UNIVERSITY OF CALIFORNIA SAN DIEGO
SAN DIEGO STATE UNIVERSITY

# Fast, Efficient, and Robust Learning with Brain-Inspired Hyperdimensional Computing

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Engineering Science (Electrical & Computer Engineering)

by

Justin Morris

Committee in charge:

University of California San Diego
Professor Tajana Šimunić Rosing, Co-Chair
Professor Ryan Kastner
Professor Farinaz Koushanfar

San Diego State University
Professor Baris Aksanli, Co-Chair
Professor Shangping Ren

2022

University of California San Diego

San Diego State University

2022

DEDICATION

***To my wife, Stephanie, and our family***

*For their endless support and encouragement to go on and complete this journey*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

## ACKNOWLEDGEMENTS

I would like to first thank my advisors, Professor Tajana S. Rosing and Professor Baris Aksanli for their guidance and support during my Ph.D. I am truly grateful for their support through the past years. Tajana's enthusiasm to explore a wide range of research directions pushed me to never leave a stone unturned, as well as, learn about different fields through collaboration. I'd like to thank Baris for his support in all aspects of life throughout the years, giving me the motivation I needed to continue on and for bringing new perspectives through his experience to my research. I would also like to thank my dissertation committee members, Professors Shangping Ren, Farinaz Koushanfar, and Ryan Kastner, for their feedback and discussions related to my Ph.D. work. I would like to thank all my colleagues in SEELab for their active collaboration and continuous support. I would like to give special thanks to Mohsen Imani, Yeseong Kim, Saransh Gupta, Behnam Khaleghi, Anthony Thomas, Yilun Hao, Kazim Ergun, Minxuan Zhou, and Onat Gungor. My research was made possible by funding from the National Science Foundation (NSF) Grant 1527034, 1730158, 1826967, 1830331, 1911095, 2003279, CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and SRC-Global Research Collaboration grant.

I owe so much to all my family for their understanding, patience, and encouragement that helped me navigate through the difficult moments during the past few years. Finally, I would like thank all my friends for their constant support.

The material in this dissertation is based on the following publications.

Chapter 2, in part, is a reprint of the material as it appears in J. Morris, S. Set, G. Rosen, M. Imani, B. Aksanli, and T. Rosing, "AdaptBit-HD: Adaptive Model Bitwidth for Hyperdimensional Computing" in IEEE International Conference on Computer Design (ICCD), 2021. The dissertation author was the primary investigator and author of this material.

Chapter 3, in part, is a reprint of the material as it appears in J. Morris, Y. Hao, R. Fernando, M. Imani, B. Aksanli , T. Rosing, "Locality-based Encoder and Model Quantization for Efficient Hyper-Dimensional Computing". *IEEE Transactions on Computer-Aided Design* *of Integrated Circuits and Systems (TCAD)*, 2021. The dissertation author was the primary investigator and author of this material.

Chapter 4, in part, is a reprint of material as it appears in J. Morris, Y. Hao, S. Gupta, R. Ramkumar, J. Yu, M. Imani, B. Aksanli, T. Rosing, ”Multi-label HD Classification in 3D Flash”. IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC), 2020. The dissertation author was the primary investigator and author of this material.

Chapter 5, in part, is a reprint of material as may appear in J. Morris, K. Ergun, B. Khaleghi, M. Imani, B. Aksanli , T. Rosing, ”HyDREA: Utilizing Hyperdimensional Computing For A More Robust and Efficient Machine Learning System.” *ACM Transactions on Embedded Computing Systems (TECS)*, 2022. The dissertation author was the primary investigator and author of this material.

My co-authors (Prof. Baris Aksanli, Kazim Ergun, Roshan Fernando, Sarnash Gupta, Yilun Hao, Mohsen Imani, Behnam Khaleghi, Ranganathan Ramkumar, Gadi Rosen, Prof. Tajana S. Rosing, Si Thu Kaung Set, and Jeffrey Yu listed in alphabetical order) have all kindly approved the inclusion of the aforementioned publications in my dissertation.

## VITA

| | |
|---|---|
| 2018 | Bachelor of Science in Computer Science and Engineering (Computer Engineering), University of California, San Diego, USA |
| 2022 | Doctor of Philosophy in Engineering Science (Electrical & Computer Engineering), University of California, San Diego, USA and San Diego State University, USA |

## PUBLICATIONS

Justin Morris, Yilun Hao, Saranash Gupta, Behnam Khaleghi, Baris Aksanli, Tajana Rosing. ”Stochastic-HD: Leveraging Stochastic Computing on the Hyper-Dimensional Computing Pipeline”, *Frontiers in Neuroscience*, 2022.

Behnam Khaleghi, Jaeyoung Kang, Hanyang Xu, Justin Morris, Tajana Rosing ”GENERIC: Highly Efficient Learning Engine on Edge using Hyperdimensional Computing”, *Design Automation Conference (DAC)*, 2022.

George Armstrong, Cameron Martino, Justin Morris, Behnam Khaleghi, Jaeyoung Kang, Jeff DeReus, Qiyun Zhu et al. ”Swapping Metagenomics Preprocessing Pipeline Components Offers Speed and Sensitivity Increases.” *Msystems* (2022): e01378-21.

Justin Morris, Kazim Ergun, Behnam Khaleghi, Mohsen Imani, Baris Aksanli, and Tajana Rosing, ”HyDREA: Utilizing Hyperdimensional Computing For A More Robust and Efficient Machine Learning System.” *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

Justin Morris, Hin Wai Lui, Kenneth Stewart, Behnam Khaleghi, Anthony Thomas, Thiago Marback, Baris Aksanli, Emre Neftci, and Tajana Rosing, ”HyperSpike: HyperDimensional Computing for More Efficient and Robust Spiking Neural Networks,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022.

Yilun Hao, Saransh Gupta, Justin Morris, Behnam Khaleghi, Baris Aksanli, and Tajana Rosing, ”Stochastic-HD: Leveraging Stochastic Computing on Hyper-Dimensional Computing,” in *IEEE 39th International Conference on Computer Design (ICCD)*, 2021.

Justin Morris, Si Thu Kaung Set, Gadi Rosen, Mohsen Imani, Baris Aksanli, and Tajana Rosing, ”AdaptBit-HD: Adaptive Model Bitwidth for Hyperdimensional Computing,” in *IEEE 39th International Conference on Computer Design (ICCD)*, 2021.

Alice Sokolova, Mohsen Imani, Andrew Huang, Ricardo Garcia, Justin Morris, Tajana Rosing, and Baris Aksanli, ”MACcelerator: Approximate Arithmetic Unit for Computational Acceleration,” in *22nd International Symposium on Quality Electronic Design (ISQED)* 2021.

Justin Morris, Yilun Hao, Roshan Fernando, Mohsen Imani, Baris Aksanli, and Tajana Rosing, ”Locality-based Encoder and Model Quantization for Efficient Hyper-Dimensional Computing,” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.

Behnam Khaleghi, Hanyang Xu, Justin Morris, and Tajana Šimunić Rosing, ”tiny-HD: Ultra-Efficient Hyperdimensional Computing Engine for IoT Applications,” in *Design, Automation Test in Europe Conference  Exhibition (DATE)*, 2021.

Justin Morris, Kazim Ergun, Behnam Khaleghi, Mohsen Imani, Baris Aksanli, and Tajana Rosing, ”Hydrea: Towards more robust and efficient machine learning systems with hyperdimensional computing,” in *Design, Automation  Test in Europe Conference  Exhibition (DATE)*, 2021.

Yunhui Guo, Mohsen Imani, Jaeyoung Kang, Sahand Salamat, Justin Morris, Baris Aksanli, Yeseong Kim, and Tajana Rosing, ”HyperRec: Efficient Recommender Systems with Hyperdimensional Computing,” in *26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021.

Rebecca K. Fielding-Miller, Smruthi Karthikeyan, Tommi Gaines, Richard S. Garfein, Rodolfo A. Salido, Victor Cantu, Laura Kohn et al. ”Wastewater and surface monitoring to detect COVID-19 in elementary school settings: The Safer at School Early Alert project.” *Medrxiv* (2021).

Saransh Gupta, Justin Morris, Mohsen Imani, Ranganathan Ramkumar, Jeffrey Yu, Aniket Tiwari, Baris Aksanli, and Tajana Simunic Rosing, ”THRIFTY: Training with Hyperdimensional Computing across Flash Hierarchy,” in *IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2020.*

Justin Morris, Yilun Hao, Saransh Gupta, Ranganathan Ramkumar, Jeffrey Yu, Mohsen Imani, Baris Aksanli, and Tajana Simunic Rosing, ”Multi-label HD Classification in 3D Flash,” in *Proceedings of IFIP/IEEE International Conference on VLSI and System-on-Chip (VLSI-SoC), 2020.*

Mohsen Imani, Justin Morris, Samuel Bosch, Helen Shu, Giovanni De Micheli, and Tajana Rosing, ”Adapthd: Adaptive efficient training for brain-inspired hyperdimensional computing,” in *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2019.

Justin Morris, Mohsen Imani, Samuel Bosch, Anthony Thomas, Helen Shu, and Tajana Rosing, ”CompHD: Efficient hyperdimensional computing using model compression,” in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2019.

Mohsen Imani, Justin Morris, John Messerly, Helen Shu, Yaobang Deng, and Tajana Rosing, ”Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing,” in *Proceedings of the 56th Annual Design Automation Conference (DAC)*, 2019.

Mohsen Imani, Justin Morris, Helen Shu, Shou Li, and Tajana Rosing, ”Efficient associative search in brain-inspired hyperdimensional computing,” in *IEEE Design Test 37*, no. 1 2019 28-35.

Mohsen Imani, Tarek Nassar, Justin Morris, and Tajana Rosing, ”DNA Sequencing using Brain-inspired Hyperdimensional Computing,” in *GOMACTech Conference*, 2019.

