AENEAS
HYPERCOMPUTE

Reference Manual

Version 1.0.0

Marco Angioli[1],
Saeid Jamili[2]

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

Contents

1 The Aeneas Hypercompute framework

1.1 Motivations

In recent years, Hyperdimensional Computing (HDC) has gained traction as a promising alternative for executing learning tasks on resource-constrained edge devices, owing to its inherent efficiency and robustness (check Section 4). HDC processes are intrinsically simpler, facilitating straightforward implementations. Despite this simplicity, today the literature offers various design choices that can be adopted to enhance model accuracy or reduce its complexity. However, while the impact of these configurations on accuracy has been extensively studied, in the existing literature, there is a notable gap concerning how different choices impact power consumption, area occupation, and execution time when translated into hardware. As a result, hardware implementations frequently rely on ad-hoc decisions inspired by state-of-the-art approaches without a proper justification or consideration of the specific problem and the available resources, possibly resulting in heavy suboptimal solutions. To address these challenges and enhance the deployment of HDC models, we present AeneasHDC—an automated, open-source platform that facilitates the easy and efficient deployment of HDC models across both software and hardware environments, enabling systematic exploration and assessment of design choices.

1.2 Model Overview

AeneasHDC is an open-source framework that offers a streamlined and automatic workflow for instantiating, managing, and evaluating Hyperdimensional Computing models, both in software and hardware. The environment is extremely customizable, supporting a wide range of the most common techniques adopted in the literature for HDC learning models, enabling users to easily assess the impact of different design choices on model accuracy, memory usage, execution time, energy consumption, and area requirements.

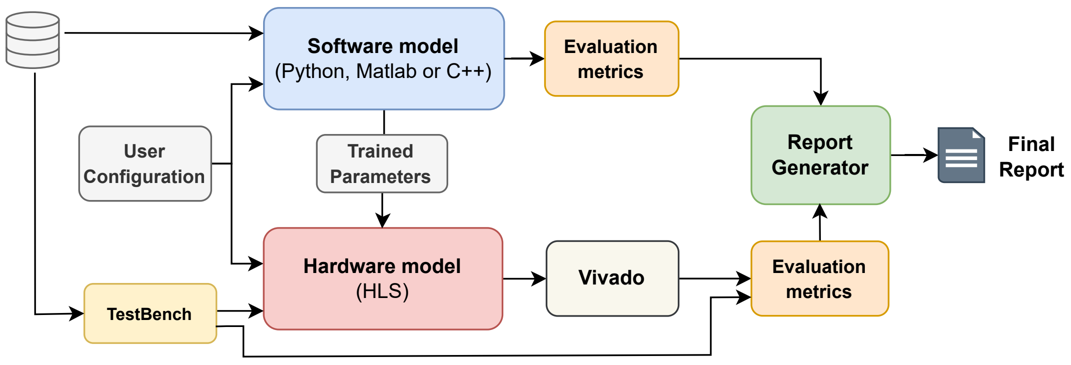


Figure 1: Overview of the Aeneas Hypercompute framework

Figure 1 illustrates AeneasHDC's workflow. The user can specify:

- The problem characteristics, including dataset details, the number of features and classes, the desired programming language, and the target FPGA. how to distribute the workload between the software and the hardware implementations.
- Details about the desired hardware accelerator, such as the required hardware parallelism at different stages of the learning task.
- How to distribute the workload between the software and the hardware implementations. The hardware can be generated to accelerate the entire classification process (train, inference, retrain) or just some tasks, deciding, for example, to execute the train and retrain in software and instantiate only the hardware for the inference, as commonly done for embedded applications.

The system automatically generates all the configuration files required by the following entities and instantiates the model in software and hardware. Thanks to the versatile AeneasHDC library, the software model can be produced in Python, Matlab, or C++. The model is executed, and the results in terms of accuracy and other performance metrics are collected. The data required for inference are extracted and passed to the hardware model.

The hardware architecture is generated using the Xilinx Vitis High-Level Synthesis (HLS) tool and then synthesized, simulated and tested using Vivado 2023. The data about energy consumption, execution time and resource allocation are collected from the design.

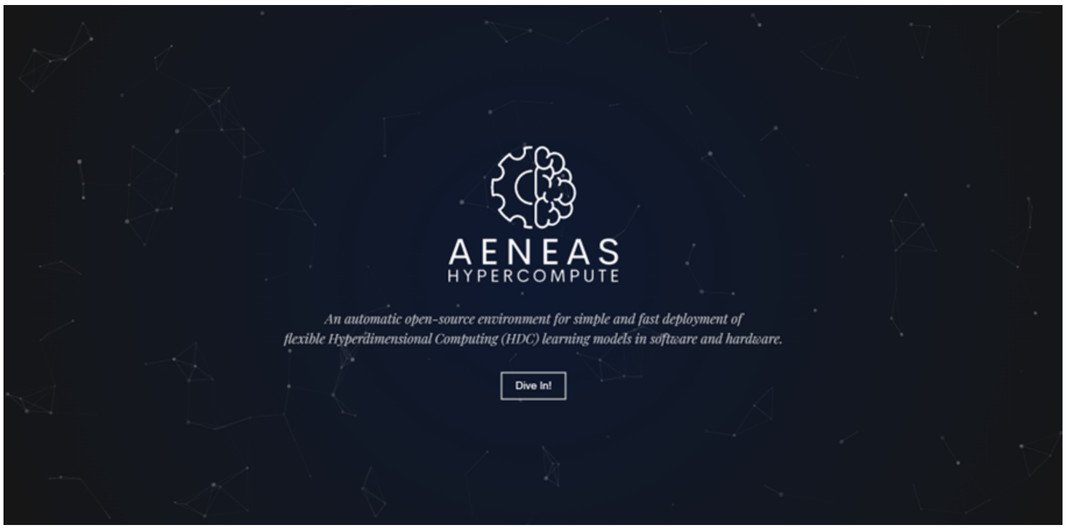
All the results obtained from the software and the hardware implementations are finally relayed to a dedicated report-generation module that compiles a comprehensive HTML-based summary, encapsulating all the information from the run simulations. The summary includes details of the model configuration, dataset specifications, chosen programming language, target hardware platform, and key performance metrics such as execution time, memory usage, energy consumption, and resource utilization. This report is stored locally and can be readily accessible through the user-friendly GUI, which organizes all conducted tests and allows to review and compare different runs. The entire process is engineered to be fully automated, abstracting complex or time-consuming tasks associated with coding models or designing custom hardware architectures. The GUI is an efficient singular point of control that enables users to manage the model at every stage. However, AeneasHDC is completely open-source and makes all its libraries and scripts accessible. Extensive documentation is provided, covering the basic usage of the model, the theoretical concepts behind parameter selections, and detailed insights into the underlying code and library functions.

The AeneasHDC environment is always in evolution and we are constantly adding new models and techniques, aiming to keep up with all the innovations in the literature However, the model is completely open-source, and the user can easily extend the software and hardware libraries to include new desired models. The detailed how-to-use manual is on GitHub. We always encourage community collaboration to enhance and evolve the framework over time. If you have any questions or need help, please contact the support team.

1.3 Graphical User Interface (GUI)

The Aeneas HyperCompute Platform offers a robust, user-friendly interface designed to simplify the deployment of Hyperdimensional Computing (HDC) models across both software and hardware environments. The platform is built to accommodate researchers and developers looking to rapidly prototype and deploy flexible HDC systems in hardware.

