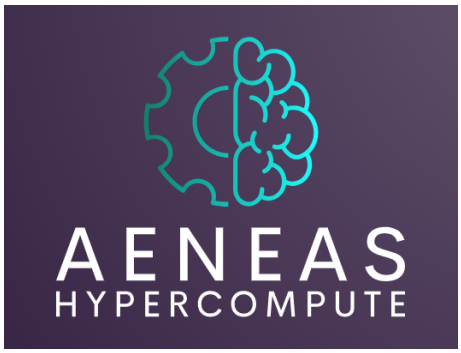


Aeneas HyperCompute Reference Manual

Version 3.0.0

Marco Angioli[1], Saeid Jamili[2]

[1]Contact: marco.angioli@uniroma1.it
[2]Contact: saeid.jamili@uniroma1.it

Contents

1 Hyperdimensional Computing: From Theory to Practice

In this section of the documentation, we will delve into the fundamental theory behind Hyperdimensional Computing (HDC) and explore the various techniques provided by our framework.

1.1 Fundamentals

Hyperdimensional Computing (HDC) or Vector Symbolic Architecture (VSA) is a computing framework that uses high-dimensional distributed representations inspired by the human brain's architecture.

In the brain, the information is *distributed* across an extremely high number of neurons. Each neuron can be activated or deactivated in response to a specific stimulus, and it is the state of all the neurons together that represents a particular concept or object. This means that the representation of an object is **distributed**: each object is represented by a subset of activated neurons, but each neuron can belong to the representations of many objects. This means that, in distributed representations, the state of individual components of the vector can not be interpreted without knowing the states of the others. In other words, the semantics of an individual component are usually undefined.

HDC emulates this behaviour through high-dimensional distributed holographic representations. In this framework, every object or concept is represented with a hypervector (HV) which components can be binary, bipolar, real or complex numbers, as we will see in the next sections. In the following we will refer to the size of the HVs as *HD_DIM*.

1.2 Why HDC? Application and properties

The rise of the Internet of Things (IoT) has sparked considerable interest in deploying Artificial Intelligence (AI) models on edge devices. However, current state-of-the-art neural networks (NN) are not suitable for such applications due to their requirements of high precision, computational complexity, execution time, and energy consumption.

While NNs are also inspired by the brain, their modern implementations increasingly deviate from neural plausibility. These methods often incorporate

choices that are not neurally realistic, possess significant depth, and rely on backpropagation as a training mechanism.

In this context, HDC has garnered significant attention in the field, offering several advantages:

- Low latency;
- High energy efficiency;
- Very high robustness against noise and hardware faults;
- Transparent models;
- One-shot learning;
- Almost same architecture for train and inference;
- Extreme parallelism.

All these properties naturally emerge from the fundamental structure of the hyperspace. In the following, some of them are analyzed in detail.

1.2.1 Fault Resilience

By its very nature, HDC is extremely robust in the presence of failures, defects, variations and noise, all of which are synonymous of ultralow energy consumption. It has been, in fact, demonstrated how the HDC degrades very gracefully in the presence of temporary and permanent faults compared to baseline KNN classifiers, for example. In a language recognition task, by injecting hardware errors, HDC tolerates 8.8x higher probability of failure per individual memory cell. Considering the permanent hard errors, HDC tolerates 60x higher probability of failures.

Such robustness of HDC is achieved thanks to the pseudorandomness hyperdimensionality and fully distributed, holographic representations. The starting atomic vectors are represented with i.i.d. components and are then combined using arithmetic operations that preserve the i.i.d.. When combining different letters HVs for creating an n-gram, the components of the final vectors will be identically distributed and nearly independent. This means that a failure in a component does not propagate. Also all the errors are still compensated by the holographic nature: the error-free components can still provide a useful representation that is similar enough to the original one.

This inherent robustness also eliminates the need for asymmetric error protection in memory units. This type of robustness is actually unique and enables aggressive scaling of the device.