ABSTRACT OF THE DISSERTATION

Fast, Efficient, and Robust Learning with Brain-Inspired Hyperdimensional Computing

by

Justin Morris

Doctor of Philosophy in Engineering Science (Electrical & Computer Engineering)

University of California San Diego, 2022
San Diego State University, 2022

Professor Tajana Šimunić Rosing, Co-Chair
Professor Baris Aksanli, Co-Chair

With the emergence of the Internet of Things (IoT), devices will generate massive data streams demanding services that pose huge technical challenges due to limited device resources. Furthermore, IoT systems increasingly need to run complex and energy intensive Machine Learning (ML) algorithms, but do not have the resources to run many state-of-the-art ML models, instead opting to send their data to the cloud for computing. This results in insufficient security, slower moving data, and energy intensive data centers. In order to achieve real-time learning in IoT systems, we need to redesign the algorithms themselves using strategies that more closely model the ultimate efficient learning machine: the human brain.

This dissertation focuses on increasing the computing efficiency of machine learning on IoT devices with the application of Hyperdimensional Computing (HDC). HDC mimics several desirable properties of the human brain, including: robustness to noise, robustness to hardware failures, and single-pass learning where training happens in one-shot without storing the training data points or using complex gradient-based algorithms. These features make HDC a promising solution for today's embedded devices with limited storage, battery, and resources, and the potential for noise and variability. Research in the HDC field has targeted improving these key features of HDC and expanding to include even more features. There are four main paths in HDC research: (1) Algorithmic changes for faster and more energy efficient learning, (2) Novel architectures to accelerate HDC, usually targeting lower power IoT devices, (3) Extending HDC applications beyond classification, (4) Exploiting the robust property of HDC for more efficient and faster inference, and (5) HDC Theory, its connection to neuroscience and mathematics. This dissertation contributes to four of these research paths in HDC.

Our contributions include: (1) We introduce the first adaptive bitwidth model for HDC [2]. In this work we propose a new quantization method and during inference we iterate through the bits along all dimensions taking the hamming distance. At each iteration, we check if the current hamming distance passes a threshold similarity, if it does, we terminate execution early to save energy and time. (2) We create a redesign of the entire HDC process with a locality-based encoding, quantized retraining, and online dimension reduction during inference, all accelerated by a new novel FPGA design [3]. In this work we our locality-based encoding removes random memory accesses from HDC encoding as well as adds sparsity for more efficiency. We also introduce a general method to quantize to any desired model bitwidth. Finally, we propose a method to find any insignificant dimensions in the HDC model and remove them for more energy efficiency during inference. (3) We extend HDC to support multi-label classification [4]. We perform multi-label classification by creating a binary classification model for each label. Upon inference, our models determine if each label exists independently. This is different than prior work that took the power set of the labels to reduce the problem to a single label classification

as HDC scales poorly with this method. (4) Finally, we experimentally evaluate the robustness of HDC for the first time and create a new analog PIM architecture with reduced precision Analog to Digital Converters (ADC), exploiting that robustness [5]. We test HDC robustness in a federated learning environment where edge devices send encoded hypervectors to a central server wirelessly. We evaluate the impact of any wireless transmission errors on this data and show that HDC is 48$\times$ more robust than other classifiers. We then use this knowledge that HDC is robust to create a more efficient analog PIM circuit by reducing the bitwidth of the ADCs.

# Chapter 1

# Introduction

We live in a world where technological advances are continually creating more data than what we can cope with. With the emergence of the Internet of Things (IoT), devices will generate massive data streams demanding services that pose huge technical challenges due to limited device resources [6, 7, 8, 9]. For example, IoT devices are increasingly supporting many Machine Learning (ML) applications. However, these devices do not have the required computing resources or even battery life to support state-of-the-art ML models such as DNNs. Instead, these embedded devices send their data to a cloud server, where models are run at a data center scale. This poses multiple problems as sending data to the cloud for processing is not scalable, cannot guarantee the real-time response, and is often not desirable due to privacy and security concerns. Much of IoT data processing will need to run at least partly on devices at the edge of the Internet.

In order to achieve real-time learning in IoT systems, we need to rethink the algorithms we use for machine learning and redesign them using strategies that more closely model the ultimate efficient learning machine: the human brain. Neuromorphic, or brain-inspired, models aim to close this gap of available resources at the edge and required resources for ML applications. Hyperdimensional Computing (HDC) is one of the Neuromorphic computing models that can offer, brain-like efficiency [10]. HDC is based on a short-term human memory model, the sparse distributed memory, that emerged from theoretical neuroscience. A key benefit of HDC is its natural robustness to noise. This is of critical importance in IoT systems, where noise and high error rates are common during communication. In the rest of this chapter, we discuss an overview of the HDC algorithm, related work, and the contributions of this thesis in more detail.

## 1.1 Hyperdimensional Computing

Our research has been instrumental in developing practical implementations of HDC - a computational technique modeled after the brain [10]. The HDC system enables large-scale learning in real-time, including both training and inference. HDC is motivated by the observation that the key aspects of human memory, perception, and cognition can be explained by the mathematical properties of high-dimensional spaces. It models data using points of a high-dimensional space, called hypervectors. These points can be manipulated with formal algebra operations to represent semantic relationships between objects. HDC mimics several desirable properties of the human brain, including: robustness to noise, robustness to hardware failures, and single-pass learning where training happens in one-shot without storing the training data points or using complex gradient-based algorithms. These features make HDC a promising solution for today's embedded devices with limited storage, battery, and resources, and the potential for noise and variability.

HDC has 3 main parts, encoding, training, and inference. In the next subsections we give an overview of each of these modules and the HDC algorithm for classification. Figure 1.1 shows an overview of HDC clssificaiton.

### 1.1.1 Encoding

The first step of HDC is to math the input feature vector to high dimensional space. Consider a feature vector $\mathbf{v} = \langle v_1, \dots v_n \rangle$. The encoding module takes this $n$-dimensional vector and converts it into a $D$-dimensional hypervector (HV) ($D >> n$). We utilized the encoding module proposed in [11]. The encoding is performed in three steps, which we describe below. The first step is to create two sets of HVs, *ID* HVs and level HVs. Both ID HVs and *level* HVs

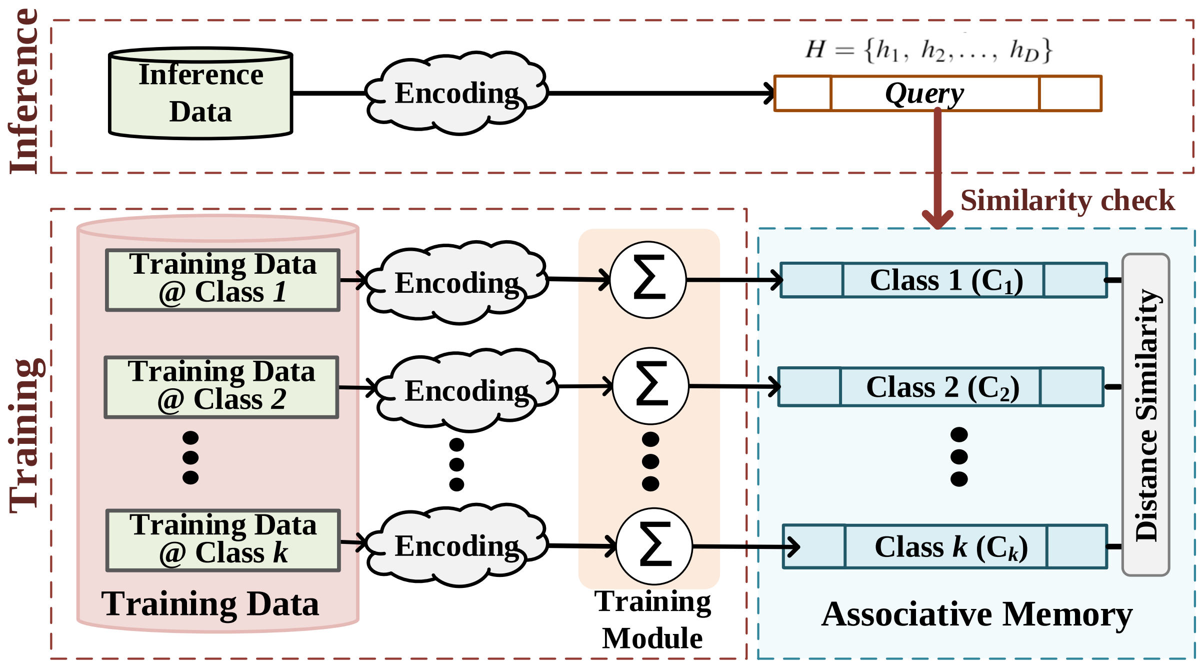


**Figure 1.1.** Overview of creating an HD model and performing inference with an HD model

are $D$ dimensional HVs where each element is either $-1$ or $1$. The encoding scheme assigns a unique channel $ID$ HV to each feature position. $ID$s are hypervectors which are randomly generated such that all features will have orthogonal channel $ID$s, i.e., $\delta(ID_i,\ ID_j) < 5{,}000)$ for $D = 10{,}000$ and $i \neq j$; where the $\delta$ measures the element-wise similarity between the vectors. The HD computing encoder also generates a set of *level* HVs to consider the impact of each feature value. To create these level hypervectors, we compute the minimum and maximum feature values among all data points, $\mathbf{v}_{min}$ and $\mathbf{v}_{max}$, then quantize the range of $[\mathbf{v}_{min}, \mathbf{v}_{max}]$ into $m$ levels. Each level is then assigned a corresponding level HV: $\mathbf{LV} = \{\mathbf{LV}_1, \cdots, \mathbf{LV}_m\}$. To encode a feature vector, the encoder looks at each position of the feature vector and element-wise multiplies the channel ID ($ID_i$) with the corresponding level hypervector ($hv_i$). The following equation shows how an $n$-length feature vector is mapped into the HD space with this encoding scheme:

$$H \ = \ [hv_1 * ID_1 \ + \ hv_2 * ID_2 + \ldots \ + \ hv_n * ID_n]$$

$$hv_j \in \{LV_1,\ LV_2, \ldots,\ LV_m\},\ \ 1 \leqslant j \leqslant m$$

$$ID_i \in \{-1,1\}^D, LV_j \in \{-1,1\}^D$$

### 1.1.2 Training

The next step in HDC is to train the model. HDC supports efficient one-pass training. To build a one-pass model, the encoder maps all training data to training HVs ($\mathscr{H}$). For all training HVs within a class ($\{\mathscr{H}_i^1,\ \mathscr{H}_i^2, \ldots, \mathscr{H}_i^j\}$), HD computing adds them together to create a single class HV ($\mathscr{C}_i$).