The landing page in Fig. 2 presents a clean and minimalistic design centred around easy navigation of all the platform's features. Users are greeted with an overview of the platform's capabilities and an inviting "Dive In!" button to start their project configuration immediately.

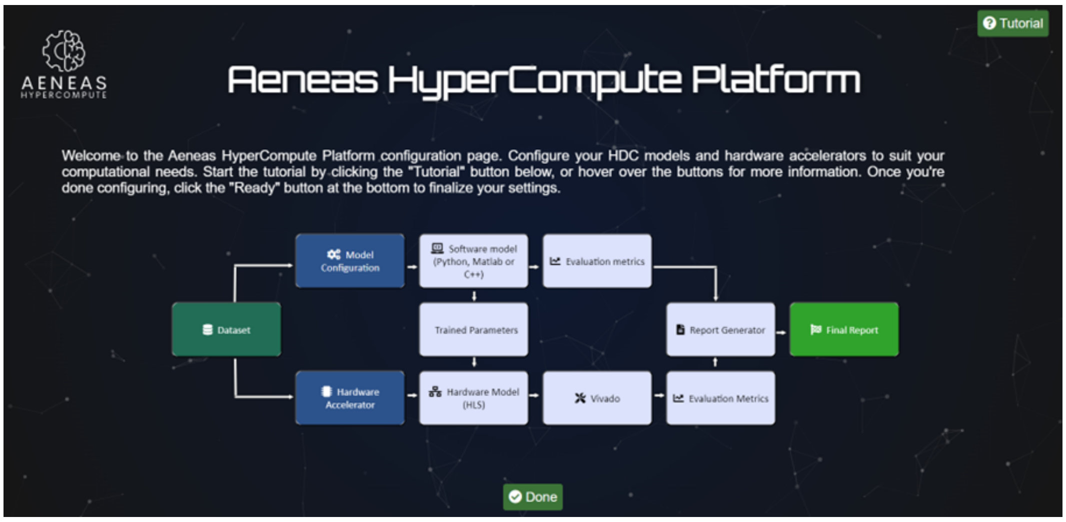


An automatic open-source environment for simple and fast deployment of flexible Hyperdimensional Computing (HDC) learning models in software and hardware.

Figure 2: GUI: Landing Page

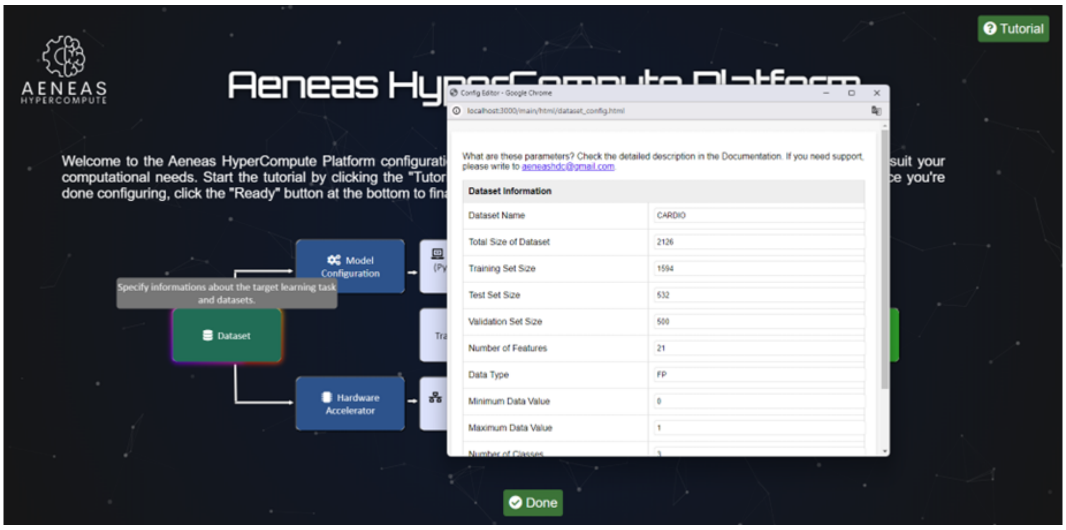
The Configuration Dashboard in Fig. 4 and 3 serves as the central hub where users can define the specifics of their HDC model. The dashboard allows users to configure various aspects of their HDC model. Users can define the dataset's characteristics, specify the learning task, and make desired design choices for the HDC model. They can also select the software implementation language, choosing from Python, Matlab, or C++. Additionally, users can specify the target FPGA and select the target learning phase for hardware acceleration, whether it be training, inference, retraining, or all phases. Finally, the dashboard provides options to configure the desired level of hardware parallelism.

The intuitive layout ensures all settings are accessible in a logical, streamlined manner, facilitating a seamless configuration process. For more granular control over the configuration settings, modal windows provide detailed options for each component of the HDC model.



Welcome to the Aeneas HyperCompute Platform configuration page. Configure your HDC models and hardware accelerators to suit your computational needs. Start the tutorial by clicking the "Tutorial" button below, or hover over the buttons for more information. Once you're done configuring, click the "Ready" button at the bottom to finalize your settings.

Figure 3: GUI: Configuration Page

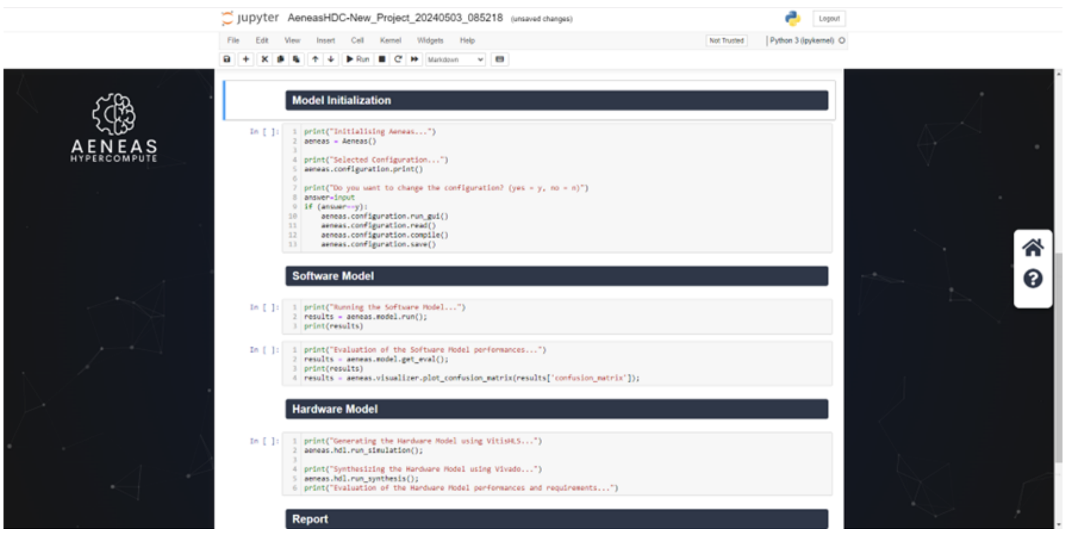

Tutorial
AENEAS HYPERCOMPUTE
Aeneas HyperCompute Platform
Config Editor - Google Chrome
localhost:3000/main/html/dataset_config.html
Specify informations about the target learning task and datasets.
Model Configuration
Dataset
Hardware Accelerator
Done

What are these parameters? Check the detailed description in the Documentation. If you need support, please write to aeneashdc@gmail.com.