1.2.2 Transparency

Thanks to the well-defined set of arithmetic operations with inverses, HDC produces transparent (i.e. analyzable) codes with interpretable features. This is a key and important property of HDC that strongly contrasts with the usual blind application of conventional learning methods that are treated and considered as "black boxes".

1.2.3 One-shot learning

In contrast to other neuro-inspired approaches in which learning is more computationally demanding than inference, in HD, the learning procedure is based on the same algorithms as classification. In particular, the training process works in "one shot": in a single pass, only a portion of the total training dataset can be used to train the model. For example, HD reaches the same performance as state-of-the-art approaches using only 1/3 of the total training dataset, which means 3x faster.

Another important property of HD is that the same algorithms are actually used both for training and testing. This make the architecture ideal for online and continuous learning. New examples can be learned by just incrementally updating the associative memory (AM). This is also really important to face nonstationary problems.

1.2.4 Energy Efficiency

HDC is all about manipulating and comparing large patterns within the memory itself. The operations allow a high degree of parallelism by needing to communicate with only local components or immediate neighbours. This also means that the logic can be tightly integrated with the memory, and all computations can be distributed to save energy. This forms a fundamental departure from the traditional Von-Neumann architectures, where data must be transported to the processing unit and back, creating memory walls.

Also, HDC requires far fewer operations than other approaches such ass SVMs, CNNs, KNNs, MLP. For instance, for EMG applications, HDC requires 1/2 of the total power compared to SVM, when implemented on the commercial embedded ARM Cortex M4. **[FONTE]**

The same is true about memory access, this compensates for the very wide words used in HDC. The memory requirements for HDC scales linearly, while it is not true for other networks. This means that, for example, a KNN for

language recognition requires 500x larger memory than HD. In the seizure detection task, the MLP demands 5x-13x larger memory than HDC to store its weights, also supposing to perform quantization **[FONTE]**

Finally, a very good point for HD is the sparsity in time and space. At any time instant, only a very low fraction of the memory/logic will be active. For instance, a sparse binary representation- where the number of ones is significantly less than zeros-can lower the switching activity and, then, the power consumption.

1.3 HV properties and basic operations

Exploring the size, data type, density and basic arithmetic of HyperVectors, this subsection examines the foundational elements central to HDC models. The HV size (D) determines the number of independent elements that can be stored in the hyperspace, also called capacity, following an exponential relationship [citation]. Often, D is set to a high value, such as thousands of dimensions (e.g., $D = 10,000$), but in resource-constrained contexts, selecting the appropriate size is crucial, as it directly impacts computational complexity, memory usage, energy consumption, and hardware requirements.

The HV's elements data type is another essential characteristic of the high-dimensional space. HDC allows for various data types, including binary {0,1}, bipolar {-1,1}, ternary {-1,0,1}, quantized, integers or real [**?**]. The wider the range of data, the better the model's performance, but this improvement comes at the cost of increased complexity, energy consumption, execution time, and area occupation.

Another important parameter is the HV's *density*. In dense HV, all the elements have an equal distribution probability, while in a sparse model, the 0 value dominates the HV with a low presence for 1 or -1. In [**?**], the impact of different density percentages has been investigated on the model's computational complexity and accuracy.

A hyperdimensional computing model is defined by three main operations that are used to manage and combine HVs:

- Bundling (or superposition);
- Binding;
- Permutation;

and by a *similarity function* used to compute the distance between HVs in the reference hyperspace. In this section, all the basic concepts about the HVs arithmetic are analyzed and discussed.

1.3.1 Superposition or Bundling

The superposition is the most basic operation used to form an HV that represents several HVs. In analogy to simultaneous activation of neural patterns, this operation is usually implemented as the addition of HVs.

$$s = a + b + c$$

Properties:

- the result of the bundling is is similar to both its input HVs.
- the bundling is commutative
- the bundling is associative

1.3.2 Binding

The bundling operation alone, due to its commutative property, leads to the superposition catastrophe. During the recursive application of the superposition, in fact, the information about combinations of the initial objects (groups) is lost. Ex:

$$(a + b) + (c + d) = a + b + c + d$$

This issue needs to be addressed to represent compositional structures in the hyperspace.

