, United States
[2] Department of Informatics, Systems and Communication, University of Milan—Bicocca, Milan, Italy
[3] Institute of Health Policy Management and Evaluation, University of Toronto, Toronto, Ontario, Canada
[4] School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA, United States
[5] Department of Molecular Medicine, Cleveland Clinic Lerner College of Medicine, Case Western Reserve University, Cleveland, Ohio, United States

ABSTRACT

Vector-symbolic architectures (VSAs) provide a powerful, brain-inspired framework for representing and manipulating complex data across the biomedical sciences. By mapping heterogeneous information, from genomic sequences and molecular structures to clinical records and medical images, into a unified high-dimensional vector space, VSAs enable robust reasoning, classification, and data fusion. Despite their potential, the practical design and implementation of an effective VSA can be a significant hurdle, as optimal choices depend heavily on the specific scientific application. This article bridges the gap between theory and practice by presenting ten tips for designing VSAs tailored to key challenges in the biomedical sciences. We provide concrete, actionable guidance on topics such as encoding sequential data in genomics, creating holistic patient vectors from electronic health records, and integrating VSAs with deep learning models for richer image analysis. Following these tips will empower researchers to avoid common pitfalls, streamline their development process, and effectively harness the unique capabilities of VSAs to unlock new insights from their data.

Subjects Bioinformatics, Computer Architecture, Data Mining and Machine Learning
Keywords Hyperdimensional computing, Vector-symbolic architectures, Biomedical sciences, Tips and pitfalls, Biomedical data, Bioinformatics, Cheminformatics, Medical informatics

Submitted 9 October 2025
Accepted 16 January 2026
Published 10 March 2026

Corresponding authors
Fabio Cumbo, cumbof@ccf.org
Daniel Blankenberg, blanked2@ccf.org

Academic editor
Nicole Nogoy

Additional Information and Declarations can be found on page 28

DOI **10.7717/peerj-cs.3682**

INTRODUCTION

Consider representing individuals using only three attributes: *height*, *weight*, and *age*. While useful, this system quickly fails. Two very different people might have similar values, and complex traits like personality or relationships are impossible to capture. Now, imagine you could use 10 thousand different attributes. In this vast feature space, every individual would have a unique, rich description, and we could represent relationships through mathematical combinations of these descriptions. This is the core idea behind Hyperdimensional Computing (HDC) (*Kanerva, 2009*, *2022*).

Also known as vector-symbolic architectures (VSAs), HDC is a brain-inspired computational framework that represents and processes information in a fundamentally different way from traditional computing. HDC operates on high-dimensional vectors, or hypervectors, as long list of numbers (often 1,000 to 10,000+). These vectors are often implemented with bipolar scalars (+1, −1) distributed through the 1-dimensional structure. In this paradigm, a hypervector is not just a point in space. It is a holistic and distributed representation of a concept.

The power of HDC emerges from a small yet capable set of mathematical operations used to combine symbolic vectors; collectively, they are known as the Multiply-Add-Permute (MAP) model. These simple yet powerful operations enable holistic association and combination of information on vectors. For example, *multiply* (*binding*) can associate a patient-ID vector with a diagnosis code (or biomarker) vector to form a "patient-diagnosis" record. *Add* (*bundling*) can aggregate multiple symptoms/biomarkers into a single composite representation, and *permutation* (*rotation*/*shifting*) can encode the temporal order. More intuitively, binding is used to build an interaction and yields a vector that is distinct from its inputs; bundling combines records into a representation that remains similar to its constituent vectors; and permutation creates a variant of a record, often used to encode order (and, in some cases, to improve orthogonality/near-orthonormality in representations). While this article focuses on the MAP model, often implemented with bipolar or binary vectors, it is important to note that VSA encompasses a diverse family of architectures. Other frameworks include Holographic Reduced Representations (HRR), which use real-valued vectors (*e.g.*, binding *via* circular convolution, superposition *via* vector addition (*Plate, 1995*)), and Binary Spatter Codes (BSC), which operate on dense binary vectors (*e.g.*, binding *via* XOR, superposition *via* majority vote/thresholded sum (*Kanerva, 2000*)). These models differ primarily in their vector spaces and the specific mathematical realizations of their operations. For a comprehensive survey and mathematical comparison of these different VSA frameworks, we refer the reader to the extensive reviews by *Kleyko et al. (2022*, *2023)*, the systematic taxonomy and cross-VSA comparison by *Schlegel, Neubert & Protzel (2021)*, and hardware-oriented perspective on dense binary HDC/VSA operations by *Schmuck, Benini & Rahimi (2019)*.

In the MAP model used in this work, the operations are defined as:

(1) Binding (element-wise multiplication, $\otimes$): links two concepts represented in the vectors together. For example, binding the vector representation of the concept *serum level* with the vector representation of the concept *high* creates a new, distinct hypervector representing the composite idea *high serum level*;

(2) Bundling (element-wise addition, $\oplus$): combines multiple concepts in vector format. One could bundle the vectors for *high serum glucose*, *high blood pressure*, and *patient ID: 123* to create a single vector representing a patient's state. This operation highlights a key property of HDC: the ability to easily integrate heterogeneous information, numerical values, categorical states, or identifiers, into one coherent data structure;

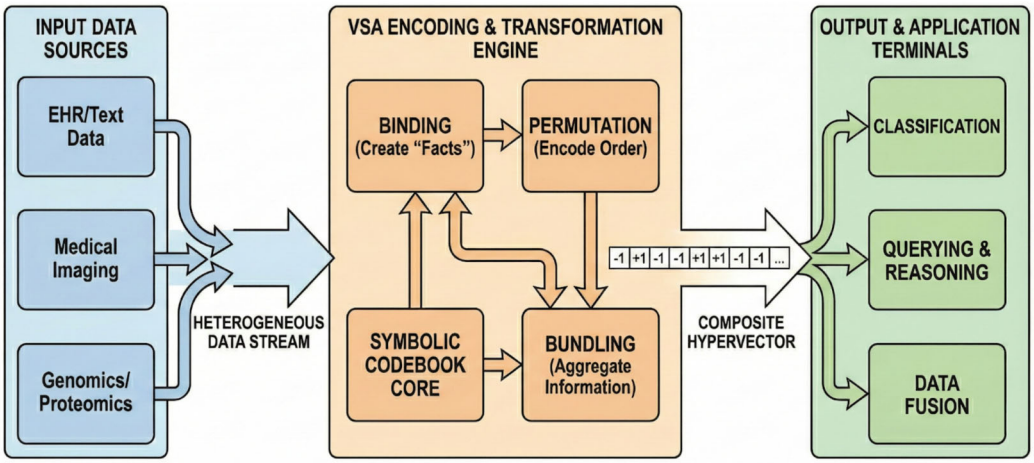

Figure 1 VSA workflow. Diverse biomedical data types (input) are first converted into symbolic hypervectors using a codebook and core operations like binding (for facts) and permutation (for sequences). This information is aggregated *via* bundling into a single, composite hypervector (output). This unified representation can then be used for various downstream tasks like classification, data fusion, and interpretable querying. Full-size DOI: 10.7717/peerj-cs.3682/fig-1

(3) Permutation (rotation, ρ): encodes order of sequences, which is critical for representing sequential data like DNA strands or time-series events in a patient's record. In this operation, each hypervector is shifted or rotated according to its position in the sequence, ensuring that the same concept appearing at different positions is represented uniquely. This positional encoding preserves order while keeping the representations nearly orthogonal (*i.e.*, their cosine similarity is approximately zero), thereby allowing clear distinction between sequences and their components.

This approach yields remarkable properties. Because information is distributed across thousands of components, HDC models are inherently robust to noise and errors (flipping a few values in a hypervector does not meaningfully change the concept it represents; much like how the brain can function despite the loss of individual neurons) (*Zhang et al., 2021*; *Zhang, Juretus & Jiao, 2025*). Furthermore, these models are computationally efficient and excel at one-shot or few-shot learning, enabling rapid model training from very little data (*Burrello et al., 2020*, *2018*; *Rahimi et al., 2017*; *Nair & Purushothaman, 2019*).

Given its unique strengths, HDC is poised to address some of the most pressing challenges in the biomedical sciences (*Cumbo & Chicco, 2025*; *Stock et al., 2024*). To help researchers harness its potential, this article offers *ten practical tips* for designing and implementing these powerful architectures in diverse biomedical research settings. Rather than providing generic advice, the tips are structured around specific applications, offering tailored guidance for distinct problems, from analyzing genomic sequences and processing clinical records to interpreting medical images. This application-driven approach, illustrated schematically in Fig. 1, provides a clear roadmap for using HDC to solve real-world challenges in computational biology and medicine. Finally, to make this effort

truly hands-on, each tip is accompanied, where applicable, by practical examples using the *hdlib* Python library (*Cumbo, Weitschek & Blankenberg, 2023*). For self-contained concepts, we provide ready-to-use code, while for more complex scenarios involving technologies outside the scope of this article, we use structured pseudocode to illustrate the core logic.

The rationale for this work stems from a growing gap between the theoretical potential of VSAs and their practical application by domain experts. While VSAs offer compelling solutions for handling the noisy, heterogeneous, and high-dimensional data common in the biomedical sciences, a clear, hands-on overview for implementation has been lacking. This manuscript aims to bridge that gap. The intended audience is therefore twofold: (i) biomedical researchers and data scientists who may be unfamiliar with VSAs but are seeking robust, interpretable methods for their data, and (ii) computational biologists already familiar with the theory who would benefit from a practical, application-driven resource with reproducible code. By providing this resource, we aim to lower the barrier to entry and facilitate the broader adoption of these powerful techniques.

SCOPE AND LITERATURE FOUNDATION

To ensure a comprehensive and practical foundation for the tips presented, we conducted a targeted review of contemporary literature. The search was performed across multiple scientific databases, primarily PubMed, Google Scholar, and the arXiv and bioRxiv preprint servers, to capture both peer-reviewed publications and recent computational advancements.

Our search strategy employed a combination of keywords. The core search terms included: “Hyperdimensional Computing”, “Vector-Symbolic Architectures”, and “VSA”. These were combined with application-specific terms relevant to the biomedical sciences, such as: “genomics”, “medical imaging”, “electronic health record”, “biosignal”, “EEG”, and “knowledge graph”.

Inclusion criteria for sources were: (i) direct application of a VSA/HDC model to a problem in biology or medicine, (ii) a clear description of the encoding methodology, and (iii) publication between 2018 and the present, to focus on the latest techniques. Exclusion criteria were: (i) purely theoretical articles with no clear application and (ii) articles focused solely on hardware implementations.

Inclusion criteria for sources were based on the following points:

- Application domain: the primary biomedical problem being addressed (*e.g.*, sequence analysis, image classification, knowledge inference);
- HDC methodology: the specific VSA operations used (binding, bundling, permutation), and the overall encoding strategy for the specific data type;
- Stated challenges and solutions: any discussion of practical hurdles, such as handling missing data, scalability, or model interpretation, and the solution proposed;
- Open science practices: the availability of source code and data, which informed our final tip on reproducibility.

The extracted information was not used to perform a systematic review of trends as indeed already available in literature (*Cumbo & Chicco, 2025*; *Stock et al., 2024*), but rather to serve as the evidence base for our ten tips. Each tip represents a distillation of successful and recurring patterns observed across multiple studies within a specific application domain. The pitfalls section for each tip was informed by the common challenges and limitations discussed in this body of literature. This synthesis process allows our manuscript to act as a bridge, translating the findings of diverse research articles into a single, accessible, and actionable framework for practitioners.

TIP 1: FOR ELECTRONIC HEALTH RECORDS—USE BUNDLING TO CREATE HOLISTIC PATIENT VECTORS

Electronic Health Records (EHRs) are a goldmine of information, but they are notoriously complex and messy (*Sæthre et al., 2025*). A single patient's record is a patchwork of different data types: structured codes for diagnoses and billing, numerical values for lab results, text from clinical notes, and lists of prescribed medications. For unstructured sources like clinical notes, Natural Language Processing techniques can first be employed to extract key symbolic entities that are then encoded using VSA (*Tavabi et al., 2024*; *Sheikhalishahi et al., 2019*; *Kuo et al., 2016*). Crucially, this data is almost always incomplete (*Wells et al., 2013*; *Jones & Farnell, 2007*). One patient might have a full panel of recent lab work, while another might have none. Traditional machine learning models often struggle with this kind of sparse, heterogeneous data, typically requiring complex preprocessing and imputation to fill in the missing values (*Joel, Doorsamy & Paul, 2022*; *Emmanuel et al., 2021*; *Rizvi et al., 2023*).

HDC offers an elegant solution through the bundling operation, which is almost always a simple vector addition. The core idea is to create a single, holistic hypervector for each patient that acts like a summary of all their known information. This patient vector is not sensitive to what is not in the summary. It just represents the sum of what is. This approach is incredibly robust to missing data and naturally integrates different information types into one unified representation (*Bai et al., 2018*).

The HDC pipeline is schematically represented in Fig. 2 and deeply discussed below.

How to create a patient vector

First, you need a dictionary that maps every elementary piece of information to its own unique and random 1-dimensional vector in 10,000 size. This constitutes the codebook. Your atomic concepts would include:

- Keys for data types: *diagnosis*, *medication*, *lab_test*, *value*;
- Specific diagnose: *diabetes_type_2*, *hypertension*;
- Specific medications: *metformin*, *lisinopril*;
- Specific lab tests: *hba1c*, *creatinine*;
- Discretized lab values: *high*, *normal*, *low*.