$$\mathscr{C}_i\ =\ \mathscr{H}_i^1\ +\ \mathscr{H}_i^2\ +\ \ldots\ +\ \mathscr{H}_i^j$$

Once this is done for every class, we have an HD model that can be used for inference. However, we can significantly improve the accuracy of our HD model with retraining [11]. We retrain the HD model by inputting each training data point through the HD model as a query hypervector ($Q$). We look at the similarity of the query hypervector to all stored class hypervectors; (i) if the query is correctly classified by the current model, our design does not change the model. (ii) If it is incorrectly matched with the $i^th$ class hypervector ($C_i$), when it actually belongs to $j^th$ class ($C_j$), our retraining procedure subtracts the query hypervector from the $i^th$ class and adds it to $j^th$ class hypervector: $C_i = C_i - Q$ and $C_j = C_j + Q$. Retraining can run until a user-provided error threshold is met, or a maximum number of iterations is reached. After training, class HVs are stored in the classifier.

### 1.1.3 Inference

Upon inference, the encoder first maps the input data into a query HV ($\mathscr{Q}$), using the same encoding that was used to train the HD model. A similarity metric is used to determine the strength of a match between the query HV and each class HV. The most common metric used in HD computing is cosine similarity, but note that other metrics (e.g. Hamming distance) could be appropriate depending on the problem [12]. After the similarity is computed between the query HV and each class HV in the classifier, the class with the highest similarity is chosen as the output class.

Research on HDC has focused on five main topics: Algorithmic improvements, Hardware/Software Co-Design, Extending Applications, Exploiting Robustness, and HDC Theory, its connection to neuroscience and mathematics. In this dissertation, we focus on the first four research paths.

### 1.1.4 Algorithmic Improvements

HDC is a relatively new topic of research in the machine learning field and has been rapidly growing since 2017 [13]. One issue with HDC is data size due to encoding to high dimensional space (D=10,000). To address this, model quantization has been proposed to reduce the bitwidth of the hypervectors and therefore, reducing the overall data size [14, 15]. There have been recent works to apply similar algorithmic techniques to improve HDC energy efficiency, execution time, and accuracy that have previously been implemented on Neural Networks as well. For instance, we proposed the first work to implement a learning rate to the HDC algorithm [16]. This work achieves an adaptive learning rate by finding the similarity difference between the incorrectly selected class and the correct class with the query. Then, inputs that difference into a lookup table to select the learning rate. Finally, the learning rate is multiplied to the query before it is used to update the model. Lastly, work has been done to reduce the dimensionality of the HDC model without significantly impacting accuracy [17]. This work proposes two methods to reduce the dimensionality by finding insignificant dimensions. The first is class-wise eliminating the dimensions with the lowest variances across the classes. The second is dimension-wise, eliminating the dimensions in each class that are closest to 0.

### 1.1.5 Hardware/Software Co-Design

There has also been lots of work on creating new efficient architectures with the HDC algorithm in mind. For instance, one of the main benefits of HDC is that it has highly parallel simple operations. This property makes it a perfect match for Processing in Memory (PIM) architectures. HDC has been accelerated by both analog and digital PIM [18, 5, 19]. Work has also been done to remove the reduction operations in HDC on the algorithm side to further improve the energy efficiency in PIM as PIM architectures struggle with reduction operations [20]. This work achieves this by mapping all HDC operations to the stochastic domain. In the stochastic domain, all mathematical operations are approximated by bitwise operations. With this method, the authors were able to eliminate the reduction part of the dot product, which was the main bottleneck in digital PIM. There has also been work on creating more efficient FPGA or ASIC designs for HDC [17, 21].

### 1.1.6 Extending Applications

Recent work has been done to extend HDC applications beyond classification. For instance, HDC has recently been extended to support clustering [22]. Furthermore, HDC has been extended to support recommender systems for the first time [23]. There is also work on using HDC for DNA classification [24]. However, there are still multiple different ML applications HDC has not been mapped to yet. For instance, in this dissertation, we introduce the first work on utilizing HDC for multi-label classification.

### 1.1.7 Exploiting Robustness

One of the key properties of HDC is that it is robust to noise and other sources of errors such as bitflips in hardware. Multiple previous works cite this property and utilize it indirectly to create more efficient architectures [17, 15, 25]. For instance, work in [17] attempts to remove a large portion of dimensions for more efficient inference. This can be viewed as introducing errors into the inference operation, but because HDC is robust to these errors, the HDC model is able to maintain accuracy. However, none of these previous works explicitly demonstrated the robustness of HDC compared to other classifiers.

## 1.2 Thesis Contributions

Our contributions include: (1) The first adaptive model bitwidth for HDC [2], (2) a new hardware friendly encoding that creates a sparse locality-based encoding with a fixed memory access pattern, a general method for quantizing HDC, and a method to reduce the dimensionality of the HDC model [3], (3) extending the HDC algorithm to support multi-label classification [4], and (4) evaluating and exploiting the robustness of HDC for more efficient hardware [5]. .

### 1.2.1 Adaptive Model Quantization for Hyperdimensional Computing

As mentioned in Section 1.1.4, research on HDC algorithmic changes have extended ideas from other ML algorithms to HDC, such as adding a learning rate, model quantization, and sparsity. However, all previous work on model quantization for HDC has been static quantization. For instance, work in [15] quantizes only to binary and work in [14] only supports ternary quantization. This leads to two extremes in the energy and accuracy trade-off curve. Either, highly accurate models with less aggressive quantization and more energy consumption. Or, highly efficient models with aggressive quantization, such as binary models, but at the cost of accuracy loss. In this dissertation, we propose a new model quantization method that adapatively changes the effective bitwidth for every sample [2]. We do this by iterating over the bits of the quantized numbers along all dimensions. At each iteration, we take the hamming distance and check if the highest similarity passes a threshold. If it does, we can terminate execution early saving energy and time. This enables our design to achieve energy efficiency and execution time comparable to a binary model, while also achieving a similar accuracy to the full precision model. This work is discussed in Chapter 2 of this dissertation.

### 1.2.2 A Rework of the Hyperdimensional Computing Pipeline and Acceleration on FPGA

There are multiple different works on mapping HDC to different hardware platforms and using hardware/software co-design to achieve more efficient solutions, as mentioned in Section 1.1.5. However, previous designs to not solve the problem of encoding to HD space. Previous encoding algorithms require random memory accesses and a massive amount of element wise multiplications and additions in the dimensionality of 10,000. These encodings are inefficient to map to hardware designs [11]. In this dissertation, we propose a new hardware friendly encoding that removes random memory accesses and replaces them with a fixed memory access pattern with sparsity and a subsequent FPGA architecture that takes advantage of the changes [3]. We additionally propose a general quantization method to quantize HDC to any bitwidth. Finally, we propose a dimension reduction method to remove insignificant dimensions in the HDC model. Overall, our new architecture achieves 64× energy efficiency and 10× faster execution time than the previous state of the art FPGA implementation of HDC. This work is discussed in Chapter 3 of this dissertation.

### 1.2.3 Extending Hyperdimensional Computing Applications to Support Multi-Label Classification

As mentioned in Section 1.1.6, HDC is a growing field of research and much work has been done on extending HDC to other ML applications beyond classification. However, there were no works on extending HDC to support multi-label classification. In this dissertation, we present our work on extending HDC to support multilabel classification [4]. Prior work on other classifiers map to multilabel by simply taking the power set of the possible labels and creating new labels for each combination. This reduces the multilabel problem to a single label problem. This works well for other classifiers as they don't scale as poorly with an increase in classes. However, HDC scales linearly with the number of classes, so it scales exponentially with the number of labels in a multilabel problem. Therefore, instead of using the power set idea seen in other models, we propose a binary classification model for each label. Overall, with this idea, we achieve 47× faster execution time, 48× better energy efficiency, and 5% higher accuracy than other multilabel classifiers. This work is discussed in Chapter 5 of this dissertation.

### 1.2.4 Evaluating and Exploiting Robustness to Create a More Efficient Analog Processing-in-Memory Accelerator for Hyperdimensional Computing

As mentioned in Section 1.1.7, multiple works in the HDC space cite that HDC is robust to noise and hardware errors and often take advantage of that property in their designs. However, before our work in [5] there were no empirical results to back up the claims of robustness or comparisons with other ML models. We include this work in this dissertation and demonstrate experimentally that HDC is 48× more robust to noise than other machine learning models. We furthermore demonstrate how to exploit this property with more efficient hardware. Previous analog PIM designs have a vital flaw where up to 90% of energy is used in the analog to digital conversion (ADC) [26]. However, we alleviate this issue when mapping HDC to analog PIM by reducing the ADC bitwidth. This reduces ADC energy consumption by half for every bit dropped, but results in inaccurate conversions. However, because HDC is robust to these errors, it is able to tolerate the inaccurate operations up to a point. Overall, our design is able to achieve 289× better energy efficiency than simply mapping HDC onto the existing architecture. This work is discussed in Chapter 4 of this dissertation.