HDC uses a *binding* operation, denoted as $\cdot$ and usually implemented with multiplication-like procedures, to solve this problem.

- The binding does not preserve similarity between the resulting hypervectors and the operands. The resulting HV obtained after is dissimilar to both its input HVs.
- The binding preserves the so-called structured similarity. If HV a is similar to HV a' and b is similar to b', then: $d = a \cdot b$ is similar to $d' = a' \cdot b'$
- The binding operation is invertible. The result of binding can be reversed by the *unbinding or release* technique. In many models the binding operation is self-inverse, meaning that the unbinding is the same operation, simply applied again.

1.3.3 Permutation

operation denoted as $\rho(a)$. The permuted HV is dissimilar to the original one. A tipical implementation of the permutation operation is the cyclic shift that is very easy to implement. As for the binding, the permutation preserve the structured similarity. If a is similar to a' then $\rho(a)$ is similar to $\rho(a')$

1.3.4 Clipping

When superimposing binary or bipolar vectors, the resulting vector won't belong anymore to the starting hyperspace but will have integer elements. Clipping techniques can be applied to adjust the data in different ranges, such as binary, bipolar, tripolar, power of two or quantized. For instance, for obtaining bipolar HVs, the sign function is applied to each element, while for binary HVs, the majority rule shown in (6) is used, where $BundledHV_d$ is the d^{th} dimension of the $BundledHV$. This makes a trade-off between accuracy and hardware complexity.

$$BundledHV_d = \begin{cases} 0 & BundledHV_d < \frac{\text{n.bundled HVs}}{2} \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

1.3.5 Similarity Function

In HDC the concept of similarity plays a pivotal role for performing classification and logical tasks. Similarity measures provide a means to compute the degree of resemblance or difference between two HVs. Essentially, these measures calculate the angle between HVs in the hyperspace, offering a geometric interpretation of similarity.

As outlined in Table 1, the choice of similarity measure depends on the type of HVs involved. For binary HVs, the Hamming Distance is used, while for integer or bipolar ones, the Dot Product and Cosine Similarity are preferred. The Dot Product is a straightforward and computationally efficient measure that quantifies similarity in terms of the sum of the products of corresponding elements in the HVs. On the other hand, Cosine Similarity offers a normalized measure independent of the magnitude of the HVs, making it ideal for comparing the directional similarity between two vectors.

An interesting aspect to note is that when HVs are quantized in powers-of-two, the computation of Cosine Similarity can be significantly optimized. In such cases, the traditional multiplication and division operations required for calculating Cosine Similarity can be replaced with bitwise shifts. This approach leverages the binary nature of the data, leading to a substantial increase in computational efficiency. Such an optimization is particularly

valuable in large-scale HDC applications where speed and resource efficiency are of the essence.

Type of HV	Similarity Measures
Binary HV	Hamming Distance
Integer or Bipolar HV	Dot Product, Cosine Similarity

Table 1: Similarity Measures in HDC based on HV Type

1.3.6 Context Dependent Thinning (CDT)

In sparse distributed hypervectors, the bundling operation requires an additional step denoted as context-dependent thinning (CDT). CDT is a mathematical operation that reduces the sparsity after a bundling, keeping it constant. In SDR, the bundle can be a simple OR operation. However, every time an OR is performed, the HV's sparsity decreases, meaning that the density of the resultant HD vector increases with the number of superimposed HVs. This can be controlled through CDT, which formulation is reported in 2

$$\begin{aligned} \mathbf{Z} &= \bigvee_{i=1}^{G} \mathbf{x}_i \\ \langle \mathbf{Z} \rangle &= \bigvee_{k=1}^{T} (\mathbf{Z} \wedge \rho_k(\mathbf{Z})) = \mathbf{Z} \wedge \left(\bigvee_{k=1}^{T} \rho_k(\mathbf{Z}) \right) \end{aligned} \tag{2}$$

1.4 Mapping: from Basics to Complex Structures

In this section, we delve into an extensive analysis of the mapping mechanism exploited in HDC to encode the input data in the Hyperspace. We start with fundamental and basic elements and progressively introduce more complex data structures that are necessary to model, treat and automate classification problems.