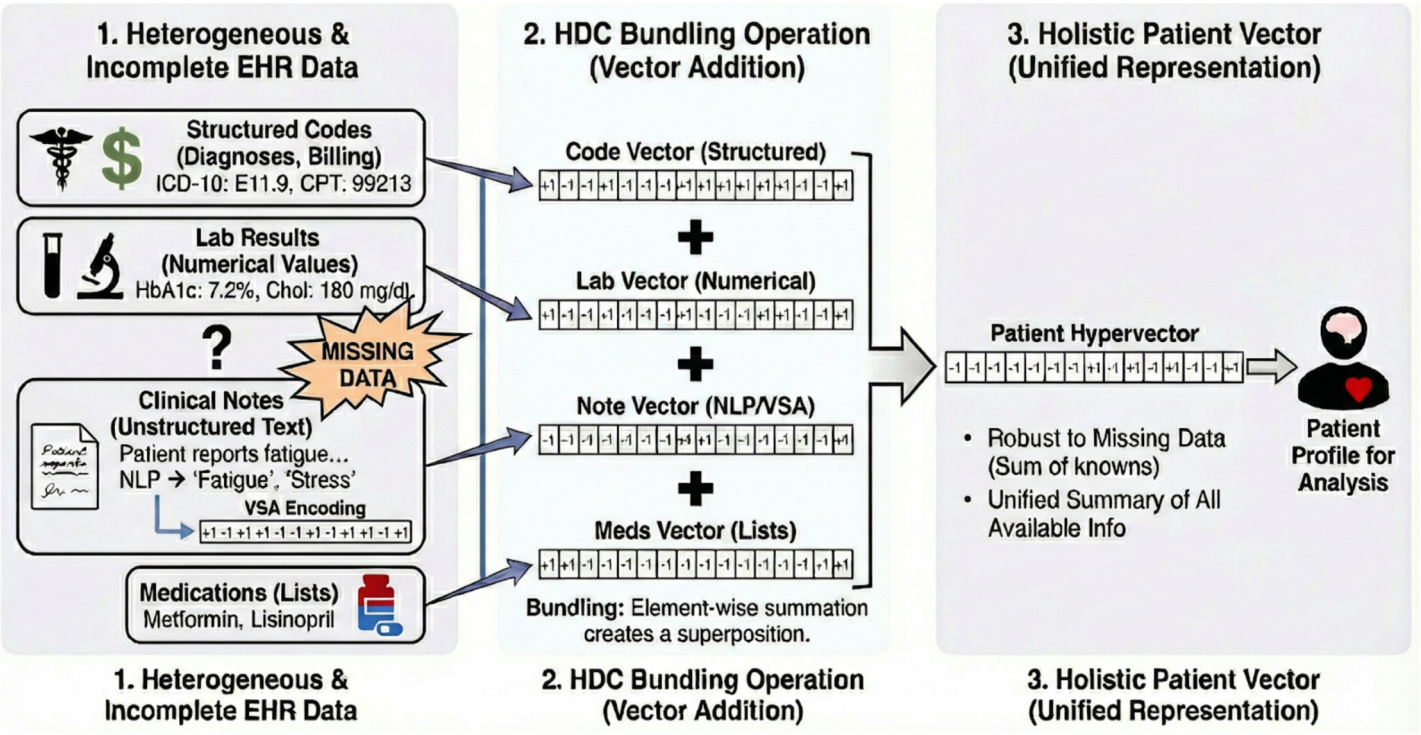

Figure 2 Tip 1 workflow. Unified representation of the aggregation of heterogeneous data sources, ranging from structured diagnostic codes and numerical lab results to unstructured clinical notes and medication lists. The central panel demonstrates the bundling operation (element-wise sum), where individual feature vectors are superimposed. This also highlights the architecture's inherent robustness to sparse data: despite the "Clinical Notes" being unavailable, the model successfully generates a unified Patient Vector by summing up only the vectors of the available data. This results in a holistic representation of known facts that allows for downstream analysis without requiring any imputation steps at all.

Full-size DOI: 10.7717/peerj-cs.3682/fig-2

Using *hdlib*, we can create this with a simple list of concepts, as shown in Supplemental Code Snippet 1.

For each piece of information in a patient' record, a vector is created by binding the key to its value. Binding combines them to create a new, distinct product vector that is dissimilar to its original components:

- A diagnosis of hypertension becomes: $patient\_diagnosis = diagnosis \otimes hypertension$
- A prescription for metformin becomes: $patient\_prescription = medication \otimes metformin$
- A HbA1c result of 8.1% (which we categorize as high) becomes: $patient\_hba1c = (lab\_test \otimes hba1c) \oplus (value \otimes high)$

Notice that we bundled two bound pairs to represent a more complex fact (the *test* and its *result*).

This translates to the code in Supplemental Code Snippet 2 using *hdlib*.

Next, we bundle all of a patient's fact vectors together to create their final, comprehensive representation.

Let us consider Patient A: high hypertension, takes metformin, has high HbA1c. Their patient vector would be calculated as:

$$patient\_A = (diagnosis \otimes hypertension) \oplus (medication \otimes metformin) \oplus ((lab\_test \otimes hba1c) \oplus (value \otimes high)).$$

Now, consider Patient B, for whom we only have a diagnosis: has hypertension. Their vector is simply:

$$patient\_B = diagnosis \otimes hyertension.$$

This can be simply achieved using *hdlib* as shown in Supplemental Code Snippet 3.

Why this works

- Robustness to missing data: Patient A's vector was created without knowing their creatinine level. Patient B's vector was created with only one piece of information. The model does not break or require modification. It works with whatever data is available (*Zhang, Juretus & Jiao, 2025*);
- Natural data fusion: the patient vector seamlessly combines different types (diagnoses, meds, labs) into a single mathematical object. You do not need separate models for each data type (*Chang et al., 2019*; *Zhao et al., 2024*);
- Similarity-based reasoning: the key advantage appears when comparing patient vectors. The vector for Patient B is a component of the vector for Patient A. Mathematically, this means that their two vectors have a high degree of similarity. Thus, in general, you can find patients with similar clinical profiles by simply looking for vectors having high cosine similarity. Note that *hdlib* provides a *dist* function as Vector's instance method to perform the cosine distance between the specific instance vector and a different Vector object, defined as 1 *minus* the cosine similarity. Following the previous example shown in Supplemental Code Snippet 3, we can compute the cosine distance between *patient_A* and *patient_B* vectors using hdlib as: *patient_A.dist(patient_B, method="cosine")* (*Fanizzi & d'Amato, 2025*).

There are however a couple of pitfalls that you should be careful of while creating holistic patient vectors:

Pitfall 1: unbalanced feature influence—if one type of data has far more entries per patient than another, simply summing them can create a patient vector that is dominated by the more frequent data type. For instance, if a patient has 50 lab results and only 1 diagnosis, the resulting patient vector will be overwhelmingly dominated by the lab data, rendering the diagnosis invisible to similarity searches.

How to avoid pitfall 1: normalize by feature type—before bundling, sub-vectors can be created for each category. Normalize these sub-vectors so they have a magnitude of 1, and then bundle the normalized sub-vectors. This ensures each data modality contributes equally to the final patient vector, regardless of the number of entries.

Pitfall 2: over-discretization of numerical values—when converting a continuous value like blood pressure into a symbolic vector (*e.g.*, *high*, *normal*, *low*), choosing poor

thresholds can lose critical information. A patient with a value just barely in the normal range might be treated the same as one with a perfect value.

How to avoid pitfall 2: use more granular categories or continuous encoding—instead of just *high*, consider multiple levels like *normal*, *elevated*, *hypertension_stage_1*. For strictly ordinal or continuous data, use float-to-vector mappings. These schemes map numerical values to hypervectors such that numerical proximity is preserved as vector similarity (*e.g.*, the vector for 0.5 is highly similar to 0.51, but orthogonal to 0.9), preventing the loss of information inherent in simple discretization.

TIP 2: FOR GENOMICS AND PROTEOMICS—LEVERAGE PERMUTATIONS TO ENCODE SEQUENCES

In genomics and proteomics, the order of elements defines function. For example, the DNA sequence *ACG* (encoding the amino acid threonine) is fundamentally different from *GCA* (alanine), even though they contain the exact same nucleotides. A model that cannot distinguish *ACG* from *GCA* is useless for most bioinformatics tasks.

Encoding positions

HDC solves this problem using the positional vector binding or the permutation operator ρ. Permutation systematically shuffles the elements of a hypervector in a deterministic way. For example, a single permutation ρ^1 might rotate the vector's 10,000 elements one position to the left. Applying it again ρ^2 rotates it one more position. Each amount of shift represents a specific location in a sequence, thereby preserving positional information during encoding. Crucially, each permutation creates a new hypervector that is dissimilar (quasi-orthogonal) to the original and to other permutations of itself. This property allows us to encode an element's relative position within a sequence (*Imani et al., 2018*; *Zou et al., 2022*; *Xu et al., 2024*). On the other hand, positional binding with separate vectors assigned to specific positions is another approach. In this method, each position (symbolic information) is represented by a unique hypervector. Let us explore the two primary ways to do this:

Method 1: binding with a position vector—This method is like putting labeled tags on items. You create a generic *position* vector and permute it to create a unique tag for each position (*position_0*, *position_1*, *position_2*). You then bind each sequence element to its corresponding position tag. This method is best suitable for answering the question "*what is at a specific position?*". This can be easily achieved using *hdlib* as in Supplemental Code Snippet 4.

Method 2: directly permuting element vectors—This more direct method alters the element's vector itself based on its position. The first element's vector is permuted 0 times, the second is permuted once, the third twice, and so on. This is best for answering the question "*where is a specific element?*". A Python snippet using *hdlib* is reported in Supplemental Code Snippet 5.

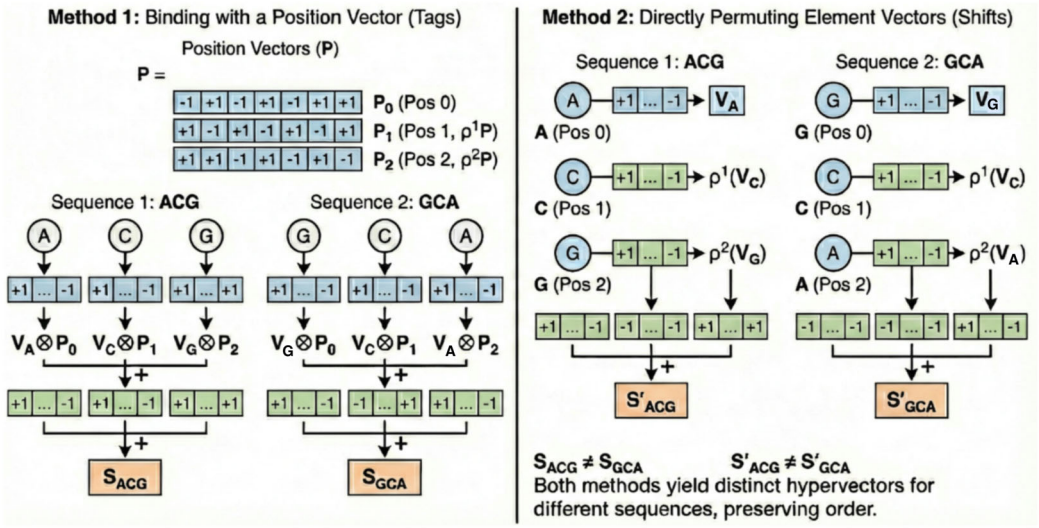

Figure 3 Tip 2 workflow. Contrast of two primary strategies for preserving sequential information, using the DNA sequences ACG and GCA as examples. Method 1 (left panel) utilizes positional tags, where a generic position vector P is permuted to create unique vectors for each index (P0, P1, P2). These tags are then bound to the corresponding nucleotide vectors before bundling. Method 2 (right panel) employs direct permutation, where the nucleotide vector itself is rotated according to its specific location in the sequence (*e.g.*, the vector representation of the nucleotide G is shifted twice for position 2). Both approaches aggregate the components into a single composite hypervector *via* bundling. Both methods ensure that the vector for ACG is mathematically distinct from GCA, allowing the model to distinguish sequences that contain the same elements but in different order.

Full-size DOI: 10.7717/peerj-cs.3682/fig-3

Returning to the example of *ACG* and *GCA*; when encoding *ACG* by permuting approach, the vector for *A* is left unshifted for the first position, while *C* and *G* are permuted by one and two positions, respectively, before binding and bundling into a single representation. For *GCA*, the vector for *G* remains unshifted, while *C* and *A* are permuted by one and two positions, respectively. As a result, even though both sequences contain the same symbols, their encodings yield distinct hypervectors, preserving order information.

Both Method 1 and Method 2 are shown in Fig. 3 detailing the ACG/GCA examples discussed above.

Furthermore, this encoding can be extended to capture higher-level biological knowledge, such as the codon wobble base phenomenon. By bundling the sequence vector of a codon with the vector for the amino acid it encodes, we can create biologically-aware representations. In this model, vectors for synonymous codons (*e.g.*, *GCA* and *GCC*) become highly similar, as they share the same core amino acid vector, while vectors for non-synonymous codons remain dissimilar. This demonstrates how VSAs can move beyond literal sequence representation to encode abstract functional relationships.

Which method is preferable in practice?