Finally, in Chapter 6, we summarize our work on HDC and discuss future directions of research in HDC. We discuss two directions of future work: 1) utilizing feature extraction with HDC and 2) further eploiting and evaluating the robustness of HDC.

# Chapter 2

# Adaptive Model Quantization for Hyperdimensional Computing

## 2.1 Introduction

Existing HD computing quantization methods have two main challenges: (i) the trade-off between accuracy and energy efficiency has to be decided before training the model, and the model would have to be retrained from scratch to change bitwidths if the accuracy and energy efficiency trade-off requirements change. (ii) Existing model quantization techniques ignore that not all samples need to be quantized with the same value. Some samples can be classified with simple binary representations, while others require higher bitwidths for accurate classification. In other words, there exists no adaptive bitwidth quantization for HD computing. Adaptive bitwidth quantization adds another level of fine tuning for systems balancing the accuracy and energy efficiency trade-off.

In this chapter, we propose AdaptBit-HD, which, to the best of our knowledge, is the the first Adaptive Model Bitwidth Architecture for accelerating HD computing. AdaptBit-HD does not change the bitwidth of the representation of the data, but operates on the bits of the quantized model in a bitserial way to save energy when fewer bits can be used to find the correct class. AdaptBit-HD can achieve both high accuracy by utilizing all bits when necessary and high energy efficiency and faster execution time by terminating execution at lower bits when our design is confident in the output. AdaptBit-HD achieves this by performing a bitseial hamming distance operation on the query HV and class HVs. We check after each bit if we are confident enough in our current answer to terminate execution early. To achieve this, we completely redesign the HD computing algorithm including training, retraining, and inference. We accordingly design an end-to-end HD FPGA accelerator for AdaptBit-HD and compare with a state-of-the-art binary quantization FPGA accelerator for HD [15] as well as a 16-bit static quantization method. Compared to binary quantization AdaptBit-HD is 1.1% more accurate at the cost of just 10% more energy consumption and 7% more execution time. Compared to 16-bit models, AdaptBit-HD is 14$\times$ more energy efficient at the cost of 0.5% accuracy.

## 2.2 Related Work

Model quantization is a widely used technique in machine learning applications to improve energy efficiency. For instance, Google's TPU for performing inference on DNNs utilizes reduced bit representations [27]. Furthermore, [28] proposes a quantization method for SVMs. Model quantization has also been used to reduce the memory requirement for a more efficient hardware design [29]. Other methods such as model compression have also been used to improve the energy efficiency of neural networks [30]. Model quantization has also been widely used to accelerate and improve the energy consumption of HD computing [14, 31]. Although model quantization has improved other machine learning methods such as DNNs, light-weight models such as HD computing continue to be more energy efficient and for applications where energy efficiency is paramount, light-weight models should continue to be utilized.

Prior work applied high-dimensional computing to different classification problems such as language recognition, speech recognition, face detection, EMG gesture detection, human-computer interaction, and sensor fusion prediction [32, 33, 34, 22, 35]. Although HD computing is more energy efficient than other traditional machine learning models such as DNNs and MLPs, there has been a significant amount of work on pushing the limits of HD computing to achieve even better energy efficiency. Prior work accelerated HD computing by removing dimensions of the class hypervectors [17], or compressing the HD model [36]. Work in [37] also proposed a dynamic dimensionality model to improve energy efficiency.

Prior work has also shown that quantizing the class hypervectors can provide significant energy and speedup improvements at a small cost in accuracy [31, 38]. Work in [14] extended the idea of binarizing the class HV weights to using a ternary model to achieve higher accuracies. However, all of the existing work on HD computing for model quantization is static. This poses a few problems. For instance, if accuracy and energy efficiency needs change, the model needs to be completely retrained to change bitwidths. Additionally, by being static, one has to choose where they land on the accuracy and energy trade-off curve at a macro level. This often leads to leaning towards one end of the spectrum, either highly energy efficient with accuracy loss, or highly accurate with high energy consumption. This problem is exaggerated for applications with varying precision needs based on the incoming data. However, in this chapter, we propose an adaptive bitwidth quantization method that chooses the best bitwidth per sample to achieve a confident classification with minimal energy usage. This leads to an overall design that achieves both high accuracy and high energy efficiency.

## 2.3 AdaptBit-HD

In this chapter, we propose AdaptBit-HD, the first Adaptive Model Bitwidth Architecture for accelerating HD computing. HD computing consists of three main modules shown in Figure 1.1: encoding, training, and inference. The encoding module maps each data point to high-dimensional space. The HD model accumulates every encoded training hypervector (HV) to create an integer model. This integer model is then used to create a quantized model. During inference, HD computing then chooses the most similar class to the query HV as the output class. AdaptBit-HD fundamentally changes the inference phase by operating on the bits of the quantized model in a bitserial way to save energy when fewer bits can be used to find the correct class. We check after each bit if we are confident enough in our current answer to terminate execution early based on a threshold of similarity. By operating with this new inference technique, AdaptBit-HD is able to achieve the energy efficiency of binary models, while maintaining the accuracy of full precision models. To further support our proposed bitserial inference design, AdaptBit-HD accordingly proposes a training approach that trains the model to create quantized HVs and tunes the model to improve the confidence of the threshold we utilize to determine if we can terminate execution early. In the following, we explain the details of both the baseline HD functionality and AdaptBit-HD functionality.

### 2.3.1 Training with AdaptBit-HD

Existing model quantization techniques result in faster and more efficient machine learning models. This quantization also leads to less area, because the model is represented with values smaller than 32 bits. However, all prior techniques quantize all samples to the same bitwidth. This leads to a non-optimal design as even binary quantizations have been shown to provide high accuracy. Therefore, many samples can be correctly classified with lower bitwidth representations and higher bitwidth quantizations should be reserved for samples that are more difficult to separate them. In this section, we go over how we can achieve this adaptive model quantization with HD computing.

Figure 2.1 demonstrates the idea that not all samples need the same bitwidth to be accurately labeled. For instance, in the figure, we can see that HD computing is able to achieve an average of 95.7% accuracy with binary values. Additionally, by moving to a $5-bit$ representation, HD is able to improve in accuracy by 1.16% on average. Comparing the $5-bit$ representations with full precision, we can see that the 16-bit precision model is only able to achieve 0.42% more accuracy than the $5-bit$ models. This demonstrates that for most samples, we can get away with aggressive model quantization. However, there are some samples that require more bits to separate the data properly to maintain high accuracy. Rather then using high precision for all of the samples to achieve high accuracy, we can adaptively select the bitwidth we need for the sample during inference to balance both accuracy and energy efficiency. This

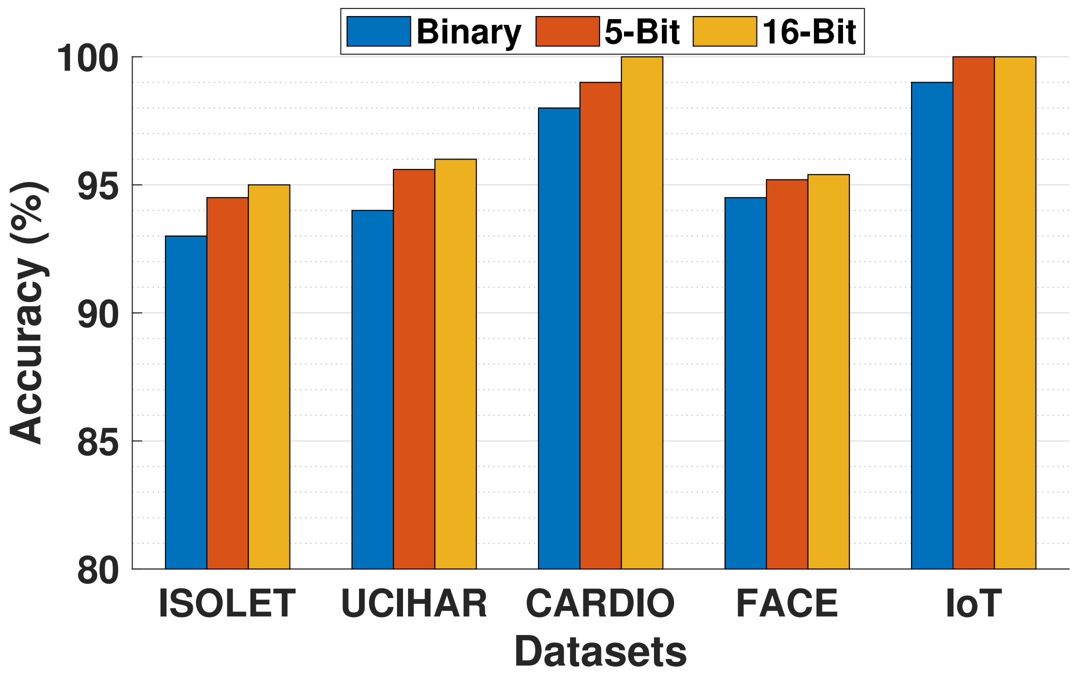


**Figure 2.1.** Difference in Accuracy with Various HD Bitwidth Representations for HD Computing

**Figure 2.2.** Overview of Creating a AdaptBit-HD Model During Retraining

balance is important for application where accuracy and energy efficiency are equally important such as when performing a medical diagnosis with a mobile device running on a battery.

**Initial Training:** The initial training for model quantization is very similar to the initial training for the baseline model without quantization, as we first build the full bitwidth model by combining all samples as described in 2.3.1. The training process for model quantization diverges from that of past work after the initial training. As Figure 2.2 shows, We first normalize all of the class HVs such that all of the dimensions are in the range [-1,1], but we still keep the non-normalized vectors around. We then quantize the normalized vectors to the nearest power of 2 in a list of quantized values.