1.4.1 Basic Object Encoding

The fundamental entities/items/concepts/symbols/scalars in a given problem are encoded in the so-called *atomicHVs* that are stored in memory and represent the dictionary available to construct all the other vectors. The approach employed to generate these atomicHVs depends on the data characteristics. In fact, while it's crucial for entirely independent and disparate

elements to exhibit zero similarity in the hyperspace, there is also a requirement to preserve relationships among continuous values, such as scalars.

These HVs are combined using the previously introduced basic operations to establish connections and represent more complex elements. This approach enables learning problems such as logical reasoning, classification, regression and clustering.

1.4.2 Independent items

When dealing with independent and dissimilar objects, the mapping procedure involves a straightforward random sampling of the HV from the hyperspace. This is attributed to the mathematical phenomenon known as the *blessing of dimensionality*, which ensures that in high-dimensional spaces, the number of quasi-orthogonal directions exponentially grows with *HD_DIM*. This means that, by increasing the dimensionality, the number of quasi-orthogonal vectors grows exponentially, even for a very tiny interval around 90 degrees. Refer to Fig. 1 for a visual depiction of this concept: an augmented dimensionality leads to a swift surge in HVs exhibiting similarity close to zero."

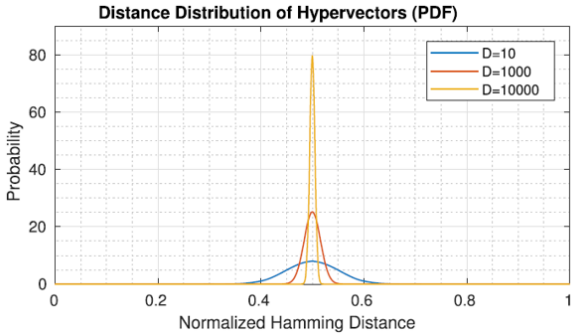


Figure 1: Caption

Consequently, by adopting this approach of random HV sampling, we are able to effectively represent dissimilar objects within the initial reference space through the utilization of orthogonal HVs. The memory containing these HVs is often called Item Memory (IM).

1.4.3 Scalar Values

When working with numeric scalar values, it is crucial to establish a mapping strategy that ensures close values in the original space are mapped in similar HVs, such that the similarity decreases with the increasing difference of scalar values. To do this, the literature highlights three primary techniques:

- Linear Mapping;
- Approximately Linear Mapping;
- Thermometer Encoding.

All these techniques start by quantizing the values in a predefined range $[Q_{min}, Q_{max}]$, using l quantization levels. Q_{min}, Q_{max} and l strongly depend on the target problem and should be accurately chosen by the designer analyzing the input data distribution.

1.4.3.1 Linear Mapping

In Linear Mapping encoding, the level Q_{min} is mapped to a random HV. Conversely, the remaining levels are generated using linear interpolation. For each level m, $\frac{d}{2}(m-1)$ bits are flipped randomly in comparison to the previous level. Importantly, this method inherently enforces orthogonality between the minimum and maximum levels, as they are situated $d/2$ bits apart. It's important to highlight that this method necessitates sequential generation of HVs, given that each level builds upon the previous one.

1.4.3.2 Approximately Linear Mapping

In Approximately Linear Mapping encoding, the level Q_{min} and Q_{max} are mapped in random HVs. Intermediate levels are generated by concatenating parts of the two HD vectors. Specifically, the level m will be represented by an HV obtained by combining

- $1 - l/m$ elements of HV_{Qmin};
- l/m elements of the HV_{Qmax}

Approximate linear mapping does not guarantee ideal linear characteristic of the mapping, but the overall decay of similarity between feature levels will be approximately linear. In contrast to the ideal linear mapping, values of similarity between neighbouring levels could slightly vary. Approximate linear mapping, however, requires the storage of only two random HV: one for the first level and one for the last level. HV vectors for intermediate levels are generated by the concatenation of parts of the two HD vectors.