For general sequence comparison, both methods are equally effective. They both successfully create distinct, dissimilar vectors for sequences with different orderings.

The choice of method depends on the type of queries the model is expected to support:

- Choose method 1 if your primary task is querying by position. For instance, if one frequently needs to ask questions like "*what nucleotide is at position 257?*", method 1 excels because one can directly query the full sequence vector with the *position_257* vector to retrieve the answer;
- Choose method 2 if the goal is general sequence comparison, alignment, or if one needs to ask questions like "*at what positions does Adenine ("A") appear?*". This approach is more elegant and memory efficient and requires a smaller codebook.

It should also be noted that repeatedly permuting a hypervector to represent many positions may eventually be limited by the vector size. When symbolic position information spans many locations, Method 1 can be a better option, since randomly generated position vectors are guaranteed to be quasi-orthogonal, avoiding the potential for cyclical collisions that can occur with repeated permutations.

Pitfall: boundary crosstalk—when a very long sequence is encoded, the permutations can become so extensive that the vector for position 1,000 might, by chance, become similar again to the vector for position 10. This is called crosstalk and can corrupt the positional information.

How to avoid the pitfall: use hierarchical encoding—recursive bundling is the standard method for representing hierarchical structures. Instead of encoding a massive sequence (like a whole genome) into a single monolithic vector, break it down naturally. Create a vector for each gene, then bundle the gene vectors for the genome. This chunking contains the permutations within smaller, more manageable contexts (*e.g.*, position is relative to the start of the gene, not the chromosome), effectively preventing cross-talk.

TIP 3: FOR MEDICAL IMAGING—COMBINE VECTOR-SYMBOLIC ARCHITECTURES WITH CONVOLUTIONAL NEURAL NETWORKS FOR EXPLICABLE ARTIFICIAL INTELLIGENCE

Convolutional Neural Networks (CNNs) achieve state-of-the-art performance in medical image classification (*Salehi et al., 2023*; *Yao et al., 2024*; *Li et al., 2014*; *Xie et al., 2021*; *Yuan, Zhang & Fang, 2023*). However, their "black box" nature poses a challenge for clinical adoption, as they often lack the ability to provide symbolic explanations for their decisions. A CNN might correctly classify a histology slide as malignant, but it cannot easily explain why. It cannot answer a doctor's follow-up questions like "*which region of the image led to this conclusion?*" or "*what specific cellular structures did you find?*". This lack of interpretability is a significant barrier to trust and adoption in clinical settings.

Building a queryable scene description

Instead of replacing CNNs, we can augment them with VSAs to create a hybrid, explainable AI model with a more interpretable pipeline. The key idea is to use CNN as a feature extractor on localized image patches for low-level patterns (textures, edges, shapes) in an image. The VSA layer then serves as a codebook and composition mechanism for

assigning symbolic hypervectors to recurring feature patterns and spatial locations, enabling a compositional "scene description" of the image. It takes previous findings, gives them symbolic names (*e.g.*, "spindle-shaped cells", "high nuclear density"), notes their locations, and assembles everything into a structured, queryable report.

This hybrid model combines the pattern-recognition capabilities of deep learning with the reasoning and interpretability of symbolic architectures (*Hassan et al., 2022*; *Luczak, Slot & Kucharski, 2022*; *Lee et al., 2023*).

Because a full implementation is highly complex, we will use a mix of descriptive steps and pseudocode to illustrate the logic:

Step 1: divide the image and extract features with a CNN—First, we break the input image (*e.g.*, a large histology slide) into a grid of smaller patches (*e.g.*, 16 × 16 grid). We then feed each patch into a pre-trained CNN. Instead of using the CNN's final classification, we intercept the output from an earlier layer, a rich feature vector that numerically describes the contents of that path. As anticipated earlier, since the implementation of this step could involve getting deep into many details about neural networks in general, we are going to assume that this method is implemented in a function called *cnn_feature_extractor* that, given an image patch, returns the feature vector;

Step 2: build a VSA codebook for features and locations—Next, we create a VSA codebook to hold the symbolic representations of the image's contents. This involves two sets of concepts: vectors for each spatial location in our grid, and vectors for the low-level features the CNN finds. Since we do not know what the raw numerical features from the CNN mean ahead of time, we must first discover and then name them. This is a two-step process:

- Discover feature groups: we first run a clustering algorithm (like K-Means) on all the feature vectors produced by the CNN from a training set of images. This groups the thousands of slightly different numerical feature vectors into a small number of distinct clusters. Each cluster represents a recurring visual pattern, like a certain texture or cell shape;
- Assign symbolic names: we then treat each cluster as a single, unified concept. We assign a symbolic name to each one and generate a unique, random hypervector for each of these symbolic names in our codebook.

The numerical cluster centroid (the average feature in a group) is only used temporarily to determine which symbolic name a new, unseen CNN feature belongs to. The VSA model itself works exclusively with the final symbolic hypervectors.

Step 3: create the composite image vector—Now, we loop through each patch of the image. For each patch, we get its CNN feature vector, find the closest matching symbolic feature in our VSA codebook, and bind that symbolic feature to the patch's location vector. Finally, we bundle all these bound pairs into a single hypervector that represents the entire image.

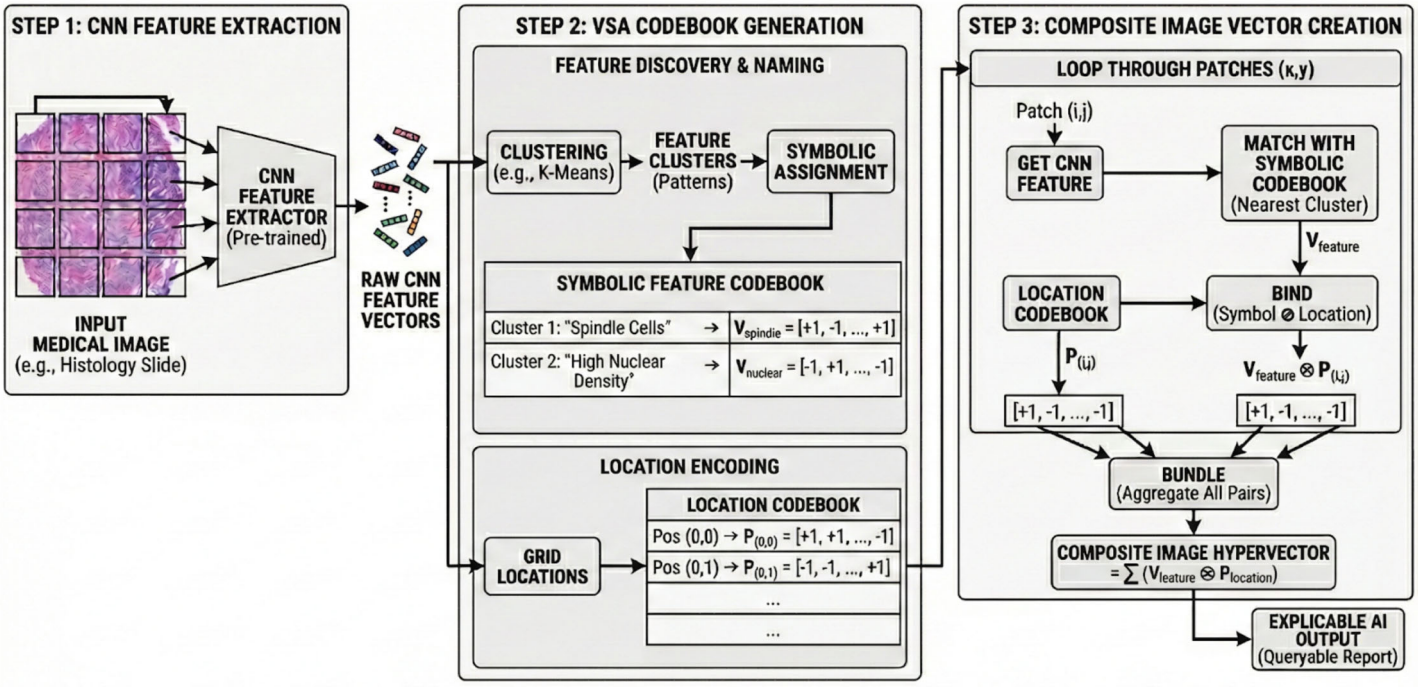

Figure 4 Tip 3 workflow. Demonstration of how to augment deep learning with VSA to create interpretable models. Step 1 involves dividing the input medical image (*e.g.*, a histology slide in this case) into patches and feeding them into a pre-trained CNN to extract raw numerical feature vectors. Step 2 focuses on translating these features into a symbolic language: raw vectors are grouped *via* clustering (*e.g.*, K-Means) to discover distinct visual patterns (labeled here as "Spindle Cells" or "High Nuclear Density") and assigned to unique hypervectors in the symbolic feature codebook. Step 3 constructs the final representation by iterating through the image patches: each patch's CNN output is matched to its corresponding symbol and bound with a specific vector from the location codebook. Finally, all bound feature-location pairs are bundled into a single composite image hypervector. This creates a holistic, symbolic representation of the image that supports explicable downstream tasks, such as querying specific regions for their contents. Full-size DOI: 10.7717/peerj-cs.3682/fig-4

The whole process is summarized in Fig. 4 below and in Supplemental Code Snippet 6.

Why this works

The *image_hv* in Supplemental Code Snippet 6 is a fully symbolic and queryable model of the image's contents. You can interrogate it using VSA operations to ask questions that a standard CNN cannot readily answer.

Query: "*What features are present in the top-right corner (position 0, 3)?*"

To answer this, we take our composite *image_hv* and unbind the location vector for *position_0_3*. In *hdlib*, this is done by binding, as shown in Supplemental Code Snippet 7.

This process turns the black box into an explainable model. A doctor can now not only see the final classification but can also probe the model's intermediate reasoning, potentially discovering novel biomarkers reflected in the learned features.

Pitfall: meaningless VSA feature codebook—if the clusters generated from the CNN's feature vectors are not distinct or meaningful, the VSA codebook will map to noisy, overlapping concepts (*e.g.*, *feature_7* and *feature_12* represent very similar patterns). The symbolic model will then be built on a faulty foundation.

How to avoid the pitfall: tune the CNN and clustering—ensure the CNN is well-trained for feature extraction. Experiment with different clustering algorithms and a varying number of clusters. Manually inspect the image patches corresponding to each cluster to see if they present visually coherent patterns. The goal is to create a VSA codebook where each feature corresponds to a recognizable visual element. Finally, remember that the quality of the VSA codebook directly affects vector generation: similar items should map to hypervectors with high correlation, while dissimilar items should map to nearly orthogonal hypervectors. This alignment ensures that symbolic reasoning is grounded in meaningful representations.

TIP 4: FOR BIOSIGNAL PROCESSING (EEG/ECG)—ENCODE TIME-FREQUENCY DATA SYMBOLICALLY

Biosignals like electroencephalograms (EEG) and electrocardiograms (ECG) are continuous, noisy, and incredibly dense with information. A key challenge is capturing how the signal's characteristics change over time. For EEG, a researcher is not just interested in whether alpha waves are present, but whether they are present now, in this specific region of the brain, and at what intensity. Traditional methods often struggle to represent these dynamic, multi-faceted states in a unified way, making it difficult to classify complex cognitive states or detect transient events like a seizure onset (*Alotaiby et al., 2014*; *Ein Shoka et al., 2023*).

A symbolic snapshot of the signal's state

The VSA approach is capable of transforming the raw, numerical signal into a series of symbolic snapshots. Instead of working with the waveform directly, we first use standard signal processing techniques (like the Short-Term Fourier Transform) to analyze a short window of the signal (*Rizal, Priharti & Hadiyoso, 2021*). This tells us which frequency bands (*e.g.*, Delta, Theta, Alpha, Beta) are powerful at that time stamp. We then encode this frequency information into a single hypervector that represents the signal's state for that specific window.

By stringing these snapshot vectors together, we can create a rich, symbolic representation of the signal's evolution over time, perfect for classification and event detection (*Burrello et al., 2020*; *Pale, Teijeiro & Atienza, 2021*; *Ge & Parhi, 2022*; *Schindler & Rahimi, 2021*; *Asgarinejad, Thomas & Rosing, 2020*; *Du et al., 2024*).