The list of powers of two is defined by two parameters: $n$, the number of bits, and $o$, the offset of the powers of two. The offset is to control where in the range of values $(-1,1)$ we want to have a higher resolution of representation. Higher offsets leads to better quantization near 0. We first set aside one bit for representing 0. Then the rest of the $2^{(n-1)}$ representations are defined as follows: $2^{(r-o)}$ where r = -1, -2, ..., $-(2^{(n-2)})$ and $o$ is the offset. The reason we only iterate $r$ $2^{(n-2)}$ times is because we additionally represent the same powers of two on the negative side. Each power of two representation is then assigned a unique $n-bit$ binary string representation. For example, if $n = 3$ and $o = 0$, we would be able to represent the following powers of 2 in our model quantization: $(-2^{-1}, -2^{-2}, 0, 2^{-1}, 2^{-2})$, where each one of these values is assigned a unique $3-bit$ sequence of $0s$ and $1s$. Once we set each dimension to the closest power of two we can represent, we then have an HD model where each dimension is an $n-bit$ value.

To encode our weights to unique $n-bit$ values, we assign the first bit to indicate if the value is negative or positive. This ensures that the first bit hamming distance is equivalent to how binary models are created for HD. We reserve the second bit to indicate if the value is zero or not. Thus, the calculation of the hamming distance for the second bit is equivalent to counting the number of matching zeros. For the rest of the bits, because we use hamming distance as our similarity metric (which is explained in Section 2.3.2) on the binary representation of our values instead of the cosine similarity of the values themselves, it is important that values near each other have a small hamming distance score between each other. To achieve this, we use a grey code encoding to assign the last bits of the binary strings to each power of 2. This ensures that any adjacent quantized value differs by only one bit.

**Retraining:** The retraining process for model quantization is also similar to the retraining process for the baseline model without quantization. Throughout training, we store both a full precision model and an $n-bit$ representation model of the class hypervectors. We retrain the quantized model by iterating through the training set. In a single iteration of model adjustment, HD computing checks the similarity of all training data points, say $\mathbf{H}$, with the class hypervectors in the quantized binary model. The data point is assigned to the class with which it has the closest similarity. If the datapoint is correctly classified, normally, no model update is needed. However, in Section 2.3.2, we modify this to support adaptively stopping the similarity check in

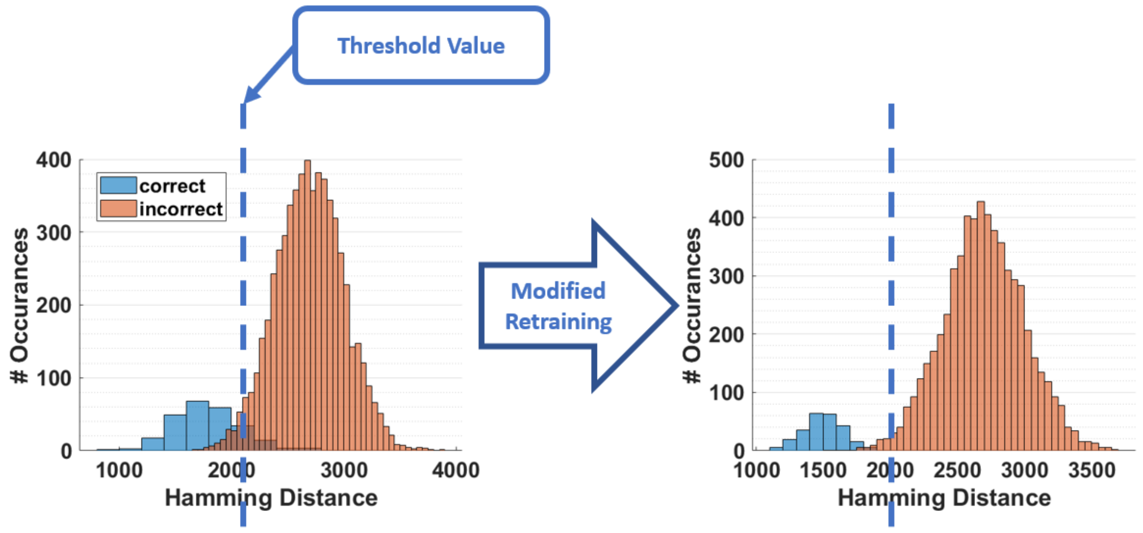


**Figure 2.3.** Distributions of Hamming Distance Calculations Before (left) and After (right) Retraining

a bit-serial manner. If a data point is incorrectly classified by the model, HD updates the model by (i) adding the incorrectly classified hypervector to the class the input data point belongs to ($\widetilde{\mathbf{C}}^{correct} = \mathbf{C}^{correct} + \mathbf{H}$), and (ii) subtracting it from the class to which it is wrongly matched ($\widetilde{\mathbf{C}}^{wrong} = \mathbf{C}^{wrong} - \mathbf{H}$). These changes are made to the full precision model saved from training because adding to and subtracting from the quantized model would drastically change the model. To update the quantized model, the updated class hypervectors from the integer model are quantized via the same process described in Section 2.3.1. Saving the full precision model does incur an overhead, but after retaining is complete, we can discard the full precision model.

## 2.3.2 Inference with AdaptBit-HD

To support our bitserial hamming distance check, the query is quantized the same way as the class HVs. Then, AdaptBit-HD calculates the hamming distance between the first bit of the class HVs and the query HV across all the dimensions. We then check to see if the class with the highest similarity passes a threshold value. If the similarity threshold is passed, then execution can stop prematurely and output the current highest similarity class. However, if the

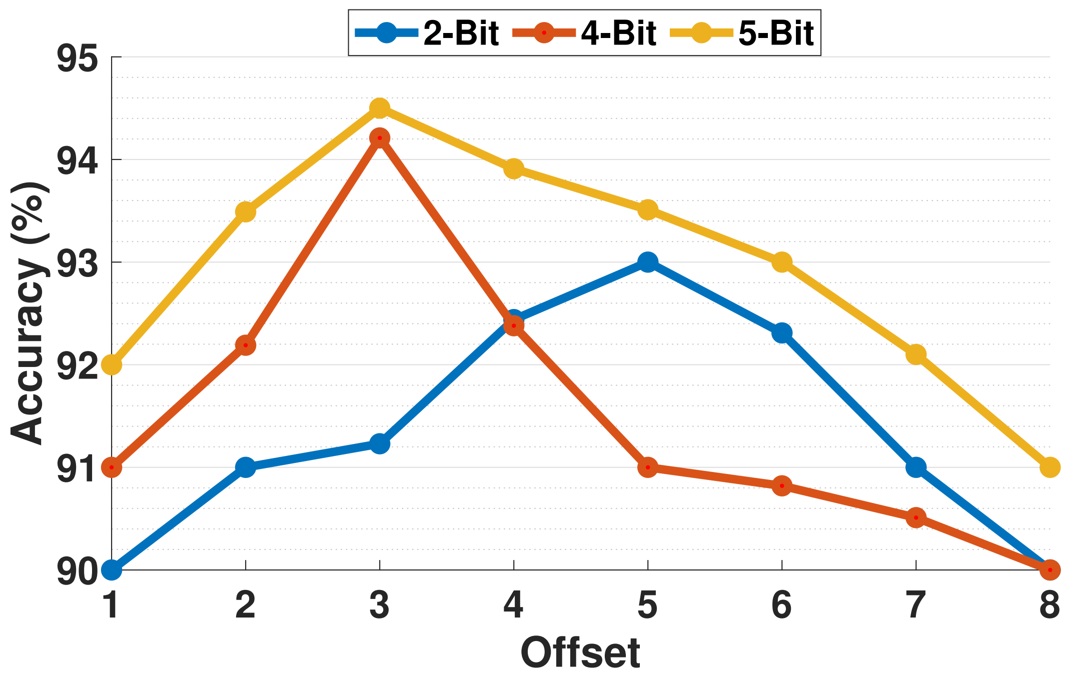


**Figure 2.4.** Effect of AdaptBit-HD Parameters on Accuracy

threshold is not passed, then computation continues to the next bit and the hamming distances are accumulated. We then check the if the similarity threshold is met again and if it is not, we continue the process. If the similarity threshold is not met and we are on the last bit, the most similar class is the output.

**Bitserial Thresholding** In order to support the bitserial hamming distance, we need to create a threshold for the termination condition. To do this, after the initial training, for the first iteration of retraining, we collect the hamming distance for all samples and the class HVs. Then, we get the mean and standard deviation of all samples where our model was correct as well as the mean and standard deviation of all samples where our model was incorrect. We, additionally, separate these distributions by each class, which allows us to set a threshold per class, rather than one global threshold. We then set the threshold to be the average between the *mean - standard deviation* of the incorrect distribution and the *mean + standard deviation* of the correct distribution. This heuristic makes sense to use because the threshold should separate the two distributions. As hamming distance calculates the number of mismatches, the incorrect samples should be clustered with higher hamming distance values and the correct samples should be clustered in a lower distribution. Therefore, averaging in this way gives us a good initial threshold that separates the two distributions. We do this process on a per class basis as the distribution of similarity values differs on a per class basis, therefore, we need a different threshold for each class.

Figure 2.3 shows the distribution of hamming distances for all samples for a single class. The graph on the left shows our initial threshold value. However, the graph shows that the initial distributions are not completely separated by the threshold. To fix this, we modified the retraining algorithm to actively create a greater separation in order to minimize outputting the incorrect class when the threshold is met. To do this we made the following change: if the datapoint is correctly classified, rather than doing nothing, because we are correct, we additionally check if the similarity threshold was met. If the threshold is met with an additional 10% guard-band, we do nothing. However if the threshold is not met with an additional 10% guard-band, we add the query hypervector to the class the input data point belongs to ($\widetilde{\mathbf{C}}^{correct} = \mathbf{C}^{correct} + \mathbf{H}$). As Figure 2.3 shows on the graph to the right, after retraining the distribution of incorrectly classified and correctly classified samples are further separated leading to more accurate classification when we terminate the bitserial operation early. The 10% guard-band is to help ensure we push the distribution of correct samples past the threshold value. This leads to a more accurate model when terminating early based on the threshold. As Figure 2.3 shows, when the hamming distance passes the threshold, we can be confident that it is the correct class.