1.4.3.3 Thermometer Encoding

In Thermometer Encoding, under the increasing value of the encoded quantity, more and more neighbouring elements are activated, like in the movement of

the thermometer column. The thermometer code for the minimum element has no '1's and the code for the maximum one has D/2 '1's. An intermediate-level HV, m, has the first: $\frac{m}{I}\frac{D}{2}$ elements set to '1'.

Table **??** summarizes all the differences, advantages and disadvantages of these techniques. The memory containing the HV for representing scalar values is often called Continuous item-memory (CIM)

1.4.4 Sets

In HDC, a set of symbols is usually represented by the superposition or bundling of the symbols HV.

1.4.5 Role-filler

Role-fillers or key-value pairs are a very general technique for representing structured data records. For example, the role (or key) can be the component's ID, while the filler (or value) can be the component's value. This type of data pair in HDC can be represented using binding or permutation. When using the binding, both the role and the filler are first transformed in HVs that are then bound together. When using permutation, instead, the filler is represented by an HV, while the role is associated with a certain permutation (that means a certain number of times that some fixed permutation is applied to the filler HVs).

1.4.6 Spatial Encoding

This section applies the ideas discussed so far to introduce two different techniques for encoding spatial data feature vectors. Suppose you want to encode the 4-features vector shown in Fig. 2. This vector belongs to the EMG dataset, where the stream of 4 EMG sensors is used to perform a hand gesture recognition task. The value of the EMG is comprised in the range [0, 21].

0,2	0,3	0,05	0
E1	E2	E3	E4

Figure 2: Example Feature vector in the EMG dataset

- The first step is to quantize each feature in a limited number of levels. Suppose that we want to use 20 levels.

- For each of these levels, as seen in section 1.4.3, a LevelHV is created using one of the discussed techniques (which one is not important in the following) and stored in memory. These HVs should preserve the similarity between similar values, as shown in Fig. 3.

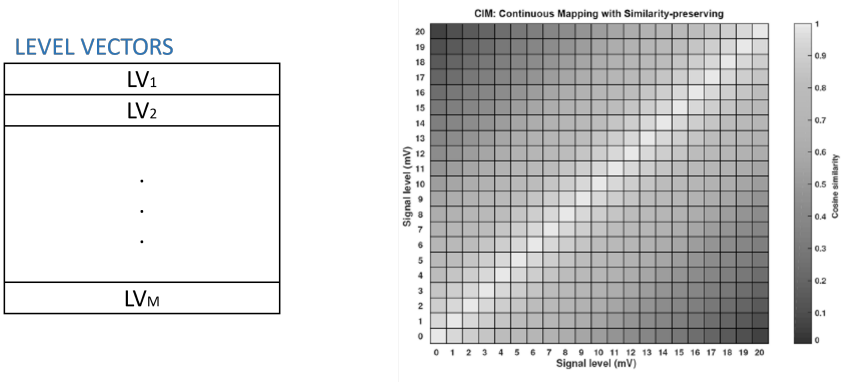


Figure 3: Caption

After these preliminary steps, the feature vector can be encoded using two main approaches: the record-based and the n-gram-based.

1.4.6.1 Record-Based encoding

In record-based encoding, a different HV, denoted as BaseVector (BV) or position HV, is created for representing the feature ID (or position) in the feature vector (i.e. the individual electrodes: E1, E2, E3, E4). Since IDs are independent symbols, we randomly generate these HVs, ensuring they are orthogonal, Fig. 4. Then, the encoding is performed as follows:

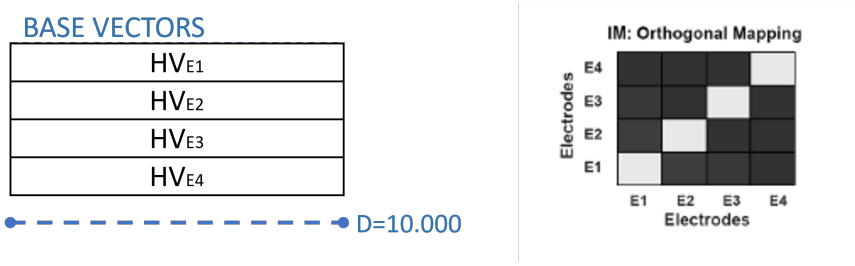
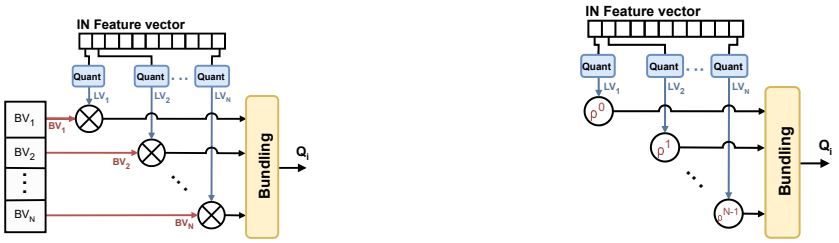


Figure 4: Base Vectors

- For each feature, a LevelHV is selected according to the input value;
- The LevelHV is uniquely associated with the featureID by the binding operation with the corresponding BaseVector;



(a) Record-based encoding for a N-element feature vector

(b) N-gram-based encoding for an N-element feature vector

Figure 5: Record-based and N-gram-based techniques, adopted for encoding the input feature vectors in HV

- All the binded couples are superimposed to create the final HV representing the whole feature vector.

This procedure is summarized in eq. 3 and figure **??**.

$$S_t = [(E1 * L_{1t}) + (E2 * L_{2t}) + (E3 * L_{3t}) + (E4 * L_{4t})] \tag{3}$$

1.4.6.2 N-gram based encoding

In the N-gram-based encoding, we do not employ base vectors. The feature positions are encoded through permutations, leveraging the inherent property of this operation to produce orthogonal HVs. This modified encoding scheme is represented in equation 4, visually illustrated in Fig. **??**, and its description is as follows:

- For each feature, a LevelHV is selected according to the input value;
- The LevelHV is uniquely associated to the featureID by permuting it by a number of time equal to the feature position. So 0 permutation for the first, 1 for the second, etc.;
- The results are superimposed to create the final HV representing the whole feature vector.

$$S_t = (\rho^0(L_{1t}) + \rho^1(L_{2t}) + \rho^2(L_{3t}) + \rho^3(L_{4t}) \tag{4}$$

1.4.7 Temporal Encoding in Sequential Data Processing

In sequential data processing, the model is tasked with handling not just isolated patterns but also temporal sequences. This necessitates operating

within time windows of varying sizes to capture temporal dependencies effectively. The encoding process aims to generate a unique Hyperdimensional Vector (HV) for each time window, employing the permutation operation as a key technique.

Initially, the process involves encoding spatial sequences into HVs, according to one of the previously discussed techniques. These HVs are then subject to the permutation operation to integrate the temporal dimension. This is accomplished by permuting each input HV a number of times corresponding to its position within the time window. For example, the HV for the first time instant is permuted zero times, the second one time, and so on. This methodology ensures that each time instant within the window is distinctively and accurately encoded, facilitating a detailed representation of temporal patterns in the data.

Imagine a scenario where we analyze weather patterns over a period. Each day's weather data, comprising spatial feature vectors like temperature, humidity, and wind speed, is encoded into an HV. For a simplified example, consider a week's weather data. The HV for Monday (Day 1), with its specific temperature, humidity, and wind speed values, undergoes zero permutations. The HV for Tuesday (Day 2) is permuted once to reflect its position on the second day, and so forth.