Here, we are going to use a simplified EEG example, but the same logic applies to ECG, EMG, or other time-series data:

Step 1: pre-process the signal into time-frequency windows—this step does not involve VSAs. You would use a standard library like *scipy* (*Virtanen et al., 2020*) or *mne* (*Gramfort et al., 2013*) in Python to take your raw EEG signal and process it. The goal is to get the average power in each standard frequency band (Delta: 0.5–4 Hz, Alpha: 8–12 Hz, *etc.*) for short, overlapping time windows (*e.g.*, 1-s windows);

Step 2: build a VSA codebook for signal features—now we create our VSA dictionary. The concepts will be the names of our frequency bands and the different brain regions (channels) from which the signal is recorded;

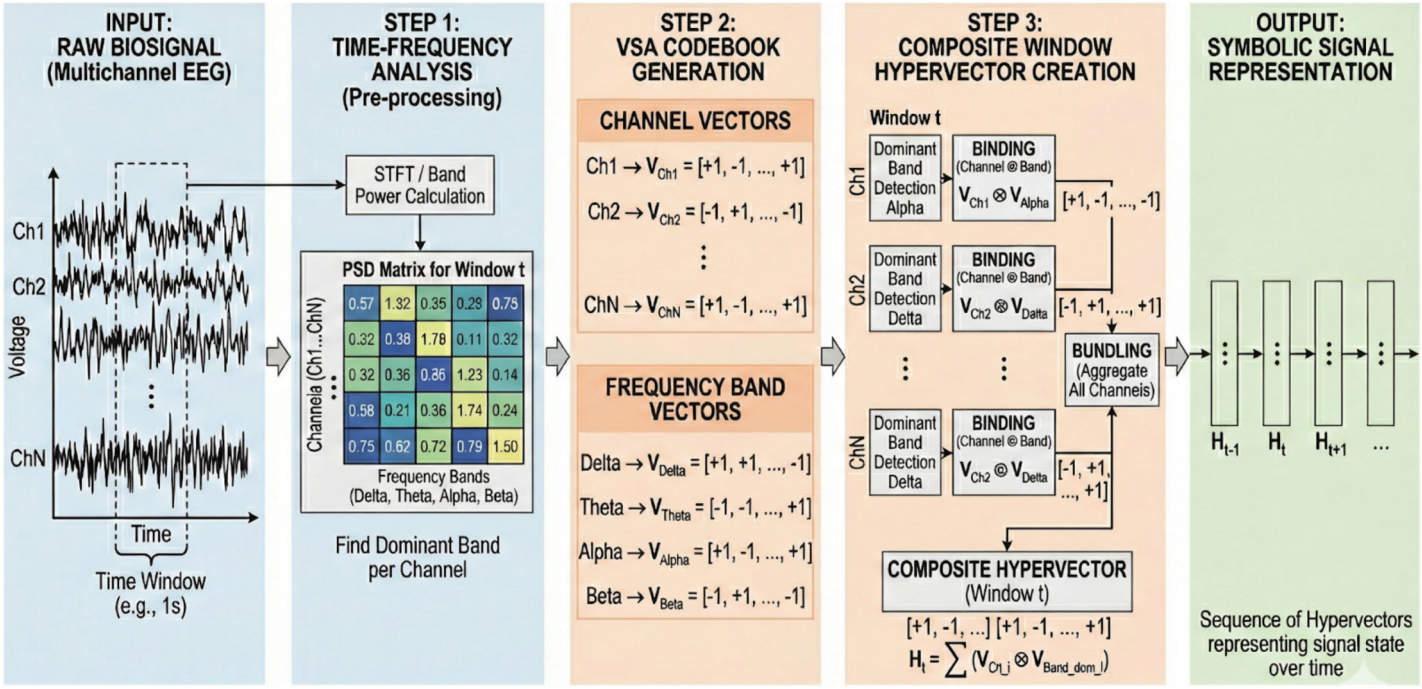

Figure 5 Tip 4 workflow. Illustration of the conversion of continuous, multichannel biosignals into a sequence of symbolic state vectors. Step 1 pre-processes the raw EEG signal by segmenting it into short time windows (*e.g.*, 1 s) and performing spectral analysis to identify the dominant frequency band (Delta, Theta, Alpha, Beta) for each channel. Step 2 establishes the VSA codebook, assigning random, orthogonal hypervectors to every channel and frequency band. Step 3 constructs the composite window hypervector by binding the specific channel vector to the vector of the band detected in that channel and bundling these pairs across all channels. The output is a temporal sequence of holistic hypervectors, each acting as a symbolic snapshot of the brain's global state at a specific moment. Full-size DOI: 10.7717/peerj-cs.3682/fig-5

Step 3: create a composite hypervector for each time window—for each window of time, we create a single hypervector that captures the state of the entire brain. We loop through each EEG channel, find its dominant frequency band for that window, and bind the channel vector to the band vector. Then, we bundle all these bound pairs together.

This process is summarized in Fig. 5 below and in Supplemental Code Snippet 8.

Why this works

After repeating this process for many time windows, you will have a set of *brain_state_hv* vectors. You can now use these for high-level tasks. For example, to build a classifier for *focused vs relaxed* states:

(1) Create a prototype: collect many *brain_state_hv* samples while a subject is focused and bundle them together to create a single *prototype_focused* vector. Do the same for the relaxed state to create *prototype_relaxed*;

(2) Classify new data: to classify a new, unseen *brain_state_hv* after following the same encoding steps of a single test sample, simply check whether it is more similar to the *focused* or *relaxed* prototype (see Supplemental Code Snippet 9 for a pseudocode using *hdlib*).

This approach transforms a complex, continuous signal processing problem into a much simpler symbolic reasoning task, making it easy to build robust and transparent classifiers for dynamic biological state.

Pitfall: wrong window size—if the time window for your analysis is too short, you will not have the frequency resolution to accurately detect low-frequency bands (like Delta waves). If it is too long, you will average out rapid, transient events and miss them entirely.

How to avoid the pitfall: match the window size to the phenomenon of interest—this is a classic signal processing trade-off. If you are looking for slow-changing cognitive states, a longer window (1–2 s) is fine. If you are trying to detect sharp, brief events like epileptic spikes, you need a much shorter window (*e.g.*, 100–250 ms). You may even need to run your analysis with multiple window sizes and bundle the resulting vectors to capture both transient and sustained events.

It is important to craft hypervectors independently, especially for band and channel symbols. Keep the random source separate for each symbol, and ideally for each concept, to preserve strong orthogonality (rather than merely near-orthogonality).

TIP 5: FOR MOLECULAR STRUCTURES—DECOMPOSE MOLECULES INTO ATOMIC FRAGMENTS

A molecule's chemical properties, like its toxicity, solubility, or ability to bind to a target protein, are determined by its 3D structure and are the arrangement of its atoms and bonds. This structure is fundamentally a graph, not a simple sequence or a fixed-size table. Representing this complex, non-linear information for machine learning is a significant challenge. How can we compare the structure of caffeine to that of aspirin in a mathematically meaningful way? Standard models often require fingerprinting methods that convert these graphs into binary vectors, but these can sometimes lose granular information (*Ma, Thapa & Jiao, 2022*; *Jones et al., 2024*; *Cumbo et al., 2025b*).

A sum of its parts

The VSA approach treats a molecule as a composition of its local atomic environments. Instead of trying to encode the entire molecular graph at once, we break it down into smaller, overlapping fragments. A fragment can be as simple as an atom and its immediate bonding partner. We then create a hypervector for each of these fragments and bundle them together to form a single hypervector that represents the entire molecule.

This method elegantly captures the principle that a molecule's function is determined by the sum of its constituent chemical groups (*e.g.*, hydroxyl groups, benzene rings). Molecules with similar functional groups will have similar hypervectors, allowing us to predict properties and behavior.

Let's use a simplified representation of a water molecule (H-O-H) and a hydrogen peroxide molecule (H-O-O-H) to illustrate the process:

Step 1: define your atomic and *bonding* codebook—first, we create a VSA codebook for the basic building blocks, *i.e.*, the different types of atoms (elements);

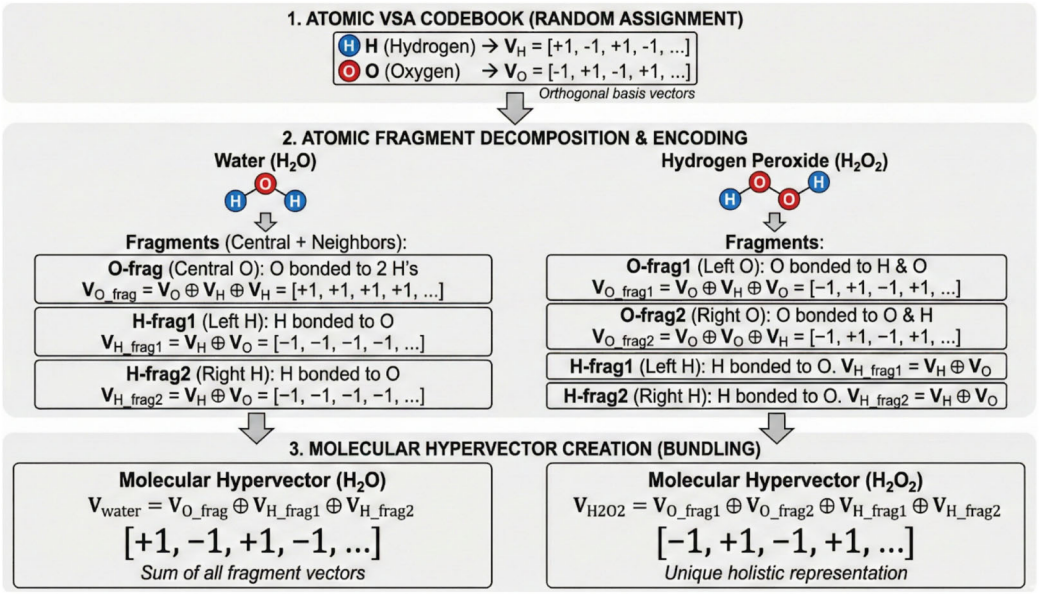

Figure 6 Tip 5 workflow. Decomposition of chemical structures into additive atomic fragments to generate holistic molecular representations. Step 1 initializes the process by assigning random, orthogonal hypervectors to atomic elements *via* the codebook. Step 2 performs atomic fragment decomposition, where each atom in the molecule is treated as a center and encoded by bundling its vector with the vectors of its immediate neighbors. The comparison between water and hydrogen peroxide highlights the context-awareness of this approach: the oxygen fragment in water (bonded to two H's) generates a different vector than the oxygen fragment in peroxide (bonded to one H and one O), capturing local structural differences. Step 3 aggregates these local environment vectors *via* bundling to produce a single molecular hypervector. This final representation acts as a superposition of all constituent substructures, allowing molecules with similar functional groups to remain mathematically similar in the vector space.
Full-size DOI: 10.7717/peerj-cs.3682/fig-6

Step 2: define and encode atomic fragments—an atomic fragment is the local environment around a central atom. We can represent it by bundling the central atom's vector with vectors representing each of the neighboring atoms. The fragments in a water molecule are defined as:

- oxygen fragment: one central oxygen atom connected to two hydrogens *via* single bonds;
- hydrogen fragment: each hydrogen is connected to one oxygen *via* a single bond.

Step 3: bundle fragments into a molecular hypervector—finally, we sum the hypervectors of all the atomic fragments in the molecule to get the final molecular representation.

Figure 6 and Supplemental Code Snippet 10 summarizes these steps.

Why this works

This compositional and context-aware approach allows the creation of highly descriptive vectors and the prediction of a molecule's properties based on the similarity of its vector to known prototypes. Imagine we have built a prototype vector for *soluble* and *insoluble* compounds by averaging the vectors of many known examples.

To predict the solubility of a new molecule, we first construct its hypervector and then see which prototype it resembles more closely.

This method effectively translates the complex graph-based problem of cheminformatics into a vector-space similarity problem allowing for fast, scalable, and surprisingly accurate predictions of molecular properties.

Pitfall: ignore stereochemistry and 3D information—the refined fragment encoding is good, but it still represents the molecule as a 2D graph. It cannot distinguish between enantiomers (left- and right-handed versions of a molecule), which can have drastically different biological effects (*e.g.*, Thalidomide).

How to avoid the pitfall: add 3D information to the codebook—enrich your fragments. In addition to *bond_to*, you can add vectors for stereochemical information like *r_config* or *s_config* at chiral centers. You can also discretize bond angles or dihedral angles and bind this information into the fragment vector to provide a richer, more 3D-aware representation. Practically, encoding left–right and numerical angle features in HDC is straightforward. For numerical angles, we can use correlation-aware codes (such as those constructed by flipping a small, fixed percentage of bits as values increase). This mapping ensures that the cosine similarity between vectors decays linearly with the difference in their values. Thus, nearby angles remain highly similar, while distant ones become orthogonal. Since angles exist in a bounded range (0–360°), building an angle codebook is simple. During molecule construction, this requires only one additional binding step to inject the tag. Multi-stage cascading of such encodings (bond type, angle/dihedral, role) remains both simple and effective. One crucial point is that symbolic roles (left, right, top, bottom, *etc.*) should remain orthogonal and be drawn from independent random sources to ensure a clear distinction.

TIP 6: FOR BIOMEDICAL KNOWLEDGE GRAPHS—USE BINDING TO REPRESENT RELATIONAL TRIPLES

The biomedical field is flooded with massive databases containing information about which genes are associated with which diseases, what proteins interact with each other, which drugs inhibit certain enzymes, and many other relational information. These networks of relationships form a knowledge graph (*Nicholson & Greene, 2020*; *Bonner et al., 2022*; *Harnoune et al., 2021*). A key challenge is how to represent this graph in a way that allows us to reason with it, infer new relationships, and answer complex questions. Traditional graph databases can be rigid and computationally expensive to query for fuzzy or incomplete matches.

Facts as vectors

VSAs provide a powerful way to encode an entire knowledge graph into a single, dense hypervector (or a collection of them). The fundamental unit of a knowledge graph is the relational triple <*subject, predicate, object*> (*e.g.*, <*caffeine, inhibits, adenosine_receptor*>).