Dataset Information	
Dataset Name	CARDIO
Total Size of Dataset	2126
Training Set Size	1594
Test Set Size	532
Validation Set Size	500
Number of Features	21
Data Type	FP
Minimum Data Value	0
Maximum Data Value	1
Number of Classes	3

Figure 4: GUI: Configuration Page, example

Once the model is configured, the Jupyter Notebook in Fig.5 is created, including all the functions to compile and execute the specified model in software and hardware. These notebooks also provide a detailed log of the execution, including performance metrics and operational logs, facilitating debugging and performance optimization. The platform includes a project dashboard for managing multiple HDC projects, offering options to open, rename, copy, or delete projects. Additionally, a dedicated reporting interface compiles and displays performance metrics and other crucial data from each run, enhancing transparency and traceability.


jupyter AeneasHDC-New_Project_20240503_085218 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3 (ipykernel)
Run
Markdown
AENEAS HYPERCOMPUTE

Model Initialization

```
In [ ]: print("Initialising Aeneas...")
        aeneas = Aeneas()

        print("Selected Configuration...")
        aeneas.configuration.print()

        print("Do you want to change the configuration? (yes = y, no = n)")
        answer=input
        if (answer==y):
            aeneas.configuration.run_gui()
            aeneas.configuration.read()
            aeneas.configuration.compile()
            aeneas.configuration.save()
```

Software Model

```
In [ ]: print("Running the Software Model...")
        results = aeneas.model.run();
        print(results)
```

```
In [ ]: print("Evaluation of the Software Model performances...")
        results = aeneas.model.get_eval();
        print(results)
        results = aeneas.visualizer.plot_confusion_matrix(results['confusion_matrix']);
```

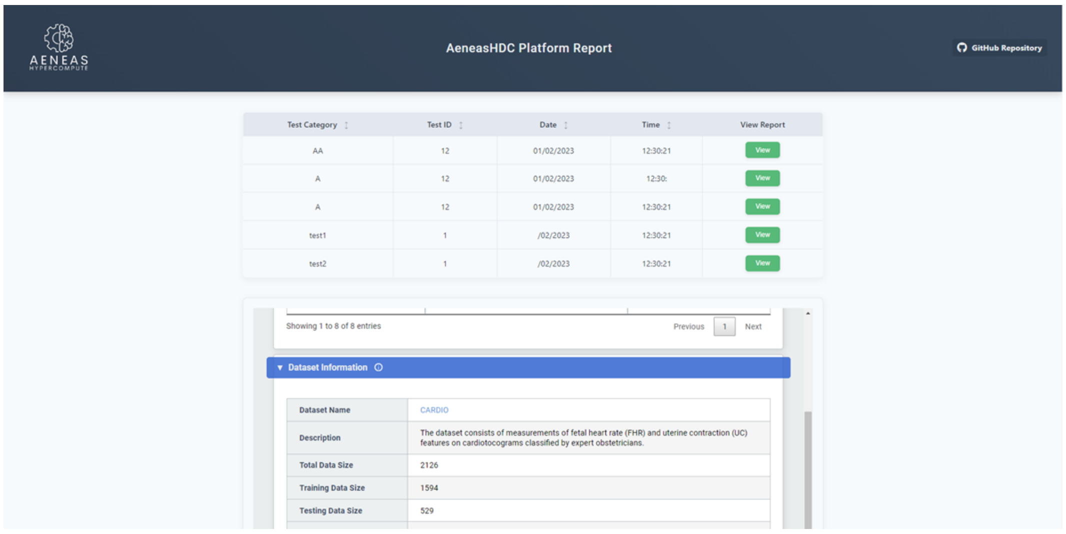
Hardware Model

```
In [ ]: print("Generating the Hardware Model using VitisHLS...")
        aeneas.hdl.run_simulation();

        print("Synthesizing the Hardware Model using Vivado...")
        aeneas.hdl.run_synthesis();
        print("Evaluation of the Hardware Model performances and requirements...")
```

Report

Figure 5: GUI: Jupyter Notebook



AeneasHDC Platform Report

Test Category	Test ID	Date	Time	View Report
AA	12	01/02/2023	12:30:21	View
A	12	01/02/2023	12:30:	View
A	12	01/02/2023	12:30:21	View
test1	1	/02/2023	12:30:21	View
test2	1	/02/2023	12:30:21	View

Showing 1 to 8 of 8 entries   Previous 1 Next

▼ Dataset Information

Dataset Name	CARDIO
Description	The dataset consists of measurements of fetal heart rate (FHR) and uterine contraction (UC) features on cardiotocograms classified by expert obstetricians.
Total Data Size	2126
Training Data Size	1594
Testing Data Size	529

Figure 6: GUI: Report Control Page

The Aeneas HyperCompute Platform’s GUI is crafted to ensure that users can focus more on exploring the impact of different design choices for HDC models and less on the intricacies of custom hardware design.

1.4 Software Libraries

AeneasHDC provides a comprehensive range of configuration options for developing the HDC model, encompassing most techniques commonly found in the literature. This is possible thanks to an extensive software library fully compatible with Python, C++, and Matlab.

Table 1 lists all the currently supported features. The hyperspace can be configured to support binary, bipolar, tripolar and integer HVs in any dimension and density. This allows to evaluate the impact of HV size and sparsity on computational complexity and model accuracy. Three techniques are available for generating LevelVectors and representing scalar numbers: Linear, Approximately Linear and Thermometer. The Linear method offers the highest accuracy at the expense of high memory usage. Conversely, the Approximately Linear and Thermometer methods reduce memory needs and power consumption, respectively. We offer support for both spatial and temporal sequence encoding. For spatial encoding, users can choose from two distinct techniques: record-based and n-gram-based, allowing customization to specific problem requirements. For temporal encoding, the framework provides the option to use custom window sizes or n-gram configurations, enhancing its suitability for a diverse range of challenges. Clipping techniques can be optionally applied post-encoding, post-training, or at both

Table 1: Explorable Design Space in the software library

Parameters	**Explorable space**
Learning Problem	[Classification, Clustering, Regression]
HV Dimension	Any, ex:[200, 500, 1.000, 2.000, 10.000]
HV DataType	[Binary, Bipolar, Tripolar, Integers]
HV Density	[Dense, Sparse]
HV Sparsity	Any, ex: [0.01, 0.02, 0.05, 0.1, 0.2]
Similarity Measure	[Hamming, Cosine, Dot Product]
LevelHVs Mode	[Linear, Approx. Linear, Thermometer]
Number of LevelHVs	Any, ex: [5, 10, 30, 50, ...]
Spatial Encoding Mode	[Record-based, N-gram based]
Temporal Encoding Mode	[Yes, No]
Temporal Encoding Window	Any, ex: [2,3,5]
Clipping Technique	[Binary, Ternary, PowerOfTwo, Quantized, Integer]
Retraining	[Yes, No]
Retrain Epochs	Any, ex: [5, 10, 20, 50]
Learning Rate Mode	[Data-dependent, Iteration Dependent]

stages. These include standard binary, bipolar, and ternary clipping, along with quantized formats optimized for hardware implementations. Of particular note is the power-of-two quantization, which allows multiplication and division operations required for encoding and cosine similarity to be replaced by simpler arithmetic shifts in hardware. Our platform facilitates retraining, supporting multiple epochs and state-of-the-art learning rate strategies. It can be conducted on the same training dataset, as is common in literature, or on any tailored one. Finally, AeneasHDC currently supports clustering, regression and classification problems, but it can be easily extended to other applications.

All the details about the Python library functions and their input parameters are described in section 2.