Figure 2.4 shows the impact of using different offsets and bitwidths for AdaptBit-HD on the ISOLET dataset, however, the results are similar for all datasets tested. As the figure shows, there is a balance between having too high and too low of an offset. This is because with too low of an offset, we have less quantization resolution near 0. However, with too high of an offset, we again will not have enough resolution at the ends of our distribution ($[-1,1]$). The figure shows that generally, an offset of 3 gives the best balance in quantizing the range of values. The exception is when using 2 bits as with so few bits, we can only choose to have good resolution at either the ends of the distribution or near 0. It turns out that having higher resolution near 0 is more important leading to higher accuracies with higher offsets when only using 2 bits. Additionally, we saw that accuracy becomes saturated at 5 bits, and is comparable to a model with no quantization in Figure 2.1. This stayed consistent across all datasets tested. Therefore, for the rest of our experiments, we use a total bitwidth of 5 and offset of 3.

## 2.4 FPGA Acceleration

AdaptBit-HD can be accelerated on different platforms such as CPU, GPU, FPGA, or ASIC. FPGA is one of the best options as AdaptBit-HD computation involves bitwise operations among long vector sizes. For example, encoding and associative search. Additionally, unlike ASICs or PIM implementations, FPGAs offer reprogramability and faster design times. General strategies of optimizing the performance of AdaptBit-HD are (i) using a pipeline and partial unrolling on a low level (dimension level) to speed up each task and (ii) using dataflow design on a high level (task level) to build a stream processing architecture that lets different tasks run concurrently. In the following, we explain the functionality of AdaptBit-HD in encoding, training, retraining, and inference phases.

### 2.4.1 Encoding Implementation

We used the locality-based random projection encoding to implement the encoding module [15]. Due to the sequential and predictable memory access patterns as well as the abundance of binary operations, this encoding approach can be implemented efficiently on an FPGA. In the hardware implementation, we represent all $\{-1,+1\}$ values with $\{0,1\}$ respectively. This enables us to represent each element of the projection vector using a single bit. Figure 3.8a shows the hardware implementation of the AdaptBit-HD encoding module. The encoding process includes reading a feature vector from off-chip DDR memory and generating a binary hypervector from them.

Calculating the inner product of a feature vector and a projection vector, $P \in \{1,-1\}^D$,

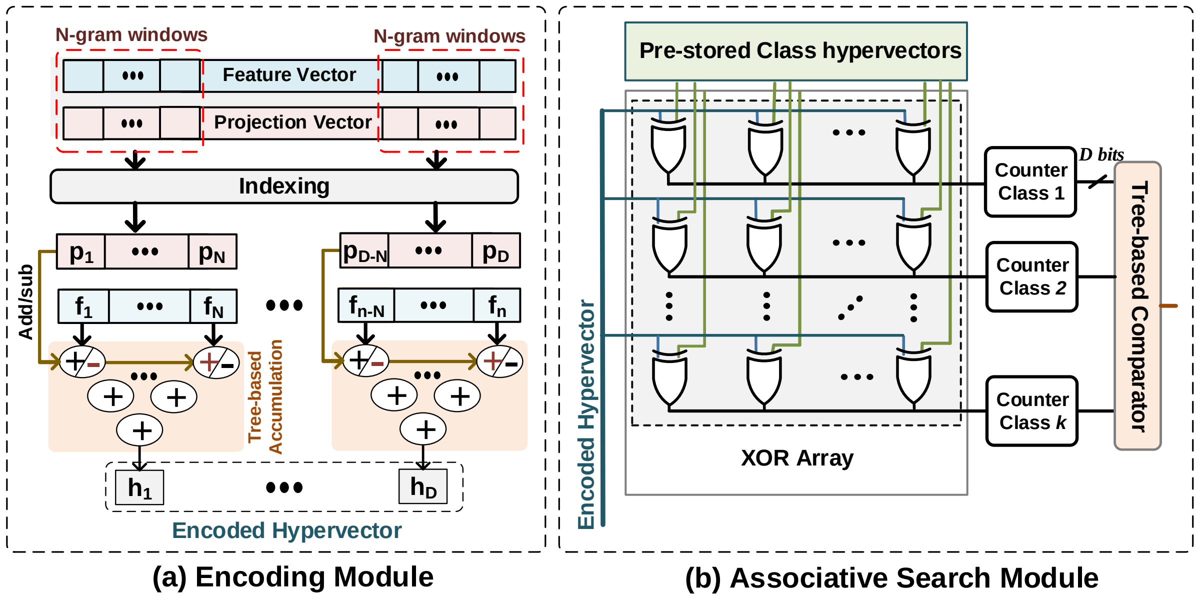


**Figure 2.5.** FPGA implementation of the encoding and associative search block.

can be implemented with no multiplications. Each element of the projection vector decides the sign of each dimension of the feature vector in the accumulation of the dot product. Thus, the dot product can be simplified to the addition/subtraction of the feature vector elements. We use Look Up Tables (LUTs) and Flip Flops (FFs) resources of the FPGA to implement the encoding module, rather than DSPs for this this simple addition, which leads to better energy efficiency. We additionally write back the resulting encoded HV to memory, only when the encoding is performed independently. Right after the encoding, the hypervectors are used for initial model training. The same encoded hypervectors needs to be accessed multiple times during the retraining process. However, the FPGA does not have enough BRAM blocks to store all encoded hypervectors, so, our design stores them into DDR memory.

## 2.4.2 Training Implementation

**Initial Training:** Like previously, initial training for AdaptBit-HD with model quantization is a single-pass process through the training dataset. The training module accesses the encoded hypervectors and accumulates them in order to create a hypervector representing each class. We exploit data-flow design implementing the encoding and initial training modules in a pipeline structure. When the training module accumulates the encoded hypervector to one of the class hypervectors, the encoding module maps the next training data into high-dimensional space, improving data throughput by increasing resource utilization. This improves FPGA throughput by maximizing resource utilization as well as hides the latency of the encoding.

After going through all of the training data, our implementation creates an n-bit quantized representation of the model. We first normalize all of the class HVs such that all of the dimensions are in the range [-1,1], but we still keep the non-normalized class HVs for use during retraining. We then quantize the normalized vectors to the nearest power of 2 in a list of quantized values as described in Section 2.3.1. Finally, the quantized n-bit model is stored in the BRAM blocks to be used for inference or retraining. Since the generating and writing are done only once during the entire training process, they do not impact the performance of the training phase. Thus, these two parts are not fully optimized, allowing our design to saves some resources for the retraining phase, which is more critical to the overall performance.

**Retraining:** Retraining is implemented separately from training, since the final result of initial training, the n-bit model, will be used in retraining, so that they are performed sequentially. The retraining phase first sequentially reads already encoded training hypervectors from the off-chip memory in batches to help hide the latency of reading from the off-chip memory. This is necessary as each read has a latency of about $15ns$, which would slow down the retraining process. Next, we check the hamming distance similarity of each data point with all trained class hypervectors. As mentioned in Section 2.3.2, this is performed in a bit serial fashion. To support this in hardware we split the hamming distance calculation into its own pipeline stage. If the threshold is met, then the pipeline continues. However, if the threshold is not met and we need to go to the next bit, the pipeline is stalled and the next bit is calculated. This process continues until the threshold is met or we are out of bits to process. This may look like a large performance impact by stalling the pipeline, however, as the experimental results section demonstrates, over 90% of all samples terminate at just one bit, so we do not need to stall the pipeline often. At the end, each data point gets a tag of a class in which it has the highest Hamming distance.

The Hamming distance similarity check is implemented using an `XOR` array which compares the bit similarity between two hypervectors. Counter blocks, shown in Figure 3.8b, calculate the number of mismatches of each class hypervector with the query data point. Finally, a tree-based comparator block finds the class with the lowest counter value. In the case of misclassification, AdaptBit-HD needs to update the model by adding and subtracting a data hypervector with two class hypervectors as defined before. Like encoding, all retraining processes can be implemented using LUTs and FFs blocks.

The uniqueness of the retraining implementation in our design is that we have two different models in similarity check and updating. This avoids the situation of similarity check and updating access to the same model simultaneously, which makes the dataflow design in a single iteration possible. After certain batch size iterations, the process stream is stopped for a while and the model in similarity check will be refreshed by the model in updating. This design will bring $2\times$ better speed and little influence on the result of retraining.

## 2.4.3 Inference Implementation

After the retraining, the quantized AdaptBit-HD model has a stable model that can be used in the inference phase. The encoding module is integrated with the similarity check module as the entire inference part. Each test data point is first encoded to high-dimensional space using the same encoding block explained in Section 3.6.1. Next, the quantized AdaptBit-HD model checks the Hamming distance similarity of the data point with all pre-stored class HVs, in a bit serial manner, in order to find a class with the highest similarity. One unique advantage of our approach is its capability to enable online training during the inference phase. Our implementation stores two HD models: one with integer values used for retraining and an n-bit quantized model which is used to perform the classification task. AdaptBit-HD quantizes the integer model to an n-bit model periodically to update the inference model. While the integer model is updated each time the classification from the n-bit model is incorrect.