The VSA solution is to use the binding operation to compress this entire triple into a single hypervector. This fact vector represents the complete relationship. By bundling

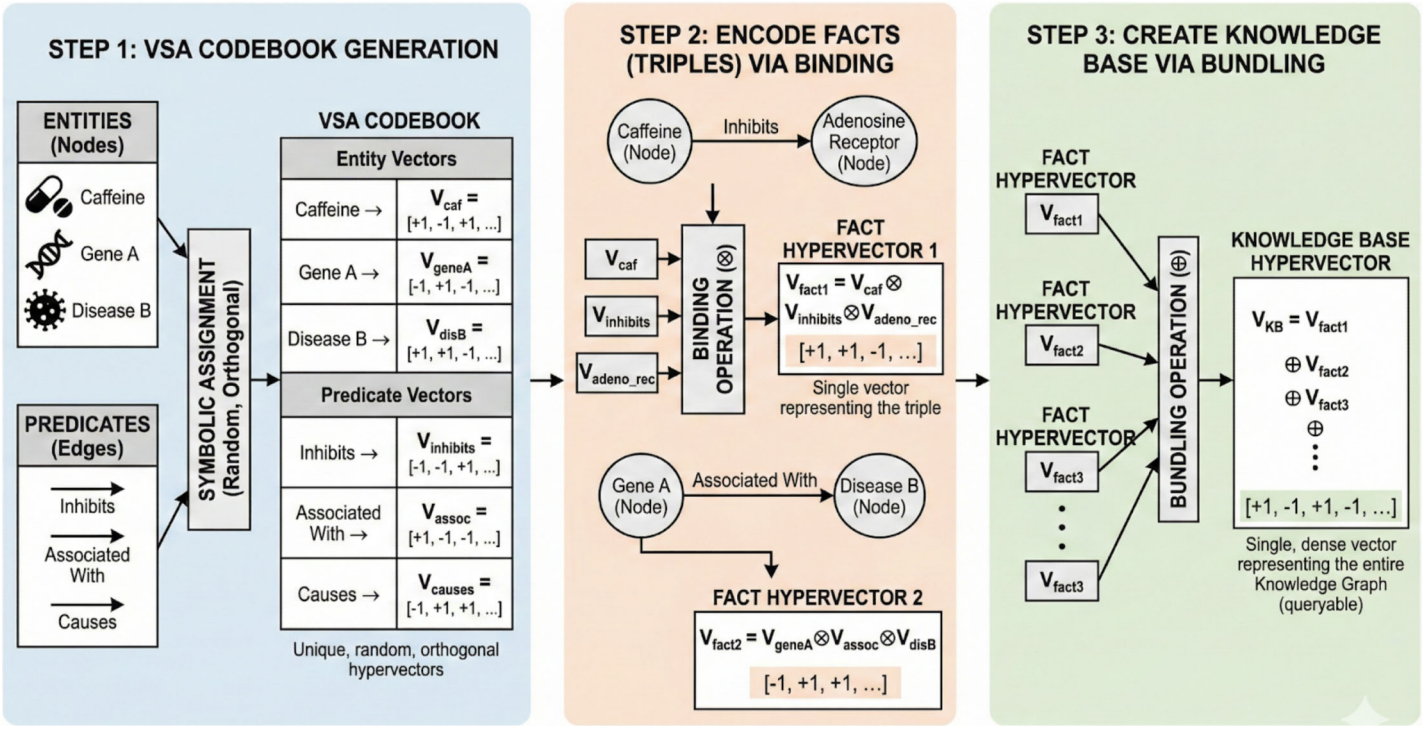

Figure 7 Tip 6 workflow. This workflow illustrates how to compress a complex network of relationships into a single queryable vector. Step 1 establishes the VSA codebook, assigning unique, random, and orthogonal hypervectors to every entity node (*e.g.*, Caffeine, Gene A) and predicate edge (*e.g.*, Inhibits, Causes). Step 2 encodes individual relational triples, such as, into distinct fact hypervectors by binding the subject, predicate, and object vectors together. Step 3 aggregates these facts into a unified knowledge base hypervector *via* the bundling operation. The resulting composite vector stores the entire graph structure in a distributed representation that supports similarity-based reasoning and querying.

Full-size DOI: 10.7717/peerj-cs.3682/fig-7

thousands of these fact vectors together, we can create a composite vector that represents a vast body of knowledge:

Step 1: building a codebook for entities and predicates—first, we create a dictionary that maps every entity (like a drug, gene, or disease) and every predicate (a relationship, like *causes*, *inhibits*, or *interacts_with*) to a unique hypervector;

Step 2: encode each fact using binding—now, for each triple in our knowledge graph, we create a fact vector by binding the subject, predicate, and object together;

Step 3: bundle facts into a knowledge base vector—to create our final knowledge base, we simply bundle all the individual fact vectors together. The single vector now contains all the information from our original graph, stored in a distributed, superimposed manner.

Figure 7 and Supplemental Code Snippet 11 summarizes the whole procedure.

Why this works

The true power of this representation lies in its ability to answer questions and infer relationships through vector arithmetic. We can query the knowledge base by providing parts of a triple and using the inverse operation to find the missing piece.

Question: "*What does caffeine inhibit?*"

To answer this question, we create a query vector by binding *caffeine* and *inhibits*, and then bind this to our knowledge base along with the query vector as shown in Supplemental Code Snippet 12.

This VSA technique allows for powerful, fuzzy queries on massive datasets. It can be used to infer novel drug-repurposing candidates, discover potential gene-disease associations, and build systems that can reason over the vast and ever-growing body of biomedical literature.

Pitfall: ambiguity of the unbind operation—when you query the knowledge base, the result vector is an approximation, not a clean answer. It is a superposition of the correct answer plus noise from all other facts in the knowledge base. If your knowledge base is very dense with similar relationships, the noise can overwhelm the signal, and the top match for your query might be incorrect.

How to avoid the pitfall: use cleanup memory—after you get your *result_hv*, do not just find the single best match in your codebook. Instead, find the top 3–5 closest matches and bundle them together. Then, use this new, cleaner vector to re-query the codebook. This process, also called iterative cleanup, helps to amplify the signal and suppress the noise, leading to more accurate query results.

TIP 7: FOR CLASSIFICATION TASKS—BUILD AND REFINE PROTOTYPE VECTORS

In many biomedical scenarios, collecting large, labeled datasets for training complex machine learning models is a difficult task. You might have detailed data for thousands of healthy individuals but only a few dozen patients with a rare disease. Training sophisticated models like deep neural networks on such imbalance or small datasets is often impractical and can lead to poor performance (*Bugnon et al., 2020*; *Pes, 2020*). The challenge is to build a reliable classifier that can learn effectively from a small number of samples (a concept known as few-shot learning) (*Cumbo, Cappelli & Weitschek, 2020*; *Cumbo et al., 2025c*, *2025a*).

Learning the average case

The VSA approach for classification is beautifully simple and effective. Instead of learning a complex decision boundary, we create a single prototype vector for each class. A prototype vector is simply the bundle of all the sample vectors belonging to that class. The learning process happens incrementally, with each sample contributing additively to the prototype hypervector. Thus, the reverse operation, unlearning, is simply subtracting that contribution.

Let us consider an hypothetical *prototype_diabetic* vector that represents the central tendency of all diabetic patients in your training set. To classify a new patient, you simply check if their vector is more similar to the diabetic prototype or the non-diabetic

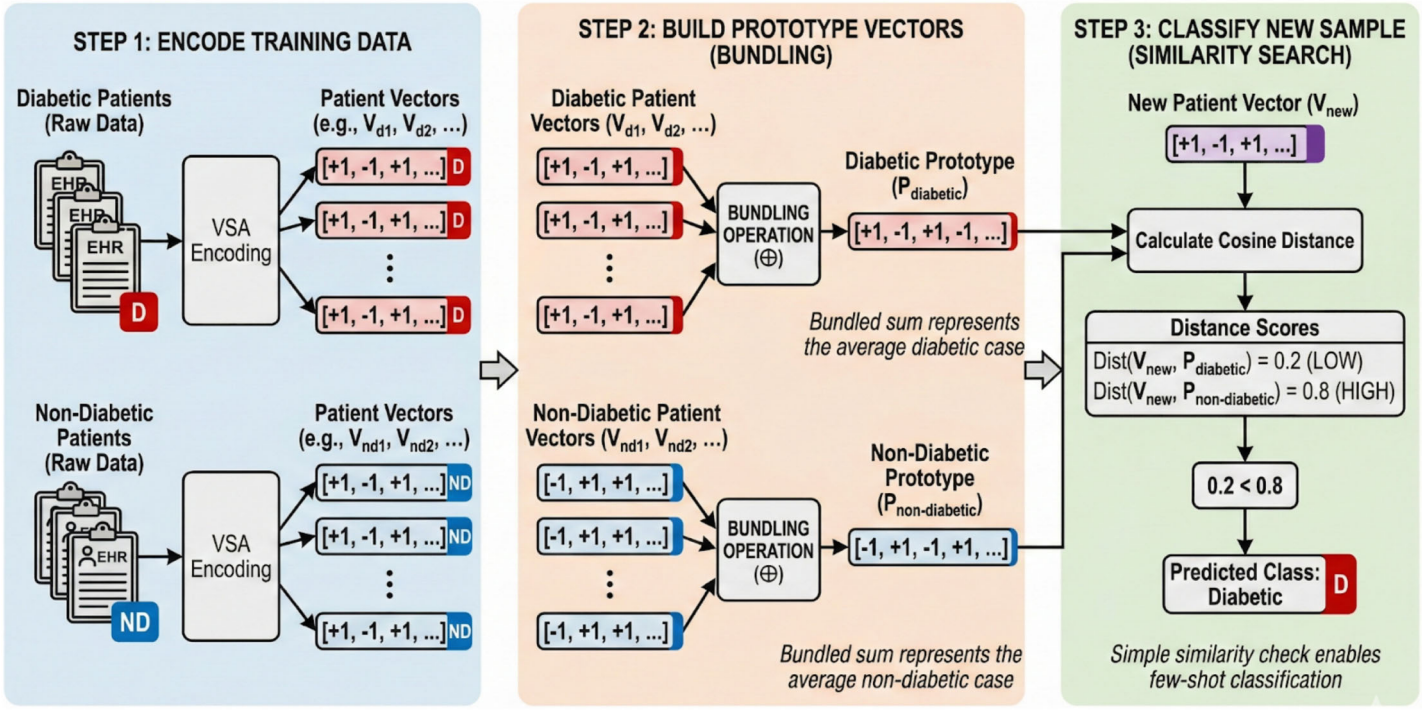

Figure 8 Tip 7 workflow. Classification approach outline. Step 1 encodes raw training data (*e.g.*, EHRs for Diabetic *vs* Non-Diabetic patients) into individual patient hypervectors. Step 2 constructs a prototype vector for each class by bundling all corresponding sample vectors. This resulting prototype represents the average or central tendency of that class. Step 3 classifies a new, unseen patient by calculating its cosine distance to each prototype. The model predicts the class associated with the lowest distance to each prototype. The model predicts the class associated with the lowest distance score (highest similarity).
Full-size DOI: 10.7717/peerj-cs.3682/fig-8

prototype. This method is transparent, computationally cheap, and works remarkably well even with very few training samples.

Let us assume we have already encoded our data into hypervectors. For this example, we will use the patient vectors created in Tip 1, where each vector represents a patient's EHR data. Our goal is to classify patients as either *diabetic* or *non_diabetic*:

Step 1: encode your training data—first, ensure every sample in your labeled training set is represented as a hypervector. We will presuppose this step is complete and we have a list of patient vectors and their corresponding labels;

Step 2: create a prototype vector for each class—now, we iterate through our training data. We will add each patient's vector to a running sum for their respective class;

Step 3: classify a new sample with similarity search—with our prototypes built, classification is incredibly straightforward. For any new, unlabeled patient vector, we calculate its cosine distance to each prototype vector. The class of the prototype with the lowest distance score is our predicted class (see Supplemental Code Snippet 13 for a code overview).

This pipeline is shown in Fig. 8 below.

Why this works

This prototype-based method is a cornerstone of applied VSA for several reasons:

- Simplicity and speed: the training phase is just one pass of addition. The prediction phase is just a few similarity calculations. This is orders of magnitude faster than training a deep neural network;
- Excellent for few-shot learning: this method creates a reasonable prototype even with just a few samples per class;
- Incremental learning/unlearning: if you get new labeled data, you do not have to retrain your model from scratch. You can simply update the existing prototype sums with the new vectors, allowing your model to learn continuously. Conversely, if data needs to be removed, the update is just a subtraction from the prototype.
- Iterative refinement for higher accuracy: while the one-pass training described above is efficient, the model's performance can be significantly improved through iterative retraining. In this context, retraining does not mean rebuilding the model from scratch. Instead, it refers to an error-correction process: we loop through the training data and, if a model misclassifies a sample, we simply subtract that sample's vector from the incorrect prototype and add it to the correct one. This iterative adjustment shifts the prototype vectors to better discriminate between classes, effectively reducing the error rate with minimal computational overhead.

Pitfall: class size imbalance dominates prototypes—if you create prototypes by summing vectors, the class with more samples will produce a prototype vector with a much larger magnitude. For example, if you have 1,000 *healthy* samples and only 50 *rare_disease* samples, the *prototype_healthy* vector will be mathematically dominant. Consequently, almost every new sample, even those with the disease, will be more similar to the stronger healthy prototype, making your classifier extremely biased toward the majority class.

How to avoid the pitfall: normalize the final prototype vectors—after you have finished summing all the vectors for each class, normalize the final prototype vectors so they all have the same magnitude. This crucial step removes the bias from class size imbalance. It ensures that a sample's classification is based purely on the pattern of the prototype, not its magnitude (the number of samples used to create it).