1.5 Flexibility in the Hardware Architectures

In addition to the software library, we also offer a comprehensive hardware library designed for High-Level Synthesis (HLS). The AeneasHDC platform is designed to transition from software-defined models to practical hardware implementations seamlessly. All the configurations discussed in the previous Section can be implemented in hardware, on any target FPGA, empowering users to directly assess how different design choices impact hardware performance and efficiency.

The accelerators synthesized and implemented via AeneasHDC feature a standard AXI (Advanced eXtensible Interface) connection with the processor and the dedicated memories that ensures straightforward integration and broad compatibility. In these designs, the encoding process follows a pipeline approach that enables simultaneous processing of various input elements at different stages, optimizing hardware resource utilization, operating frequency, and throughput. Additionally, our framework introduces two pivotal features that add significant flexibility to the hardware architectures, ensuring efficient solutions in diverse environments.

1. **Selective Model Instantiation:** synthesis of specific operational stages in the HDC learning process. For example, the architecture can be finely tuned for accelerating the entire process on high-resource FPGAs or for performing only the inference on resource-constrained systems;

2. **Partial Processing in the Aeneas framework** The inherent parallelism in HDC classification problems naturally lends itself to hardware implementations. Each element within the HVs operates independently from the others, allowing for a potential full parallel processing. However, HDC is often utilized in embedded systems and edge devices, where

resource constraints such as limited area and power consumption must be considered.

The Aeneas framework addresses this challenge by introducing additional flexibility to hardware implementations. This is achieved through a *partial processing* mechanism, which empowers users to define the degree of hardware parallelism at various stages of the learning task, allowing trade-offs between processing speed, hardware requirements and power consumption. This control can be applied at three different levels, summarized in Table 2.

(a) **Element parallelism:** the *FRAME* parameter controls the number of HV elements processed in parallel.In Figure 7, we illustrate an example of *FRAME* parallelism applied to a record-based encoding technique with four features and binary HVs. As highlighted, all the hardware required to produce one HV's element is replicated *FRAME* times. This approach represents a compromise between processing one element at a time (as in a sequential approach) and full parallel processing, offering the flexibility to tailor hardware parameters to meet specific problem requirements.

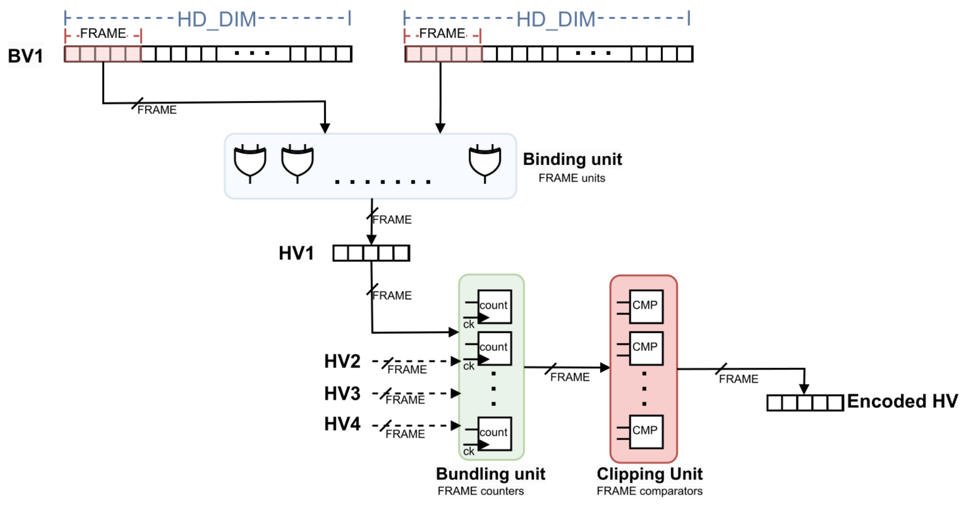


Figure 7: *FRAME* parallelism. *FRAME* elements are processed per time

As shown in Figure 7, each iteration processes *FRAME* elements of the hypervector. *FRAME* elements are taken from the base vector (BV) and the corresponding level vector (LV), and then they are binded, bundled, and clipped. These operations are performed by dedicated hardware units that, in which, thanks to the partial processing only FRAME functional units are instantiated, rather than

Table 2: Hardware Parallelism Degrees

Parameters	**Explorable space**
Element Parallelism (*FRAME*)	Any in range $[1, D]$
Feature Parallelism	Any in range $[1, N]$
Class Parallelism	Any in range $[1, K]$

HV_size. The total number of iterations is equal to the hypervector size divided by the FRAME size. So, the trade-off between FRAME size and the number of iterations is as follows:

- A smaller FRAME size requires less hardware resources and less energy consumption, but it increases the number of iterations;
- A larger FRAME size requires more hardware resources, but it reduces the execution time.

The optimal FRAME size depends on the specific application and the available hardware resources. This level of flexibility empowers users to make informed trade-offs between execution speed and hardware resource usage, adapting the framework to their specific application requirements.

(b) **Feature Parallelism:** within the Aeneas framework, users have the ability to determine the concurrent processing of features. While the inherent parallelism of HDC could theoretically replicate processing logic for all features simultaneously, practical constraints arise, especially in embedded environments dealing with datasets containing numerous features (e.g. ISOLET's 617 features).

In Aeneas, users are granted the flexibility to process features in multiple ways: sequentially, one at a time, or in parallel. They can specify any desired number within the $[1, N]$ range. Each concurrently processed feature handles FRAME elements at a time.

(c) **Class Parallelism:** reconfigurable hardware parallelism for the similarity search depicted in Fig. 8. The user can decide to compare the encoded HV with each class sequentially or to enable parallel computation of $k \in [1, K]$ classes. This is instrumental in elaborating complex multi-class datasets and can be combined with the previous two levels.