1.5 Training

The training procedure of an HDC model is shown in (5) and Fig.6 consists of encoding each input feature vector in the training set using one of the previously explained techniques and bundling all the HVs belonging to the same prediction class (label) to build the ClassHVs (CVs) prototypes. In a K-class classification problem, K CVs will be generated during the learning phase and stored in an Associative Memory (AM).

$$CV_j = \sum_{\forall i \in class_j} (H_i) \tag{5}$$

During this process, the bundling produces CVs with integer elements, even if the initial HVs were not. Clipping techniques can be applied to adjust the data in different ranges, such as: binary, bipolar, tripolar, power of two or quantized. For instance, for obtaining bipolar HVs, the sign function is applied to each element, while for binary HVs, the majority rule shown in (6) is used, where $CV_{k,d}$ is the d^{th} dimension of the CV_k. This makes a trade-off between accuracy and hardware complexity.

$$CV_{k,d} = \begin{cases} 0 & CV_{k,d} < \frac{\text{n.bundled HVs}}{2} \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

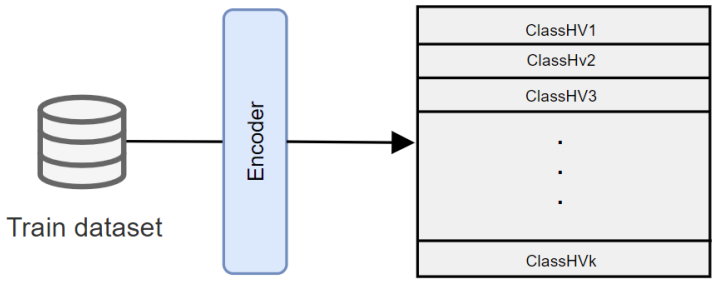


Figure 6: Train procedure in HDC

1.6 Inference

The inference phase in HDC is depicted in 7 and consists of encoding each input feature vector into a query HV, denoted as *Q*, using the same encoding module as in the training phase. The classification task is then executed by searching for the most similar CV within the associative memory. This similarity search entails seeking the minimum value when utilizing the cosine distance as the similarity measure or searching for the maximum otherwise.

It's important to note that while the encoding step typically consumes the most time during the training phase, the inference phase often places a heavier computational load on the associative search, accounting for up to 83% of the total execution time.

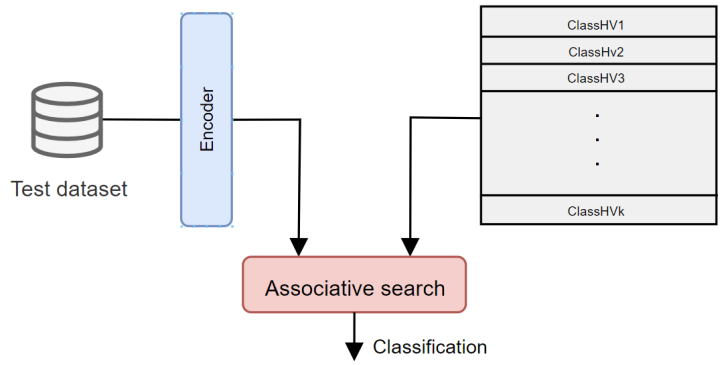


Figure 7: Inference procedure in HDC

1.7 Retraining

To improve the HDC model's accuracy, after the initial training step, an iterative process of *retrain* can be performed. During this stage, the estimated CVs are calibrated using new data or by repeating the training for multiple iterations

on the same dataset. Retraining is performed by removing the mispredicted query HVs from the selected CVs and adding them to the correct ones. The class update is performed as shown (7), where Q_j is the query HV, C_{wrong} is the mispredicted class and $C_{correct}$ is the correct one. Note that, in the case of retraining, the clipping is only applied in the last step.

$$\begin{cases} C_{wrong} = C_{wrong} - Q_j \\ C_{correct} = C_{correct} + Q_j \end{cases} \tag{7}$$

However, when performing retraining, a lack of control in the CV update can easily lead the model to slow convergence or divergence [**?**]. In [**?**], the widely known concept of learning rate in AI is extended to HDC, proposing two different adaptive training techniques: iteration-dependent and data-dependent. In the iteration-dependent approach, the learning rate linearly decays as a function of the error rate. In the data-dependent approach, the adopted learning rate depends on how far the data was misclassified.