TIP 8: FOR DATA FUSION—MAP ALL MODALITIES INTO A COMMON VECTOR SPACE

A single patient's story is often told across multiple data types, or modalities. A clinician might have an MRI scan (an image), a radiologist's report (text), genetic markers (sequential data), and basic lab results (numerical data). Each modality provides a piece of the puzzle, and the richest insights come from combining them (*Chang et al., 2019*; *Zhao et al., 2024*; *Chicco, Cumbo & Angione, 2023*). The core challenge is that these data types are fundamentally incompatible. You cannot just add a pixel to a word or a gene to a blood

pressure reading. How can you create a single, unified representation that respects the information from all sources?

A universal language

The elegance of VSAs is that hypervectors act as a universal language. As long as all your vectors share the same dimensionality, they can be mathematically combined, regardless of their origin. The strategy for data fusion is simple:

(1) Encode each modality separately into its own hypervector using the most appropriate technique (*e.g.*, methods from Tip 1 for text/EHRs, Tip 2 for genetic sequences, Tip 3 for images);
(2) Bundle the resulting vectors together to create a single, multimodal hypervector that represents the complete picture.

This allows to create a holistic patient representation that is richer than any single data source alone. Let's imagine our goal is to create a single patient vector that combines the findings from a written radiology report and the corresponding MRI scan:

Step 1: encode the text modality—first, we process the text from the radiology report. We can use a method similar to Tip 1, creating a simple bag-of-words vector by bundling the vectors for key terms found in the report;

Step 2: encode the image modality—next, we process the MRI scan. As described in Tip 3, we would use a hybrid CNN-VSA approach to convert the image into a single hypervector. This *image_hv* symbolically represents the key features and their locations within the scan;

Step 3: bundle the modalities into a unified vector—this is the easiest and most powerful step. To create the final, multimodal representation of the patient's case, we simply bundle the *report_hv* and the *image_hv* together.

This is all summarized in Fig. 9 below and in Supplemental Code Snippet 14.

Why this works

The resulting *patient_record_hv* from Supplemental Code Snippet 14 is a single vector that contains superimposed information from both the text and the image. In our example, it contains *tumor* from both sources, *left_hemisphere* from the text, and *edema* from the image. This fused vector is a more complete and robust representation than either vector alone. When used for classification (Tip 7), this richer vector will lead to more accurate predictions because it can draw on evidence from multiple sources.

Pitfall: modality imbalance—if one modality is represented in a much more complex way than another, its vector might have a larger magnitude or simply contain more information, causing it to drown out the contribution of the other modality during bundling. For instance, a complex image vector might dominate a simple text vector derived from just two keywords.

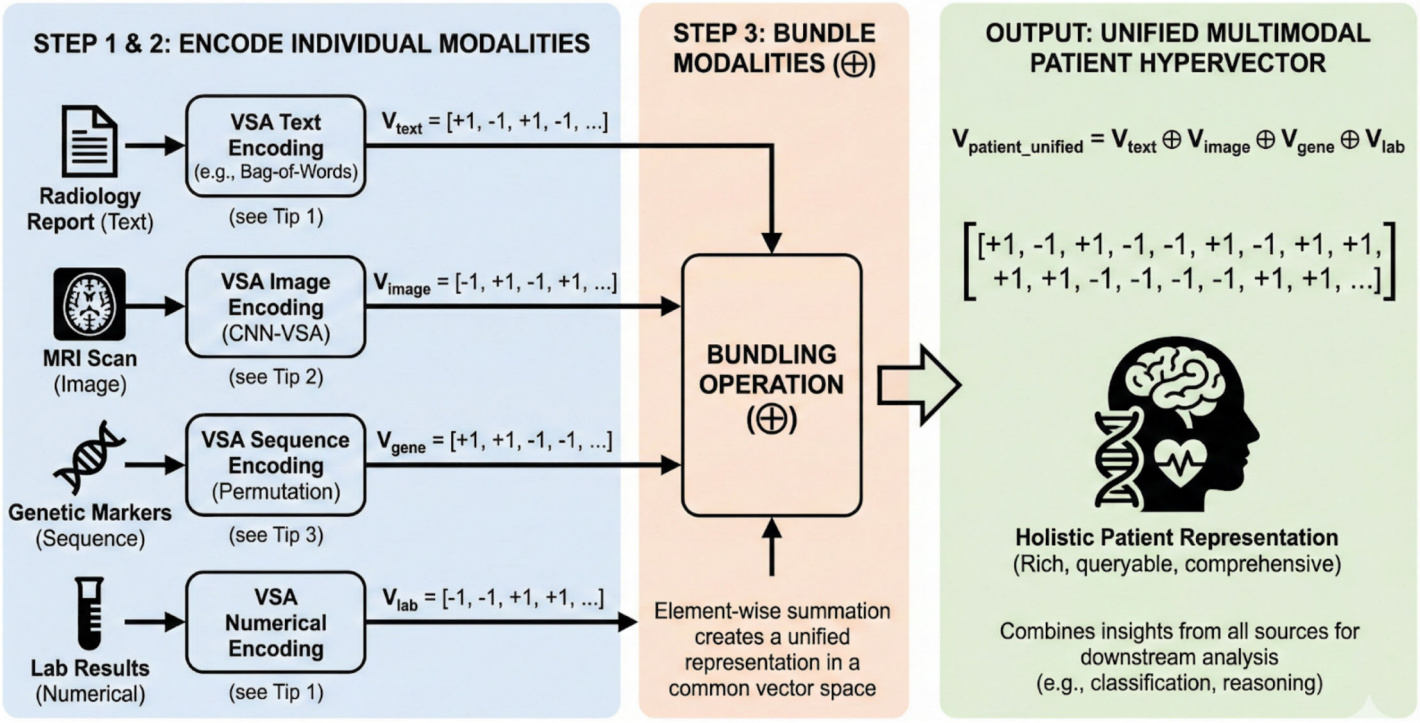

Figure 9 Tip 8 workflow. Power of HDC as a universal language for data integration. Steps 1 and 2 show distinct data modalities, such as text from radiology reports, visual data from MRI scans, genetic sequences, and numerical lab results, being encoded into hypervectors using the specific techniques detailed in previous tips (*e.g.*, Tip 1 for text/numerics, Tip 2 for sequences, Tip 3 for images). Step 3 aggregates these modality-specific vectors *via* the bundling operation. The result is a single unified multimodal patient hypervector. This composite representation superimposes information from all sources into a common vector space, enabling robust downstream tasks like classification that benefit from the combined evidence of all modalities.
Full-size DOI: 10.7717/peerj-cs.3682/fig-9

How to avoid the pitfall: normalize each modality vector before bundling—before you add the vectors together, normalize each one individually. Normalization scales a vector to have a standard magnitude without changing its direction in the high-dimensional space. This ensures that each modality contributes its pattern to the final vector with equal strength, regardless of how many components were used to create it. An alternative is to binarize each incoming vector prior to bundling while preserving a balanced bipolar distribution. This resets the accumulation to binary values and prevents any one data type (*e.g.*, scalar-heavy inputs) from dominating the superposed result.

TIP 9: FOR INTERPRETING RESULTS—PROBE COMPOSITE VECTORS WITH CLEAN POINTERS

In Tip 1, we have built a model that takes a patient's EHR data, encodes it into a single hypervector, and correctly classifies patients as having high risk for a certain disease. In practice, clinicians often need to understand the basis for such a prediction: "Which elements of the patient's record contributed most to the high-risk label?" With many machine learning models (the so-called black boxes), this question is quite challenging to answer because of the model's internal logic.

Asking your vector questions

A composite hypervector is not a black box. It is a queryable database. Because it was constructed using symbolic components, you can reverse the process to inspect its contents. The technique involves using a clean pointer and the inverse of the binding operation to ask the composite vector a question. This process essentially asks "*What information in you is associated with this pointer I am holding?*". This makes VSA models highly interpretable and transparent (*Barkam et al., 2023*; *Mitrokhin et al., 2020*; *Chang et al., 2023*; *Amrouch et al., 2022*).

Let us use the composite *patient_A* we created back in Tip 1. Recall that it was constructed from three facts: a diagnosis of hypertension, a prescription for metformin, and a high HbA1C lab result. Here is the question: "*What was this patient's diagnosis?*"

Step 1: construct your query vector (the clean pointer)—your query is the part of the fact that you already know. In this case, we know the key is *diagnosis*;

Step 2: unbind the pointer from the composite vector—to retrieve the value associated with our pointer, we use the binding operation;

Step 3: find the closest match in your codebook—the resulting vector is not a perfect, clean vector. It is a noisy approximation of the answer because it also contains remnants of all other bundled information. To get our final answer, we search our codebook to find the vector that is most similar to the retrieved one (see Supplemental Code Snippet 15).

Figure 10 below shows an overview of this three steps pipeline.

Pitfall: noisy or ambiguous answers—when unbinding a pointer, the resulting vector is an approximation that is contaminated by noise from all the other bundled vectors. If your composite vector is extremely dense (contains hundreds of bundled facts), this noise can become so strong that the similarity to the correct answer is low, and the similarity to incorrect answers might be non-zero, creating ambiguity.

How to avoid the pitfall: use cleanup memory—the process of finding the closest match in the codebook is itself a form of cleanup. To make it more robust, you can perform an extra cleanup step. Instead of taking just the single best match, take the top 2–3 matches, bundle them together, and then search the codebook again with this new, denoised vector. For most applications, however, a direct search is sufficient. A better long-term solution is to design your encoding scheme hierarchically (as discussed in the solution proposed to avoid Tip 2 pitfall) to limit how many items get bundled into a single vector.

TIP 10: FOR REPRODUCIBILITY AND IMPACT—PRACTICE OPEN SCIENCE

In computational fields, a research article is only half of the story. A article might describe a novel method and present compelling results, but if the underlying code and data are kept private, the work exists in a "silo" (*Sim, 2022*). Other researchers cannot verify the findings, replicate the analysis on their own data, or build upon the method without reinventing it from scratch. This lack of transparency slows down scientific progress and can create a

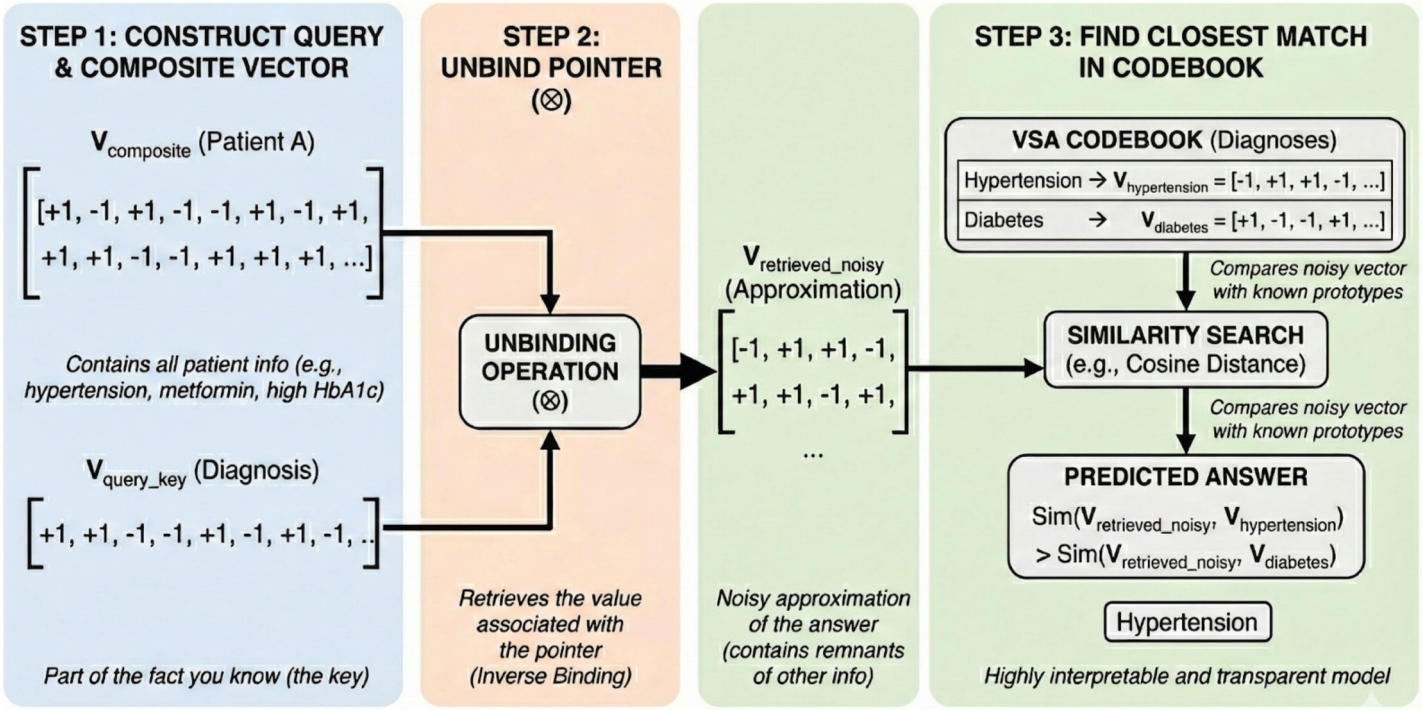

Figure 10 Tip 9 workflow. How to interrogate a composite hypervector to retrieve specific information. Step 1 defines the inputs: a composite vector representing Patient A and a clean pointer vector representing the specific attribute we want to query (*e.g.*, Diagnosis). Step 2 applies the unbinding operation to extract the information associated with the pointer. This yields a noisy retrieved vector, which is an approximation of the original value (*e.g.*, Hypertension) contaminated by the other superimposed information. Step 3 resolves the answer by comparing the noisy vector against the VSA codebook using a similarity search. The system identifies the symbolic concept (*e.g.*, Hypertension) with the highest similarity to the retrieved vector as the predicted answer. Full-size DOI: 10.7717/peerj-cs.3682/fig-10

crisis in reproducibility, where it is impossible to know if a method truly works as described.