## 2.5 Evaluation

### 2.5.1 Experimental Setup

We implemented AdaptBit-HD training, retraining, and inference in both software and hardware. In software, we implemented AdaptBit-HD with Python. In hardware, we fully implemented AdaptBit-HD using Verilog. We verify the timing and the functionality of the models by synthesizing them using Xilinx Vivado Design Suite [39]. The synthesis code has been implemented on the Kintex-7 FPGA KC705 Evaluation Kit. We compare AdaptBit-HD with baseline HD, an FPGA implementation of [15] using a binary model. We additionally compare to a static 5 bit HD computing design.

We evaluated the efficiency of the proposed AdaptBit-HD on four practical classification problems listed below: Speech Recognition (ISOLET) [40], Activity Recognition (UCI-HAR) [41], Face Detection (FACE) [42], Cardiotocography (CARDIO) [43], and Attack Detection in IoT systems (IoT) [44].

### 2.5.2 Energy Efficiency, Execution Time, and Accuracy of AdaptBit-HD vs State-of-the-Art

Figure 5.8 shows the impact of the bitserial operations of AdaptBit-HD on accuracy and compares AdaptBit-HD with both a baseline of binary quantization and comparison with a 16-bits model. The graph clearly shows that AdaptBit-HD achieves comparable accuracy with 16-bit model. AdaptBit-HD loses only 0.1% accuracy when compared to a 16-bit models on average and AdaptBit-HD is 1% more accurate than binary models on average.

Figure 2.7 compares the energy efficiency of AdaptBit-HD normalized to a 16-bit quantized model as a 16-bit model is able to achieve the same accuracy as full precision models with significantly less energy consumption. The figure also compares AdaptBit-HD with a binary quantized design which is the current state-of-the-art quantization for HD computing to achieve the best energy efficiency [15]. Figure 2.7 demonstrates that AdaptBit-HD is able to achieve

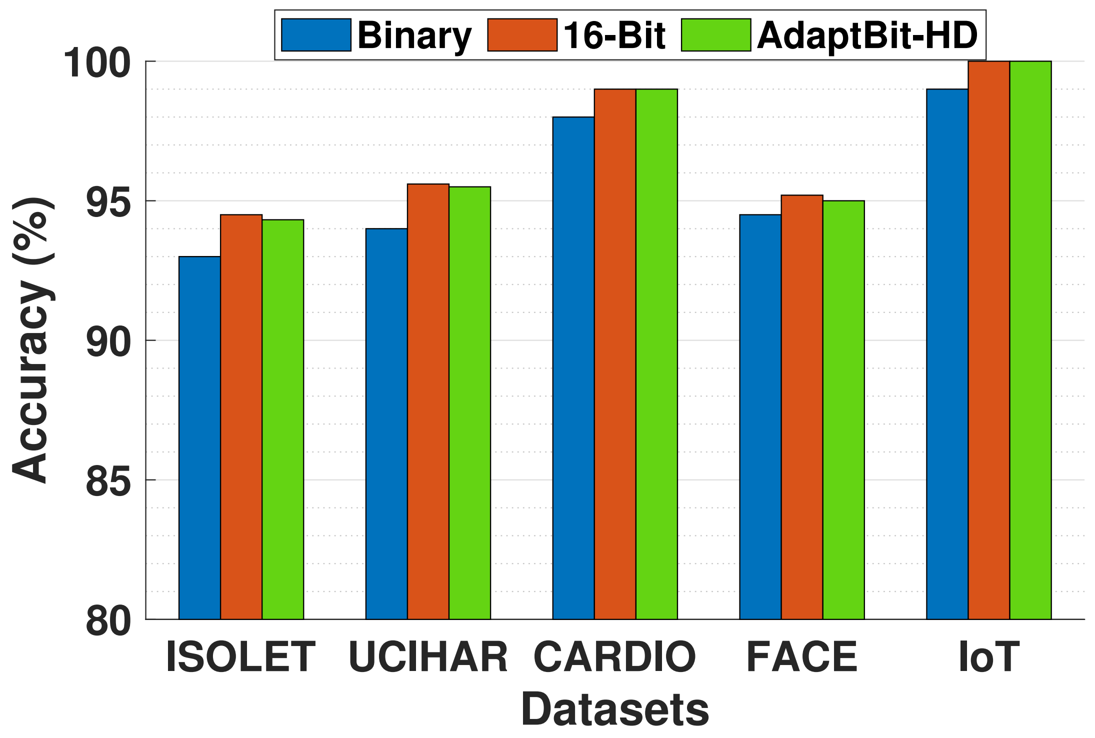


**Figure 2.6.** Comparison of the Accuracy of AdaptBit-HD to Static Model Quantization Methods

energy efficiency close to that of the binary quantized model, where we define energy efficiency as the relative amount of overall energy consumption over the entire dataset. AdaptBit-HD is only 10% less energy efficient than the binary model. This is because AdaptBit-HD is able to terminate the bitserial operation at the first bit the majority of the time, just like a binary model. This is demonstrated by the color coding of the stacked bar. The color coding of the different bits for AdaptBit-HD show the proportion of energy consumption spent on each bit. For example, the proportion of the bar that is colored blue is the proportion of energy spent on the first bit, which is approximately 90%. This is because, 90% of the time, only the first bit is used. This leads to approximately the same energy consumption as a binary model 90% of the time. Figure 2.8 additionally compares the execution time of AdaptBit-HD with a binary model and 16-bit model. The y-axis shows the speedup of AdaptBit-HD and binary models relative to a 16-bit model. As the graph demonstrates, we see a similar comparison with execution time as energy efficiency. This is because the two are closely related and we accordingly see a similar speedup as well as energy efficiency improvement.

AdaptBit-HD is additionally, able to achieve 1.1% more accuracy than the binary model. Overall, AdaptBit-HD is comparable in energy efficiency and execution time to the binary model, but more accurate. Compared to the 16-bit model, AdaptBit-HD is 14.4× more energy efficient at the cost of just 0.1% accuracy. This demonstrates that with an adaptive bitserial operation, AdaptBit-HD is able to achieve energy efficiency and execution time close the the binary model while maintaining accuracy comparable to 5-bit models and full precision models. Therefore, AdaptBit-HD offers another point to the accuracy and energy efficiency trade-off curve that aims to suit application where energy efficiency and accuracy are more equally important. Additionally, by designing an FPGA accelerator for AdaptBit-HD, FPGAs can be reconfigured based on user application needs and changes. If a user needs highly efficient energy consumption and accuracy loss is not as important, they can use the binary model we compare with. For applications on the other side of the spectrum, they can use a higher bitwdith static model, for instance, a 5-bit model. Then, for applications that need to balance accuracy and energy efficiency such as running a medical diagnosis application, similar to our CARDIO dataset, on a mobile device with a battery, then AdaptBit-HD would be the best suited quantization method to use.

One downside to AdaptBit-HD is that even though not all bits are used for each sample, we still store 5-bit quantized vectors. Therefore, binary models are approximately 5× more area efficient than AdaptBit-HD, if the design only performs inference.

### 2.5.3 AdaptBit-HD Area Comparison

Although our design is able to achieve similar accuracy to a non-quantized model with similar execution time and energy consumption to a highly quantized binary model, there is still overhead. At a glance, the most apparent drawback with our implementation of a bit-serial operation is that our design needs to store the full 5-bit class HVs all the time. This results in the area of our design being comparable to a 5-bit quantized model rather than the more space efficient binary model. Figure 2.9 shows the area comparison of a binary model with AdaptBit-

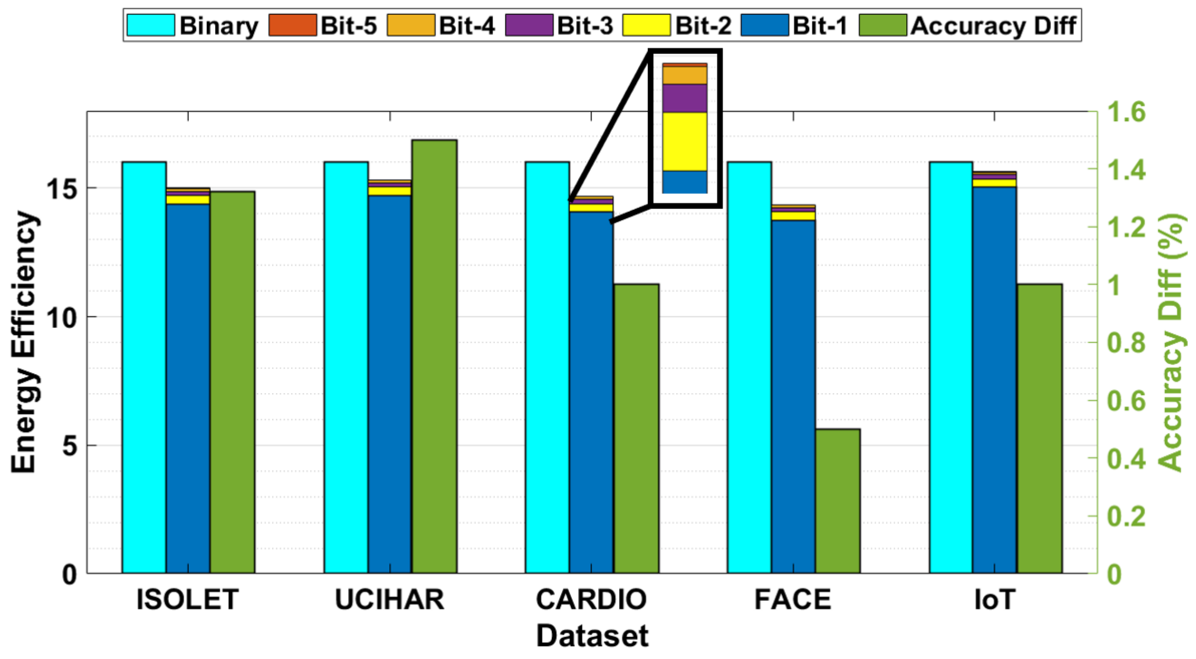


**Figure 2.7.** Energy Breakdown of AdaptBit-HD and Comparison with Static Quantization Methods. Energy Efficiency is Shown Relative to a 16-Bit Model. Accuracy Difference is Compared to a Binary Model. The color coding of the different bits for AdaptBit-HD show the proportion of energy consumption spent on each bit.