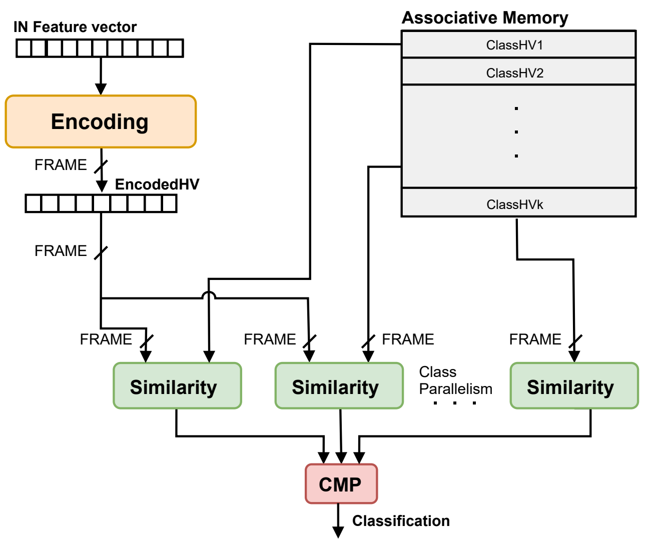


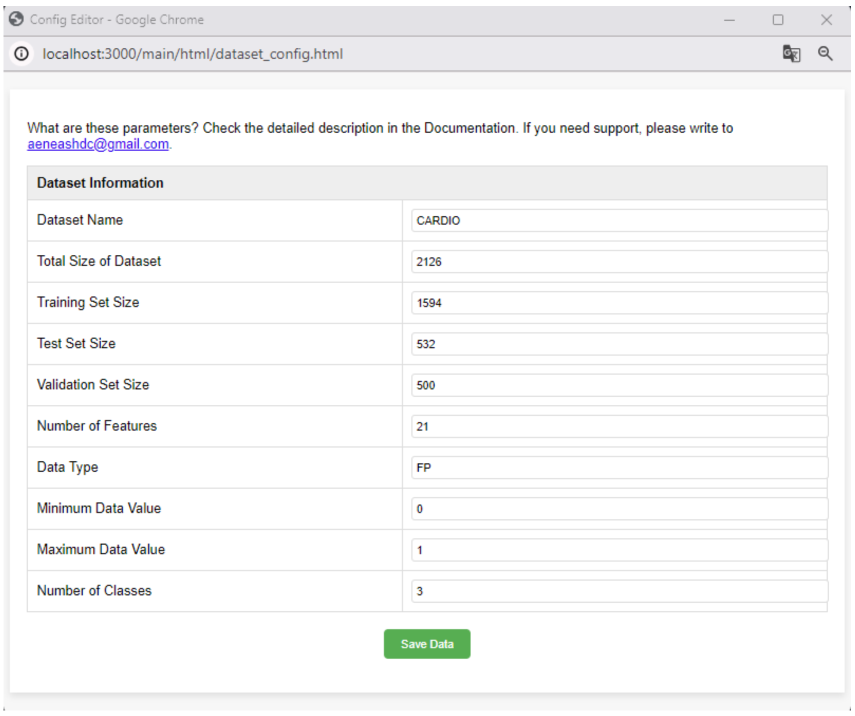
Figure 8: Class Parallelism applied to a Record-based encoding

1.6 Creating Your First Project

This step-by-step guide illustrates the process of setting up a new project on the AeneasHDC Platform. In this example, we will consider the generation of an hardware accelerator for the inference on the CARDIO dataset

1. Begin by navigating to the home page of the Aeneas HyperCompute Platform. From there, select the "Create a New Project" option. This action opens the configuration dashboard, where you will set the parameters for your project.

2. Once inside the configuration dashboard, proceed with the following steps. Configuring your project involves specifying the dataset's characteristics, model parameters, and hardware acceleration settings. For this project, the CARDIO dataset is selected. The dataset includes 2126 total data samples, with 1594 designated for training and 529 for testing. Each sample consists of 21 features, ranging from a minimum of 0 to a maximum of 1. The dataset is categorized into three distinct classes. These options are set as shown in Fig. 9.


Config Editor - Google Chrome
localhost:3000/main/html/dataset_config.html
Save Data

What are these parameters? Check the detailed description in the Documentation. If you need support, please write to aeneashdc@gmail.com.

Dataset Information	
Dataset Name	CARDIO
Total Size of Dataset	2126
Training Set Size	1594
Test Set Size	532
Validation Set Size	500
Number of Features	21
Data Type	FP
Minimum Data Value	0
Maximum Data Value	1
Number of Classes	3

Figure 9: Enter Caption

The model is configured to utilize binary hyperdimensional vectors (HVs). These vectors are defined with 256 elements, using 20 quantization levels applied through a linear technique. A record-based encoding strategy is adopted. Hardware acceleration is crucial for enhancing the model's performance, especially in real-time applications. In this example, the hardware is configured to accelerate the inference phase by processing 64 HV elements in parallel, optimizing both speed and efficiency.

3. Upon completing the setup, you can click on the "Done" button. All the configuration files are generated (you can find these files in the folder *AeneasHDC/src/config*). The platform creates and opens a Jupyter Notebook that is saved in the folder *AeneasHDC/projects*, and it is pre-populated with the necessary code to control AeneasHDC and compile and execute both the software and the hardware models. First, the software model is run in Python, and the performance results are extracted. Then, if we want to perform only the inference in hardware (as in this example), the trained HVs are extracted and sent to the hardware model. This model is generated using Vitis HLS and then simulated, synthesized and implemented on Vivado. Data on resource usage, energy consumption, execution time and accuracy are collected.

4. The final stage involves the generation of a comprehensive report that encapsulates all relevant data from the run simulations. This report includes detailed sections on model configuration, dataset specifications, and key performance metrics such as execution times, memory usage, energy consumption, and resource utilization. The report, stored locally, is easily accessible through the GUI for review and comparison against other test runs.

2 Library Documentation

AeneasHDC offers a rich array of configuration options for crafting HDC classification models. The python class encompasses versatile settings for the HV size, type, and density, allowing for fine-tuned hyperspace characteristics. Multiple encoding techniques for classification tasks are provided, including record-based, n-gram, and random-projection methods. LevelVectors can be generated using Linear, Approximately Linear, or Thermometer techniques. Moreover, various post-training clippings can be applied to the resulting ClassHVs, including options to make them binary, ternary, powers of two, and quantized. Additionally, AeneasHDC facilitates retraining, supporting state-of-the-art (SOA) learning rate strategies. Finally, users can choose from different similarity measures in the inference stage.

2.1 Class Methods

All features and settings of the AeneasHDC model are specified in the ‘config.py‘ file. This file determines the behavior of the class methods outlined below. Ensure that the ‘config.py‘ file is configured to your requirements before using these methods.

2.1.1 generate_LevelVector

Description: Generate the LevelHVs used for encoding the scalar value of a feature.
Configurable Parameters: HD_LV_TYPE in config.py file.

```
model.generate_LevelVector()
```

Attribute	Description
Dimensionality	Size of the HV.
HV_type	HV type (e.g., bipolar, binary).
density	HVs density (e.g., dense or sparse).
sparsity_factor	HV percentage sparsity (number of ones)
Number_of_features	Number of features in the dataset.
Number_of_levels	Number of levels for encoding.
num_classes	Number of classes in the dataset.
LevelTechnique	Technique used for level encoding.
similarity	Adopted Similarity measure (e.g., Hamming distance).
encoding_technique	Spatial encoding technique
n_gram	Temporal encoding (0: disabled, 1: enabled)
n_gram_size	Size of the n-grams.
clipping_encoding	Clipping technique used for encoding.
clipping_class	Clipping technique used for classes.
quantization_range	Range for quantization (min, max).
epochs	Number of retraining epochs.
lr_max	Maximum learning rate.
lr_decay	Learning rate decay method.
beta_lr	Beta learning rate.

Table 3: Attributes of the HDC Class

2.1.2 Similarity

Description: Computes the similarity between two hypervectors.
Configurable Parameters: HD_SIMI_METHOD in config.py file.
Input Parameters:

- HV1, HV2: Hypervectors whose similarity is to be computed.

```
model.similarity(HV1, HV2)
```

2.1.3 Bind

Description: Performs the binding operation between two hypervectors.
Configurable Parameters: None specific in config.py for this method. The arithmetic implementation of this operation depends by the HD_DATA_TYPE
Input Parameters:

- HV1, HV2: Hypervectors to be bound together.

```
model.bind(HV1, HV2)
```

2.1.4 Permutation

Description: Applies a circular shift (permutation) to a hypervector.
Configurable Parameters: None specific in config.py for this method.
Input Parameters:

- HV: The hypervector to be permuted.
- positions: Number of positions to shift.

```
model.permutation(HV, positions)
```

2.1.5 Bundle

Description: Combines two hypervectors using the bundling operation.
Configurable Parameters: None specific in config.py for this method. The arithmetic implementation of this operation depends by the HD_DATA_TYPE.
Input Parameters:

- HV1, HV2: Hypervectors to be bundled.

```
model.bundle(HV1, HV2)
```

2.1.6 Clip

Description: Performs clipping on a hypervector, supporting various clipping types.
Configurable Parameters: CLIPPING_TYPE in config.py file.
Input Parameters:

- HV1: The hypervector to be clipped.
- min_val, max_val: Range for quantized clipping.
- clipping_type: Type of clipping to be applied.
- bundled_HVs: Number of hypervectors bundled before clipping.

```
model.clip(HV1, min_val, max_val, clipping_type, bundled_HVs)
```

2.1.7 Context_dependent_thinning

Description: Applies Context-Dependent Thinning (CDT) to a hypervector.
Configurable Parameters: None. Always performed when HD_MODE=SPARSE.