The best way to maximize the impact of and trustworthiness of your work is to practice Open Science, guided by the FAIR Guiding Principles (*Wilkinson et al., 2016*). The goal is to make your research outputs, including data, code, and models, Findable, Accessible, Interoperable, and Reusable. This framework moves beyond just making things public, and provides a clear roadmap for creating truly useful and lasting scientific contributions:

- Ensure others can easily discover your work: assign a Digital Object Identifier (DOI) to your code and data by using a repository like Zenodo or Figshare. This makes them easily citable. Also, use rich metadata and keywords when you upload your assets. Describe what the data contains, the VSA parameters used, and the context of the study;
- Make your research available to everyone: publish your code in a public repository like GitHub or GitLab. Submit your manuscript to a fully Open Access Journal to remove paywalls. The protocol for accessing the data should be open and free;
- Ensure your data and models can be combined with other tools and datasets: use common, standard file formats for your input data instead of proprietary formats. Clearly document your VSA parameters, especially the dimensionality and the vector

type. This allows others to integrate your hypervectors with their own VSA-based tools. This documentation must include the random seed used to generate the codebook. While VSA models are statistically robust, meaning that classification accuracy is generally stable across different random initializations, the exact values of the hypervectors depend on the seed. Explicitly setting and reporting the seed is therefore the only way to ensure full reproducibility of your results;

- Enable others to effectively build upon your work: provide clear and comprehensive documentation. A *README.md* file in your code repository should explain what the project does and how to run the analysis, including listing all necessary software libraries and their version. Choose a permissive open-source license (*e.g.*, MIT or Apache 2.0) that explicitly tells others how they are allowed to reuse your code.

Following the FAIR principles directly leads to more robust science. It enhances reproducibility, foster collaborations, increase the visibility and impact of your work, and builds trust within the scientific community. Currently, considering the recent VSA architecture open-source platforms, researchers can take advantage of libraries such as TorchHD (*Heddes et al., 2023*), DistHD (*Wang, Huang & Imani, 2023*), OpenHD (*Kang et al., 2022*), HDTorch (*Simon et al., 2022*), uHD (*Aygun, Moghadam & Najafi, 2024*) and *hdlib* (*Cumbo, Weitschek & Blankenberg, 2023*), the one mainly used for this work. Unlike some other frameworks that focus narrowly on GPU acceleration or domain-specific encodings, *hdlib* was designed from the start as a general-purpose library for building VSAs. *hdlib* offers a clean, modular structure for defining high-dimensional spaces and vectors, performing the canonical operations of bind, bundle, and permute, and supporting supervised learning with built-in cross-validation, auto-tuning, and even feature selection, making it a versatile framework.

CONCLUSIONS

VSAs provide a powerful and intuitive paradigm for tackling the complex, heterogeneous, and often incomplete data that characterizes the biomedical sciences. This work has moved beyond abstract theory and offers a set of practical, application-driven tips for researchers. We have demonstrated how the core VSA operations of binding, bundling, and permutation can be combined to solve diverse challenges.

We began by showing how to construct holistic representations from disparate data sources. For electronic health records, we demonstrated how bundling can create robust patient vectors that naturally handle missing information (Tip 1), while for molecular structures, we showed how to encode a molecule as a sum of its context-aware atomic fragments (Tip 5). The critical role of order was addressed for genomics and proteomics, where permutation is key to encoding sequences (Tip 2), and for biosignals like EEG, where symbolic snapshots can capture dynamic states over time (Tip 4).

We also illustrated how VSAs can be applied to high-level reasoning and data fusion tasks. We explored a method for building prototype vectors to create simple yet powerful classifiers (Tip 7) and showed how VSAs can represent vast biomedical knowledge graphs as queryable vectors (Tip 6). Furthermore, we detailed how VSAs can augment other

computational methods, such as combining them with CNNs to build explainable models for medical imaging (Tip 3), and how they provide a universal language for the fusion of multimodal data into a single, unified representation (Tip 8).

A recurring theme throughout these tips is the shift from black box models to transparent systems. The ability to probe a composite hypervector to ask "why" (Tip 9) is a standout feature that addresses a key limitation of many conventional methods. Additionally, we emphasize the importance of renormalization as a recurring safeguard. Whether creating patient vectors (Tip 1), prototypes (Tip 7), or fused modalities (Tip 8), ensuring components contribute equally to the composite representation is critical for model performance. Ultimately, the power of any computational method is magnified by the community that uses it. For this reason, we concluded with a crucial tip on practicing FAIR and Open Science (Tip 10). A well-designed model becomes truly transformative only when its code and derived data are shared openly, allowing for verification, replication, and extension by the entire scientific community.

As the scale and complexity of biomedical data continue to grow, brain-inspired approaches like HDC will be indispensable. By moving beyond pure pattern recognition towards models that can represent, reason with, and explain complex information, researchers are now better equipped to build the next generation of tools to unlock new insights from the full spectrum of biomedical data.

ABBREVIATIONS

AI	Artificial Intelligence
CNN	Convolutional Neural Network
DOI	Digital Object Identifier
ECG	Electrocardiography
EEG	Electroencephalography
EHR	Electronic Health Record
EMG	Electromyography
FAIR	Findability, Accessibility, Interoperability, and Reusability
HDC	Hyperdimensional Computing
MAP	Multiply-Add-Permute
MRI	Magnetic Resonance Imaging
VSA	Vector-Symbolic Architecture

ACKNOWLEDGEMENTS

The authors acknowledge the use of Large Language Models (LLM—Google's Gemini 2.5 Pro and Nano Banana Pro) to enhance the clarity and readability of this manuscript. The LLM's role was strictly limited to improving prose and sentence structure. The conceptual framework of this manuscript, the formulation of tips, the design of the code examples, and all scientific recommendations were developed exclusively by the authors.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

Fabio Cumbo and Davide Chicco are Academic Editors for PeerJ Computer Science.

Author Contributions

- Fabio Cumbo conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.
- Davide Chicco analyzed the data, performed the computation work, authored or reviewed drafts of the article, and approved the final draft.
- Sercan Aygun performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.
- Daniel Blankenberg analyzed the data, performed the computation work, authored or reviewed drafts of the article, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

All source code required to reproduce the examples in this manuscript is available at GitHub and Zenodo:

- https://github.com/cumbof/Biomed-VSAs.
- Fabio Cumbo. (2025). cumbof/Biomed-VSAs: Biomed VSAs v1.0 (1.0). Zenodo. https://doi.org/10.5281/zenodo.17107790.

Supplemental Information

Supplemental information for this article can be found online at http://dx.doi.org/10.7717/peerj-cs.3682#supplemental-information.

REFERENCES

Alotaiby TN, Alshebeili SA, Alshawi T, Ahmad I, Abd El-Samie FE. 2014. EEG seizure detection and prediction algorithms: a survey. *EURASIP Journal on Advances in Signal Processing* **2014(1)**:1–21 DOI 10.1186/1687-6180-2014-183.

Amrouch H, Imani M, Jiao X, Aloimonos Y, Fermuller C, Yuan D, Ma D, Barkam HE, Genssler PR, Sutor P. 2022. Brain-inspired hyperdimensional computing for ultra-efficient edge AI. In: *2022 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Piscataway: IEEE DOI 10.1109/codes-isss55005.2022.00017.

Asgarinejad F, Thomas A, Rosing T. 2020. Detection of epileptic seizures from surface EEG using hyperdimensional computing. In: *The Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. Vol. 2020, Piscataway: IEEE, 536–540 DOI 10.1109/EMBC44109.2020.9175328.

Aygun S, Moghadam MS, Najafi MH. 2024. UHD: unary processing for lightweight and dynamic hyperdimensional computing. In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Piscataway: IEEE DOI 10.23919/date58400.2024.10546545.

Bai T, Chanda AK, Egleston BL, Vucetic S. 2018. EHR phenotyping via jointly embedding medical concepts and words into a unified vector space. *BMC Medical Informatics and Decision Making* **18(S4)**:15–25 DOI 10.1186/s12911-018-0672-0.

Barkam HE, Yun S, Chen H, Gensler P, Mema A, Ding A, Michelogiannakis G, Amrouch H, Imani M. 2023. Reliable hyperdimensional reasoning on unreliable emerging technologies. In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. Piscataway: IEEE DOI 10.1109/iccad57390.2023.10323935.

Bonner S, Barrett IP, Ye C, Swiers R, Engkvist O, Bender A, Hoyt CT, Hamilton WL. 2022. A review of biomedical datasets relating to drug discovery: a knowledge graph perspective. *Briefings in Bioinformatics* **23(6)**:167 DOI 10.1093/bib/bbac404.

Bugnon LA, Yones C, Milone DH, Stegmayer G. 2020. Deep neural architectures for highly imbalanced data in bioinformatics. *IEEE Transactions on Neural Networks and Learning Systems* **31(8)**:2857–2867 DOI 10.1109/TNNLS.2019.2914471.

Burrello A, Schindler K, Benini L, Rahimi A. 2018. One-shot learning for iEEG seizure detection using end-to-end binary operations: local binary patterns with hyperdimensional computing. In: *2018 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. Piscataway: IEEE DOI 10.1109/BIOCAS.2018.8584751.

Burrello A, Schindler K, Benini L, Rahimi A. 2020. Hyperdimensional computing with local binary patterns: one-shot learning of seizure onset and identification of ictogenic brain regions using short-time iEEG recordings. *IEEE Transactions on Biomedical Engineering* **67(2)**:601–613 DOI 10.1109/TBME.2019.2919137.

Chang C-Y, Chuang Y-C, Huang C-T, Wu A-Y. 2023. Recent progress and development of hyperdimensional computing (HDC) for edge intelligence. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* **13(1)**:119–136 DOI 10.1109/jetcas.2023.3242767.

Chang E-J, Rahimi A, Benini L, Wu A-YA. 2019. Hyperdimensional computing-based multimodality emotion recognition with physiological signals. In: *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. Piscataway: IEEE DOI 10.1109/AICAS.2019.8771622.

Chicco D, Cumbo F, Angione C. 2023. Ten quick tips for avoiding pitfalls in multi-omics data integration analyses. *PLOS Computational Biology* **19(7)**:e1011224 DOI 10.1371/journal.pcbi.1011224.

Cumbo F, Cappelli E, Weitschek E. 2020. A brain-inspired hyperdimensional computing approach for classifying massive DNA methylation data of cancer. *Algorithms* **13(9)**:233 DOI 10.3390/a13090233.

Cumbo F, Chicco D. 2025. Hyperdimensional computing in biomedical sciences: a brief review. *PeerJ Computer Science* **11(4)**:e2885 DOI 10.7717/peerj-cs.2885.

Cumbo F, Dhillon K, Joshi J, Chicco D, Aygun S, Blankenberg D. 2025a. A novel vector-symbolic architecture for graph encoding and its application to viral pangenome-based species classification. BioRxiv DOI 10.1101/2025.09.08.674958.

Cumbo F, Dhillon K, Joshi J, Raubenolt B, Chicco D, Aygun S, Blankenberg D. 2025b. Predicting the toxicity of chemical compounds via hyperdimensional computing. BioRxiv DOI 10.1101/2025.09.12.675894.

Cumbo F, Truglia S, Weitschek E, Blankenberg D. 2025c. Feature selection with vector-symbolic architectures: a case study on microbial profiles of shotgun metagenomic samples of colorectal cancer. *Briefings in Bioinformatics* **26(2)**:bbaf177 DOI 10.1093/bib/bbaf177.

Cumbo F, Weitschek E, Blankenberg D. 2023. hdlib: a Python library for designing vector-symbolic architectures. *Journal of Open Source Software* **8(89)**:5704 DOI 10.21105/joss.05704.

Du Y, Ren Y, Wong N, Ngai ECH. 2024. Hyperdimensional computing with multiscale local binary patterns for scalp EEG-based epileptic seizure detection. *IEEE Internet of Things Journal* **11(15)**:26046–26061 DOI 10.1109/jiot.2024.3395496.

Ein Shoka AA, Dessouky MM, El-Sayed A, Hemdan EE-D. 2023. EEG seizure detection: concepts, techniques, challenges, and future trends. *Multimedia Tools and Applications* **82(27)**:42021–42051 DOI 10.1007/s11042-023-15052-2.

Emmanuel T, Maupong T, Mpoeleng D, Semong T, Mphago B, Tabona O. 2021. A survey on missing data in machine learning. *Journal of Big Data* **8(1)**:1–37 DOI 10.1186/s40537-021-00516-9.

Fanizzi N, d'Amato C. 2025. The blessing of dimensionality. *Neurosymbolic Artificial Intelligence* **1**:259 DOI 10.3233/NAI-240675.

Ge L, Parhi KK. 2022. Applicability of hyperdimensional computing to seizure detection. *IEEE Open Journal of Circuits and Systems* **3(4)**:59–71 DOI 10.1109/ojcas.2022.3163075.