Input Parameters:

- Z: The hypervector to be thinned.
- bundled_HVs: Number of hypervectors bundled.
- thinning_steps: Number of steps for the thinning process.

```
model.context_dependent_thinning(Z, bundled_HVs, thinning_steps)
```

2.1.8 Encoding

Description: Encodes a feature vector into a hypervector using selected encoding techniques. It performs both the temporal encoding and the spatial one.
Configurable Parameters: ENCODING_TECHNIQUE, N_GRAM, N_GRAM_SIZE
Input Parameters:

- feature_vector: The feature vector to be encoded.
- BaseHVs: Base Hypervectors for encoding.
- LevelHVs: Level Hypervectors for encoding.
- quant_levels: Quantization levels for feature values.
- clipping_type: Clipping type to be used post-encoding.
- encoding_technique: The encoding technique to be used.

```
model.encoding(feature_vector, BaseHVs, LevelHVs, quant_levels,
   clipping_type, encoding_technique)
```

2.1.9 train

Description: Trains the HDC model using training data and labels.
Configurable Parameters: N_GRAM, DS_TRAIN_SIZE, CLIPPING_CLASS.
Input Parameters:

- train_data: Training data array.
- train_labels: Corresponding labels for the training data.
- verbose: Verbosity level.
- ds_min_value: Minimum value in the dataset for quantization.
- ds_max_value: Maximum value in the dataset for quantization.
- clip_class: Clipping type for classes.

```
model.train(train_data, train_labels, verbose, ds_min_value,
   ds_max_value, clip_class=config.CLIPPING_CLASS)
```

2.1.10 predict

Description: Makes predictions on test data using the trained HDC model.
Configurable Parameters: N_GRAM, DS_TEST_SIZE.
Input Parameters:

- test_data: Test data array.
- test_labels: Corresponding labels for the test data.
- BaseHVs: Base Hypervectors.
- LevelHVs: Level Hypervectors.
- ClassHVs: Class Hypervectors.
- quant_levels: Quantization levels.

```
model.predict(test_data, test_labels, BaseHVs, LevelHVs, ClassHVs,
   quant_levels, verbose=1)
```

2.1.11 retrain

Description: Retrains the HDC model using additional data to improve accuracy. The method adapts the model to better fit the data by adjusting the Class Hypervectors based on retraining data and labels.
Configurable Parameters: epochs, lr_max, lr_decay.
Input Parameters:

- retrain_data: Retraining data array.
- retrain_labels: Corresponding labels for the retraining data.
- test_data: Test data array for validation.
- test_label: Corresponding labels for the test data.
- BaseHVs: Base Hypervectors.
- LevelHVs: Level Hypervectors.
- ClassHVs: Class Hypervectors.
- quant_levels: Quantization levels.
- starting_accuracy: Initial accuracy before retraining.

```
model.retrain(retrain_data, retrain_labels, test_data, test_label,
   BaseHVs, LevelHVs, ClassHVs, quant_levels, verbose=1,
   starting_accuracy=0)
```

3 Test Environment

AeneasHDC features a set of pre-defined tests designed to demonstrate basic usage of the platform and explore the impact of some common design choices. Each test outputs a report including metrics like resource utilization, accuracy, processing speed, and other key performance indicators. Table 4 summarizes all the actual implemented tests.

Table 4: Overview of Test Functions

Test Name	Tested Feature
Test 1	HV Size
Test 2	HV Type
Test 3	Frame Parallelism
Test 4	Feature Parallelism
Test 5	Class Parallelism
Test 6	LV Technique
Test 7	Spatial Encoding mode
Test 8	Temporal Encoding N-Gram Size
Test 9	Clipping Technique
Test 10	Training on Hardware
Test 12	Combined Parallelism Effects
Test 13	Clipping and Similarity Combinations
Test 15	Complete test on the CARDIO dataset (Paper)
Test 16	Complete test on the EMG dataset (Paper)
Test 17	Complete test on the HEPTA dataset (Paper)
Test 18	Complete test on the Boston dataset (Paper)

Test 1: HV Size

Purpose: Assesses the impact of different HV sizes.
What to expect?: When increasing the HV size we expect the accuracy to grow. Additionally, if we keep the hardware parallelism constant (such as in this test), we won't see differences in resource usage and energy consumption. However, bigger sizes will require more execution time.
Options: Varying HV sizes, such as 512, 1024, 2048.

```
test_hd_dim(aeneas, hd_dim_range=[512, 1024, 2048])
```

Test 2: HV Type

Purpose: Analyzes performance with different HV types.
What to expect?: No difference in the hardware requirements because bipolar HVs can be efficiently treated as binary in hw. Note: this is true only if we use cosine similarity for both models.
Options: 'BINARY' (0) and 'BIPOLAR' (1).

```
test_hd_type(aeneas, hd_type_options=['BINARY', 'BIPOLAR'])
```

Test 3: Element Parallelism

Purpose: Tests different hardware parallelism at the element-level.
What to expect?: When increasing the element parallelism, the hardware

requirements and the energy consumption grow, but at the same time the execution time decreases linearly. From the plot we can see LUTs, FFs, Carry8 and dynamic power growing, but the speed increases. This is an essential trade-off for adapting the hardware architectures to different application scenarios.
Options: Varying frame sizes, e.g., 256, 512.

```
test_frame_parallelism(aeneas, frame_range=[256, 512], dim=1000)
```

Test 4: Feature Parallelism
Purpose: Assesses parallel processing of features.
What to expect?: same trade-off described for Test 3. In this case, the effect grows with the number of features in the dataset.
Options: Different levels of feature parallelism, e.g., 1, 7 (choose a divisor of the number of features).

```
test_feature_parallelism(aeneas, parallel_features_range=[1, 2])
```

Test 5: Class Parallelism
Purpose: Evaluates parallel processing of classes.
What to expect?: same trade-off described for Test 3. In this case, the effect grows with the number of classes in the dataset (and only for the inference stage).
Options: Different levels of class parallelism, e.g., 1, 3. (choose a divisor of the number of classes)

```
test_class_parallelism(aeneas, parallel_classes_range=[1, 3])
```

Test 6: Level Vector Technique
Purpose: Tests different Level Vector modes.
What to expect?: in this test, we will see that the Linear encoding achieves the best performance in accuracy. However, it requires storing all the LVs in memory. The approximate encoding, instead, requires just two level vectors in memory, that are used to build the others. In thermometer encoding, an increasing number of elements are active, for each level. Since at least half of the elements are always zero, this technique is more energy efficient.
Options: Modes like 'LINEAR', 'APPROX', 'THERMOMETER'.

```
test_lv_mode_model(aeneas, lv_mode_options=['LINEAR', 'APPROX'])
```

Test 7: Spatial Encoding
Purpose: Evaluates different spatial encoding techniques.

What to expect?: in this test, we will see that the record based encoding performs better and it is faster. Otherwise, it requires significantly more memory, since the index of each feature is modelled by a BV. Conversely, the n-gram encoding, permute each feature by a different number of positions. Permutation is slower and more complex to perform in hw.
Options: Techniques like ‘ENCODING_RECORD‘, ‘ENCODING_NGRAM‘.

```
test_spatial_encoding(aeneas, encoding_technique_options=['
    ENCODING_RECORD', 'ENCODING_NGRAM'])
```

Test 8: Temporal Encoding
Purpose: Investigates different n-gram sizes for temporal encoding.
Options: Various n-gram sizes, e.g., 2, 3.

```
test_temporal_encoding(aeneas, n_gram_size_options=[2, 3])
```

Test 9: Clipping Techniques
Purpose: Analyzes effects of different clipping techniques.
What to expect?: This these highlights the importance of selecting the correct clipping approach for the target problem. We can see how the binary clipping is the most efficient in hardware, but achieves the lowest performance. The power of two represents a perfect compromise for hardware implementations, since it allows to replace the costly multiplications and divisions with shifts.
Options: Methods like ‘CLIPPING_BINARY", "CLIPPING_QUANTIZED", "CLIPPING_POWERTWO", "CLIPPING_DISABLE‘.

```
test_clipping_techniques(aeneas, clipping_options=["CLIPPING_BINARY",
     "CLIPPING_QUANTIZED", "CLIPPING_POWERTWO", "CLIPPING_DISABLE"])
```

Test 10: Training on Hardware (TRAIN_ON_HW)
Purpose: Assesses hardware training’s effect.
What to expect?: when implementing the train on hardware, we expect a significant increase in the hardware resources. This is necessary to accumulate the HVS without losing precision. The clipping is, in fact, applied at the end.
Options: Enable (1) or disable (0) hardware training.

```
test_hw_train(aeneas, hw_train_options=[0, 1])
```

Test 12: Combined Parallelism
Purpose: Evaluates the combined effect of class, feature, and element parallelism.

Options: Various combinations of parallelism degrees.

```
test_comb_parallelism(aeneas, parallel_features_range=[1, 2],
   parallel_classes_range=[2, 4], frame_range=[256, 512])
```

Test 13: Clipping and Similarity Measures

Purpose: Evaluates combinations of clipping techniques and similarity measures.

Options: Various combinations of clipping and similarity measures.

```
test_clipping_similarity(aeneas, clip_range=['CLIPPING_BINARY', '
   CLIPPING_TERNARY'], sim_range=['SIMI_COS', 'SIMI_DPROD'])
```

Test 15: Test on CARDIO dataset

Purpose: Run the test described in the reference paper on the CARDIO dataset.

```
tests.test_dataset_cardio(aeneas)
```

Test 16: Test on EMG dataset

Purpose: Run the test described in the reference paper on the CARDIO dataset.

```
tests.test_dataset_cardio(aeneas)
```

Test 17: Test on HEPTA dataset

Purpose: Run the test described in the reference paper on the CARDIO dataset.

```
tests.test_dataset_cardio(aeneas)
```

Test 18: Test on Boston dataset

Purpose: Run the test described in the reference paper on the CARDIO dataset.

```
tests.test_dataset_cardio(aeneas)
```

4 Appendix: 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.

4.1 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 D.

4.2 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.

4.2.1 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 is achieved thanks to the distributed, holographic representation and enables aggressive scaling of the device.

4.2.2 Transparency

HVs are combined using a well-defined set of vector arithmetic operations (as we will see in the following sections) that can be inverted. Thanks to this property, HDC produces transparent (i.e., analyzable) models, which strongly contrast with conventional learning methods, which are often considered "black boxes."

4.2.3 One-shot learning

Unlike other neuro-inspired methods where learning is more computationally demanding than inference, in Hyperdimensional Computing (HDC), the training procedure is nearly identical to that of inference. This similarity enhances HDC's efficiency for hardware implementation, and make it ideal for online and continuous learning. New examples can be learned simply by incrementally updating the memory. Moreover, studies have shown that HDC requires significantly less training data while achieving the same accuracy as traditional AI approaches.

4.2.4 Energy Efficiency

Hyperdimensional Computing (HDC) operates by manipulating and comparing large patterns within the memory. As a result, the logic can be closely integrated with the memory, allowing computations to be distributed efficiently to save energy. This marks a significant departure from traditional Von-Neumann architectures, which involve moving data back and forth between the memory and processing unit, leading to memory bottlenecks.

In addition, HDC requires significantly fewer operations than other methods like SVMs, CNNs, KNNs, and MLPs. For instance, in EMG applications, HDC consumes only half the total power of an SVM when implemented on a commercial embedded ARM Cortex M4 [**?**].

The memory requirements for HDC scale linearly, unlike those for other networks. For example, a KNN for language recognition demands 500 times more memory than HDC.

Moreover, HDC benefits from temporal and spatial sparsity. At any given moment, only a small fraction of the memory or logic is active. For instance, a sparse binary representation, where the number of ones is significantly fewer than zeros, can lower switching activity and thus reduce power consumption.

4.3 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 [**?**]. 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.

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.

4.3.1 Superposition or Bundling

The *Bundling* ($\oplus$) combines two HVs into a single HV that encapsulates the essential aspects of both inputs, resulting in a vector that is similar to the original ones. Bundling is associative and commutative, simulating the simultaneous activation of multiple neural patterns, which aids in managing complex information within high-dimensional memory models.

$$s = a \oplus b \oplus c$$

Properties:

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

4.3.2 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:

$$(a \oplus b) \oplus (c \oplus d) = a \oplus b \oplus c \oplus d$$

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

HDC uses a *binding* ($\otimes$) operation, 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: $d = a \otimes b$ is similar to $d' = a' \otimes b'$
- 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.

4.3.3 Permutation

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

4.3.4 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 (1) is used, where $BundledHV_d$ is the d^{th} dimension of the $BundledHV$. This makes a trade-off between accuracy and hardware complexity.

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

4.3.5 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 5, the choice of similarity measure depends on the type of HVs involved. For binary HVs, the Hamming Distance is preferred, while for integer or bipolar ones, the Dot Product and Cosine Similarity are used. 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, Such an optimization is particularly valuable in applications for edge-devices 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 5: Similarity Measures in HDC based on HV Type

4.3.6 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

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

4.4 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 learning problems.

4.4.1 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.

4.4.2 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 D. 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. 10[**?**] for a visual depiction of this concept: an augmented dimensionality leads to a swift surge in HVs exhibiting similarity close to zero."

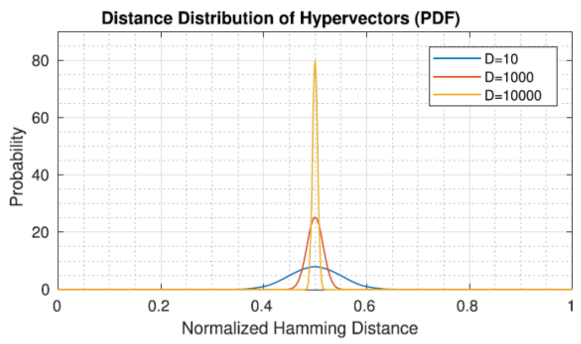


Figure 10: [**?**]

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

4.4.3 Scalar Values

When working with numeric scalar values, instead, 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.

4.4.3.1 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. So, basically, l HVs should be stored in memory.

4.4.3.2 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.

4.4.3.3 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: $\frac{m}{l}\frac{D}{2}$ elements set to '1'.

The memory containing the HV for representing scalar values is often called Continuous item-memory (CIM).

4.4.4 Sets

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

4.4.5 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.

4.4.6 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. 11. This example is taken from [**?**] and the 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 11: 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 4.4.3, 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. 12.