Gramfort A, Luessi M, Larson E, Engemann DA, Strohmeier D, Brodbeck C, Goj R, Jas M, Brooks T, Parkkonen L, Hamalainen M. 2013. MEG and EEG data analysis with MNE-Python. *Frontiers in Neuroscience* **7**:267 DOI 10.3389/fnins.2013.00267.

Harnoune A, Rhanoui M, Mikram M, Yousfi S, Elkaimbillah Z, El Asri B. 2021. BERT based clinical knowledge extraction for biomedical knowledge graph construction and analysis. *Computer Methods and Programs in Biomedicine Update* **1(2)**:100042 DOI 10.1016/j.cmpbup.2021.100042.

Hassan E, Halawani Y, Mohammad B, Saleh H. 2022. Hyper-dimensional computing challenges and opportunities for AI applications. *IEEE Access* **10(32)**:97651–97664 DOI 10.1109/access.2021.3059762.

Heddes M, Nunes I, Vergés P, Kleyko D, Abraham D, Givargis T, Nicolau A, Veidenbaum A. 2023. Torchhd: an open source Python library to support research on hyperdimensional vomputing and vector symbolic architectures. *Journal of Machine Learning Research* **24**:1–10. *Available at http://jmlr.org/papers/v24/23-0300.html*.

Imani M, Nassar T, Rahimi A, Rosing T. 2018. HDNA: energy-efficient DNA sequencing using hyperdimensional computing. In: *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*. Piscataway: IEEE DOI 10.1109/BHI.2018.8333421.

Joel LO, Doorsamy W, Paul BS. 2022. A review of missing data handling techniques for machine learning. *IJITIS* **5**:971–1005 DOI 10.15157/IJITIS.2022.5.3.971-1005.

Jones JA, Farnell B. 2007. Missing and incomplete data reduces the value of general practice electronic medical records as data sources in research. *Australian Journal of Primary Health* **13(1)**:74–80 DOI 10.1071/py07010.

Jones D, Zhang X, Bennion BJ, Pinge S, Xu W, Kang J, Khaleghi B, Moshiri N, Allen JE, Rosing TS. 2024. HDBind: encoding of molecular structure with hyperdimensional binary representations. *Scientific Reports* **14(1)**:1–16 DOI 10.1038/s41598-024-80009-w.

Kanerva P. 2000. Large patterns make great symbols: an example of learning from example. *Hybrid Neural Systems* **1778(1–2)**:194–203 DOI 10.1007/10719871_13.

Kanerva P. 2009. Hyperdimensional computing: an introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation* **1(2)**:139–159 DOI 10.1007/s12559-009-9009-8.

Kanerva P. 2022. Hyperdimensional computing: an algebra for computing with vectors. In: *Advances in Semiconductor Technologies*. Hoboken: Wiley, 25–42 DOI 10.1002/9781119869610.ch2.

Kang J, Khaleghi B, Rosing T, Kim Y. 2022. OpenHD: a GPU-powered framework for hyperdimensional computing. *IEEE Transactions on Computers* **71(11)**:2753–2765 DOI 10.1109/tc.2022.3179226.

Kleyko D, Rachkovskij DA, Osipov E, Rahimi A. 2022. A survey on hyperdimensional computing aka vector symbolic architectures, Part I: models and data transformations. *ACM Computing Surveys* **55(6)**:1–40 DOI 10.1145/3538531.

Kleyko D, Rachkovskij D, Osipov E, Rahimi A. 2023. A survey on hyperdimensional computing aka vector symbolic architectures, Part II: applications, cognitive models, and challenges. *ACM Computing Surveys* **55(9)**:1–52 DOI 10.1145/3558000.

Kuo T-T, Rao P, Maehara C, Doan S, Chaparro JD, Day ME, Farcas C, Ohno-Machado L, Hsu CN. 2016. Ensembles of NLP tools for data element extraction from clinical notes. *AMIA Annual Symposium Proceedings* **2016**:1880–1889. *Available at https://pmc.ncbi.nlm.nih.gov/articles/PMC5333200/*.

Lee H, Kim J, Chen H, Zeira A, Srinivasa N, Imani M, Kim Y. 2023. Comprehensive integration of hyperdimensional computing with deep learning towards neuro-symbolic AI. In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. Piscataway: IEEE DOI 10.1109/dac56929.2023.10248004.

Li Q, Cai W, Wang X, Zhou Y, Feng DD, Chen M. 2014. Medical image classification with convolutional neural network. In: *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*. Piscataway: IEEE DOI 10.1109/ICARCV.2014.7064414.

Luczak P, Slot K, Kucharski J. 2022. Combining deep convolutional feature extraction with hyperdimensional computing for visual object recognition. In: *2022 International Joint Conference on Neural Networks (IJCNN)*. Piscataway: IEEE DOI 10.1109/ijcnn55064.2022.9892281.

Ma D, Thapa R, Jiao X. 2022. MoleHD: efficient drug discovery using brain inspired hyperdimensional computing. In: *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. Piscataway: IEEE DOI 10.1109/bibm55620.2022.9995708.

Mitrokhin A, Sutor P, Summers-Stay D, Fermüller C, Aloimonos Y. 2020. Symbolic representation and learning with hyperdimensional computing. *Frontiers in Robotics and AI* **7**:535245 DOI 10.3389/frobt.2020.00063.

Nair DR, Purushothaman A. 2019. Brain inspired one shot learning method for HD computing. *VLSI Design and Test* **1066(9)**:286–297 DOI 10.1007/978-981-32-9767-8_25.

Nicholson DN, Greene CS. 2020. Constructing knowledge graphs and their biomedical applications. *Computational and Structural Biotechnology Journal* **18**:1414–1428 DOI 10.1016/j.csbj.2020.05.017.

Pale U, Teijeiro T, Atienza D. 2021. Systematic assessment of hyperdimensional computing for epileptic seizure detection. In: *The Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. Vol. 2021, Piscataway: IEEE, 6361–6367 DOI 10.1109/EMBC46164.2021.9629648.

Pes B. 2020. Learning from high-dimensional biomedical datasets: the issue of class imbalance. *IEEE Access* **8**:13527–13540 DOI 10.1109/access.2020.2966296.

Plate TA. 1995. Holographic reduced representations. *IEEE Transactions on Neural Networks* **6(3)**:623–641 DOI 10.1109/72.377968.

Rahimi A, Tchouprina A, Kanerva P, del Millán JR, Rabaey JM. 2017. Hyperdimensional computing for blind and one-shot classification of EEG error-related potentials. *Mobile Networks and Applications* **25(5)**:1958–1969 DOI 10.1007/s11036-017-0942-6.

Rizal A, Priharti W, Hadiyoso S. 2021. Seizure detection in epileptic EEG using Short-Time Fourier Transform and support vector machine. *International Journal of Online and Biomedical Engineering (iJOE)* **17(14)**:65–78 DOI 10.3991/ijoe.v17i14.25889.

Rizvi STH, Latif MY, Amin MS, Telmoudi AJ, Shah NA. 2023. Analysis of machine learning based imputation of missing data. *Cybernetics and Systems* **56(6)**:818–832 DOI 10.1080/01969722.2023.2247257.

Salehi AW, Khan S, Gupta G, Alabduallah BI, Almjally A, Alsolai H, Siddiqui T, Mellit A. 2023. A study of CNN and transfer learning in medical imaging: advantages, challenges, future scope. *Sustainability* **15(7)**:5930 DOI 10.3390/su15075930.

Schindler KA, Rahimi A. 2021. A primer on hyperdimensional computing for iEEG seizure detection. *Frontiers in Neurology* **12**:701791 DOI 10.3389/fneur.2021.701791.

Schlegel K, Neubert P, Protzel P. 2021. A comparison of vector symbolic architectures. *Artificial Intelligence Review* **55(6)**:4523–4555 DOI 10.1007/s10462-021-10110-3.

Schmuck M, Benini L, Rahimi A. 2019. Hardware optimizations of dense binary hyperdimensional computing: rematerialization of hypervectors, binarized bundling, and combinational associative memory. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* **15(4)**:1–25 DOI 10.1145/3314326.

Sheikhalishahi S, Miotto R, Dudley JT, Lavelli A, Rinaldi F, Osmani V. 2019. Natural language processing of clinical notes on chronic diseases: systematic review. *JMIR Medical Informatics* **7(2)**:e12239 DOI 10.2196/12239.

Sim I. 2022. Data sharing and reuse. *Principles and Practice of Clinical Trials* **37**:2137–2158 DOI 10.1007/978-3-319-52636-2_190.

Simon WA, Pale U, Teijeiro T, Atienza D. 2022. HDTorch: accelerating hyperdimensional computing with GP-GPUs for design space exploration. In: *ICCAD '22: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. Piscataway: IEEE DOI 10.1145/3508352.3549475.

Stock M, Van Criekinge W, Boeckaerts D, Taelman S, Van Haeverbeke M, Dewulf P, De Beats B. 2024. Hyperdimensional computing: a fast, robust, and interpretable paradigm for biological data. *PLOS Computational Biology* **20(9)**:e1012426 DOI 10.1371/journal.pcbi.1012426.

Sæthre E, Osborg Ose S, Krokstad S, Østgård Gismervik S. 2025. "Terrible Stuff. We've been had": hospital staff reactions to a new electronic health record and implications for employee well-being—a qualitative study. *International Journal of Medical Informatics* **204(Suppl 4)**:106039 DOI 10.1016/j.ijmedinf.2025.106039.

Tavabi N, Singh M, Pruneski J, Kiapour AM. 2024. Systematic evaluation of common natural language processing techniques to codify clinical notes. *PLOS ONE* **19(3)**:e0298892 DOI 10.1371/journal.pone.0298892.

Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J, van der Walt SJ, Brett M, Wilson J, Millman KJ, Mayorov N, Nelson ARJ, Jones E, Kern R, Larson E, Carey CJ, Polat I, Feng Y, Moore EW, VanderPlas J. 2020. SciPy 1.0 Contributors. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* **17**:261–272 DOI 10.1038/s41592-019-0686-2.

Wang J, Huang S, Imani M. 2023. DistHD: a learner-aware dynamic encoding method for hyperdimensional classification. In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. Piscataway: IEEE DOI 10.1109/dac56929.2023.10247876.

Wells BJ, Chagin KM, Nowacki AS, Kattan MW. 2013. Strategies for handling missing data in electronic health record derived data. *EGEMS* **1(3)**:1035 DOI 10.13063/2327-9214.1035.

Wilkinson MD, Dumontier M, Aalbersberg IJ, Appleton G, Axton M, Baak A, Blomberg N, Boiten JW, de Silva Santos LB, Bourne PR, Bowman J, Brookes AJ, Clark T, Crosas M, Dillo I, Dumon O, Edmunds S, Evelo CT, Finkers R, Gonzalez-Beltran A, Gray AJG, Groth P, Goble C, Grethe JS, Heringa J, Hoen PAC, Hooft R, Kuhn T, Kok R, Kok J, Lusher SJ, Martone ME, Mons A, Packer AL, Persson B, Rocca-Serra P, Roos M, van Schaik R, Swertz MA, Thompson M, van der Lai J, van Mulligen E, Velterop J, Waagmeester A, Wittenburg P, Wolstencroft K, Zhao J, Mons B. 2016. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data* **3**:1–9 DOI 10.1038/sdata.2016.18.

Xie Y, Zhang J, Shen C, Xia Y. 2021. CoTr: efficiently bridging CNN and transformer for 3D medical image segmentation. In: *Medical Image Computing and Computer Assisted Intervention—MICCAI 2021*, 171–180 DOI 10.1007/978-3-030-87199-4_16.

Xu W, Hsu P-K, Moshiri N, Yu S, Rosing T. 2024. HyperGen: compact and efficient genome sketching using hyperdimensional vectors. *Bioinformatics* **40(7)**:btae452 DOI 10.1093/bioinformatics/btae452.

Yao W, Bai J, Liao W, Chen Y, Liu M, Xie Y. 2024. From CNN to transformer: a review of medical image segmentation models. *Journal of Imaging Informatics in Medicine* **37(4)**:1529–1547 DOI 10.1007/s10278-024-00981-7.

Yuan F, Zhang Z, Fang Z. 2023. An effective CNN and transformer complementary network for medical image segmentation. *Pattern Recognition* **136(11)**:109228 DOI 10.1016/j.patcog.2022.109228.

Zhang S, Juretus K, Jiao X. 2025. Exploring hyperdimensional computing robustness against hardware errors. *IEEE Transactions on Computers* **74(6)**:1963–1977 DOI 10.1109/tc.2025.3547142.

Zhang S, Wang R, Zhang JJ, Rahimi A, Jiao X. 2021. Assessing robustness of hyperdimensional computing against errors in associative memory: (invited paper). In: *2021 IEEE 32nd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. Piscataway: IEEE DOI 10.1109/ASAP52443.2021.00039.

Zhao Q, Yu X, Hu S, Rosing T. 2024. MultimodalHD: federated learning over heterogeneous sensor modalities using hyperdimensional computing. In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Piscataway: IEEE, 1–6 DOI 10.23919/DATE58400.2024.10546794.

Zou Z, Chen H, Poduval P, Kim Y, Imani M, Sadredini E, Cammarota R, Imani M. 2022. BioHD: an efficient genome sequence search platform using HyperDimensional memorization. In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM DOI 10.1145/3470496.3527422.