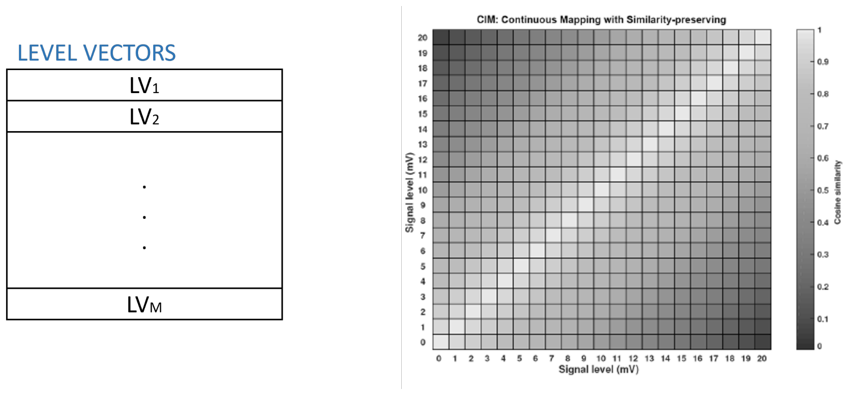


Figure 12: Caption

After these preliminary steps, the feature vector can be encoded using two main approaches: the record-based and the n-gram-based, shown in Fig. 14.

4.4.6.1 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. 13. Then, the encoding is performed as follows:

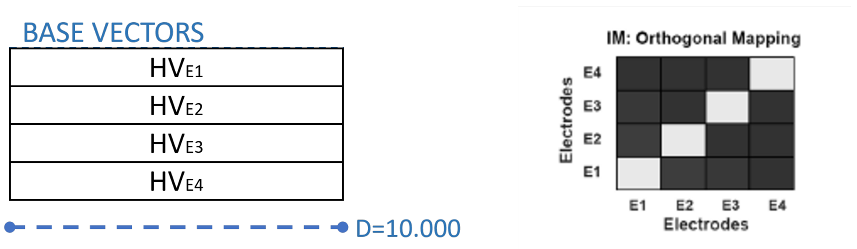


Figure 13: Base Vectors

- 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;
- 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 **??**.

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

4.4.6.2 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;

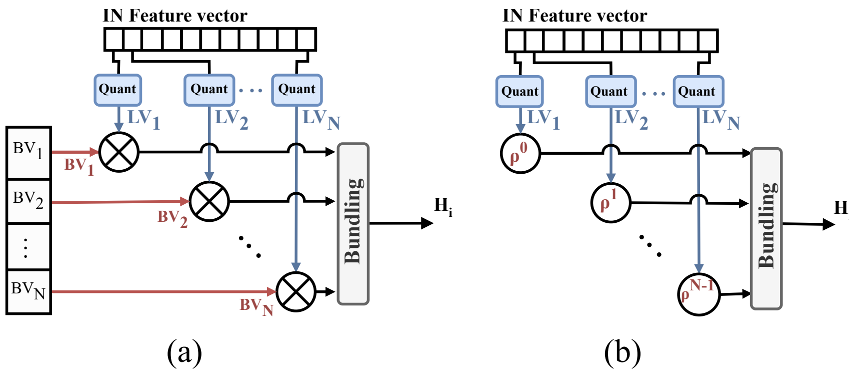


Figure 14: Visualization of encoding techniques for feature vectors: (a) Record-based (b) N-gram based

- 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.

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

4.4.7 Temporal Encoding

In sequential data processing, the model is tasked with handling not just isolated patterns but also temporal sequences. To effectively capture temporal dependencies, it necessitates operating within time windows of varying sizes. 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.

4.5 Classification in HDC

Classification is a supervised learning problem in which the model must predict categorical class labels of new input patterns based on past observations. Fig. 15(a) shows a general overview of the structure of an HDC classification model, including training, retraining and inference phases.

During *training* all the encoded H_i belonging to the same label are aggregated, as shown in (5), to build a representative prototype CV_j for each class j.

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

After training, *inference* is performed by encoding each input feature vector into a query HV (Q_i), and searching for the most similar CV in memory. It's important to note that while the encoding step typically consumes the most time during the training phase, the inference stage often places a heavier computational load on the associative search, accounting for up to 83% of the total execution time[**?**].

To improve the HDC model's accuracy, after the initial training and inference steps, 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 Q_i from the mispredicted class CV_{wrong} and adding it to the correct one, $C_{correct}$, as shown in (6). Notably, in the case of retraining, the clipping is only applied at the last step.

$$\begin{cases} C_{wrong} = C_{wrong} - Q_i \\ C_{correct} = C_{correct} + Q_i \end{cases} \tag{6}$$

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.

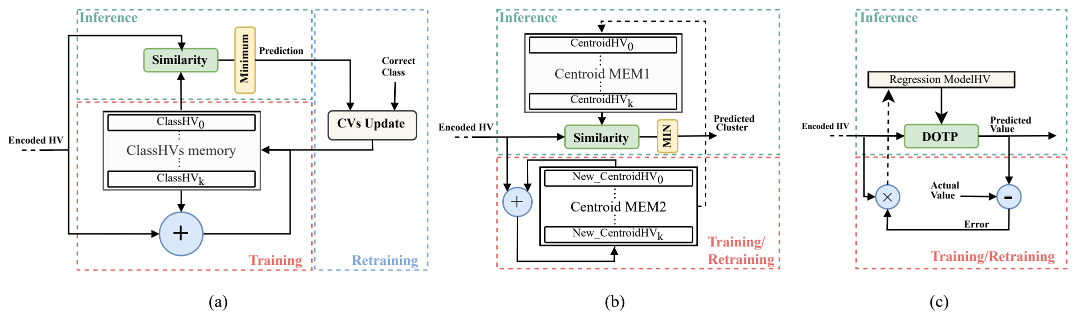


Figure 15: Visualization of typical HDC functional schemes for different problems: (a) Classification, (b) Regression, (c) Clustering

4.6 Regression in HDC

Regression, a core task in supervised learning, involves predicting the relationship between an input feature vector and its corresponding output y. This process utilizes a loss function to quantify the error between the predicted value $\hat{y}$ and the actual one y and update the model.

In the context of HDC, the regression procedure is detailed in Fig. 15(b), [**?**] and begins with initializing a Model HV (MV) to zero. Each dataset pattern i is first encoded in H_i using the techniques described in Section **??**, and then it is dot-multiplied with the MV to obtain the predicted value $\hat{y}_i$.

$\hat{y}_i$ is compared with the actual value y_i, to compute the error $E_i = y_i - \hat{y}_i$ that it is used to updated the MV as shown in (7). A learning rate α can be applied to modulate the update step. This process can be iterated over several epochs to adjust the MV towards minimizing the prediction error.

$$MV = MV + \alpha(E \times H_i) \tag{7}$$

4.7 Clustering in HDC

Clustering is an unsupervised learning task that involves organizing input patterns into K distinct clusters based on their similarity. This process in HDC is straightforward and can be implemented as illustrated in Fig. 15(c) [**?**, **?**].

Each cluster is represented by a Centroid HV (KV) that is stored in the Centroid memory and it is initialized with the HV of a pattern randomly chosen from the dataset. Patterns are encoded into HVs using the method detailed in Section **??** and, at each iteration, the model performs clustering by computing the similarity between H_i and each KV, assigning the pattern to the closest one.

During training, a second Centroid memory is used. When the cluster j

is selected for the given input pattern, the corresponding KV_j in the second memory is updated by adding H_i. After processing all the patterns in the training dataset, the first Centroid Memory is replaced with the updated one. This procedure can be repeated over several epochs.