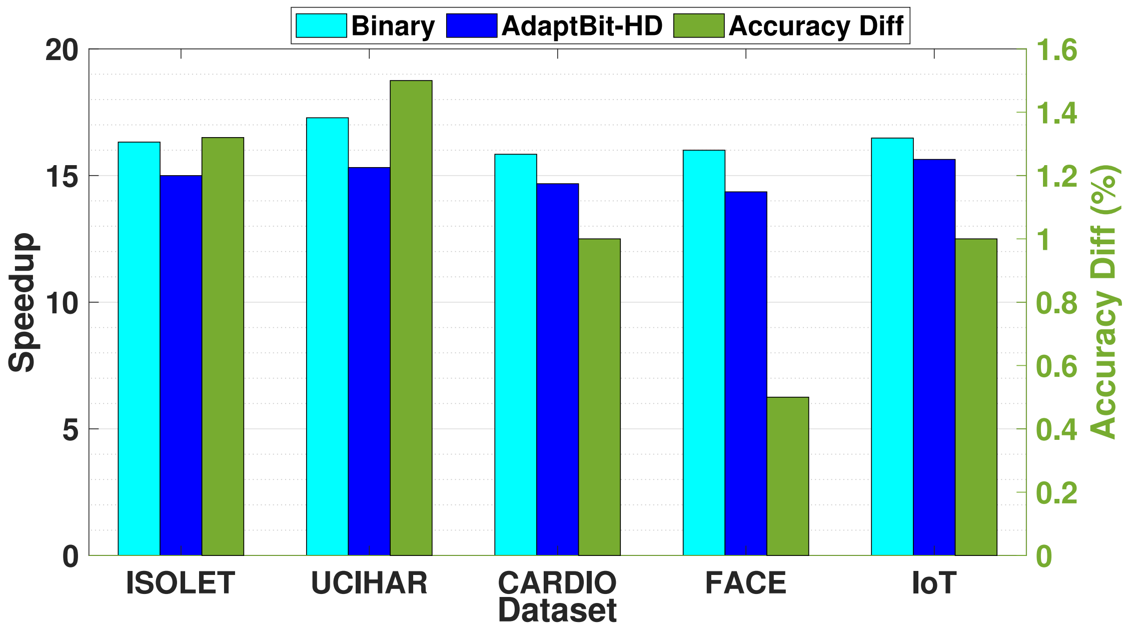


**Figure 2.8.** Speedup of AdaptBit-HD and Comparison with Static Quantization Methods. Speedup is Shown Relative to a 16-Bit Model. Accuracy Difference is Compared to a Binary Model.

HD relative to a 16-bit model. There are two comparisons, the first is during just inference and the other is an end-to-end implementation of HD, which includes encoding, training, retraining, and inference. As the figure shows, a binary design that performs inference only uses roughly 6% of the area of a 16-bit model as most of the area usage comes from storing the large class HVs. On the other hand, AdaptBit-HD uses close to 33% of the area of a 16-bit model. This is because AdaptBit-HD stores 5-bit HVs. However, if we look at an end-to-end implementation, the area usage converges. This is because both quantization methods need to store 16-bit precision class HVs during retraining, which takes a bulk of the total area.

Additionally, HD as a classification method is significantly more area efficient than other light-weight learning models such as SVMs. We compare with SVMs, because they offer similar accuracy to HD computing in most datasets and are also relatively light-weight compared to neural networks, just like HD computing. Here we show that SVMs use 110$\times$ (3.24$\times$) more area during inference (end-to-end). This is because during inference, SVMs need to store all support vectors in their original data representation (32-bit). Similar to HD, SVMs also need access to all training data during training. However, SVMs take 3.24$\times$ more space due to HD mapping the data to high dimensional binary vectors.

## 2.6 Conclusion

In this chapter we propose AdaptBit-HD, an Adaptive Model Bitwidth Architecture for accelerating HD computing. AdaptBit-HD operates on the bits of the quantized model in a bitserial way to save energy when fewer bits can be used to find the correct class. With AdaptBit-HD, we achieve both high accuracy by utilizing all the bits when necessary and high energy efficiency by terminating execution at lower bits when our design is confident in the output. Compared to binary quantization AdaptBit-HD is 1.1% more accurate at the cost of just 10% more energy consumption. Compared to a 16-bit static model, AdaptBit-HD is 14.4$\times$ more energy efficient and 15.1$\times$ faster at the cost of just 0.1% accuracy. This demonstrates that with

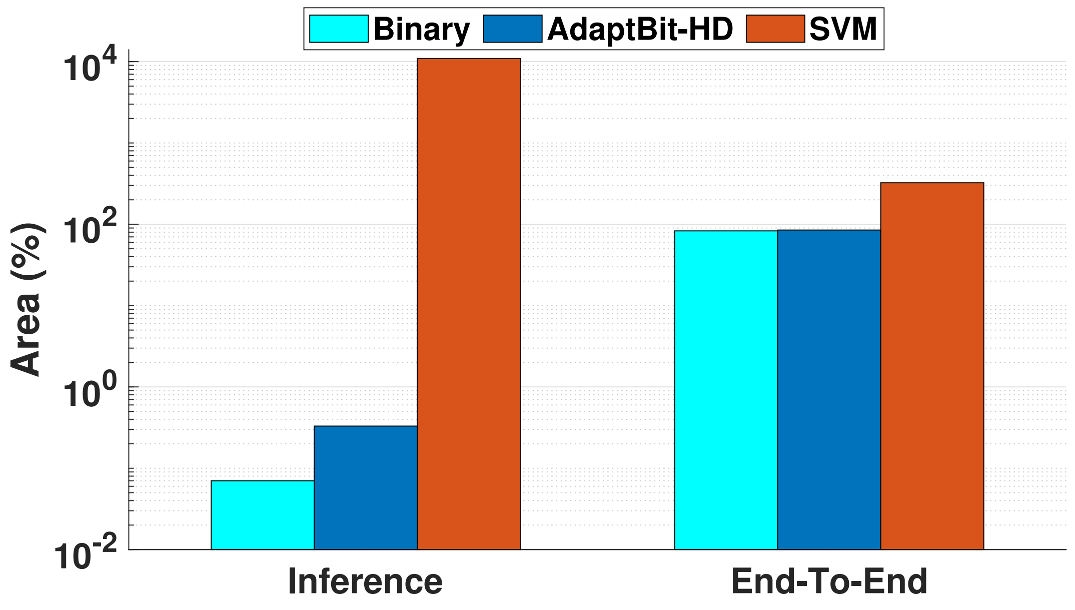


**Figure 2.9.** Area Comparison of AdaptBit-HD, Static Quantization Methods for HD, and SVMs. Area Used is Shown Relative to a 16-bit Static HD Model.

an adaptive bitserial operation, AdaptBit-HD is able to achieve energy efficiency and execution time close the the binary model while maintaining accuracy comparable to a 16-bit model. In the next chapter we discuss our work on ReHD, which is a complete rework of HDC Classification utilizing hardware/software co-design principles to create a more efficient FPGA accelerator.

## 2.7 Acknowledgements

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC-Global Research Collaboration grant Task No. 2988.001, and also NSF grants 1527034, 1730158, 1826967, 1830331, 1911095, and 2003277.

Chapter 2, in part, is a reprint of the material as it appears in J. Morris, S. Set, G. Rosen, M. Imani, B. Aksanli, and T. Rosing, “AdaptBit-HD: Adaptive Model Bitwidth for Hyperdimensional Computing” in IEEE International Conference on Computer Design (ICCD), 2021. The dissertation author was the primary investigator and author of this material.

# Chapter 3

# A Rework of the Hyperdimensional Computing Pipeline and Acceleration on FPGA

## 3.1 Introduction

The existing HD computing algorithms [33] have two main challenges: (i) the encoding is computationally expensive, as it requires the computation of thousands (e.g., 10,000) of operations to map each element of data from the original domain to high-dimensional space [45, 46]. For example, our experiments on five practical applications (described in Section 3.7) show that in HD computing the encoding module takes about 79% and 74% of the training and inference time respectively. (ii) In addition, HD computing using binary encoded vectors provides significantly lower classification accuracy. In other words, HD computing requires non-binary (integer) vectors in order to provide acceptable accuracy. However, working with non-binary vectors significantly increases the memory requirement, and the computation complexity of training and inference.

In this chapter, we propose ReHD, a full rework of Brain-Inspired HD computing to make it more hardware friendly and achieve energy-efficient and high-accuracy classification. ReHD introduces a novel encoding module based on random projection with a predictable memory access pattern that can be efficiently implemented in hardware. In contrast to existing HD computing algorithms that increase the size of encoded data by 20$\times$ [33], ReHD is the first HD-based approach which provides data projection with a 1:1 ratio to the original data. In addition, ReHD encodes all data to binary hypervectors, simplifying computation in training and inference. The low memory requirement and computation cost makes ReHD a suitable candidate for embedded devices with limited resources. To address systems that need more control over the trade-off between computational efficiency and classification accuracy, we propose n-bit model quantization. With our new model quantization method, we represent hypervector elements with n-bit integers. To further improve ReHD efficiency, we improve online dimension reduction by intelligently choosing insignificant dimensions to remove.

## 3.2 Related Work & Motivation

Prior work tried to apply the idea of high-dimensional computing to different classification problems such as language recognition, speech recognition, face detection, EMG gesture detection, human-computer interaction, and sensor fusion prediction [32, 33, 45, 34, 22, 24, 35]. For example, work in [46] proposed a simple and scalable alternative to latent semantic analysis. Additionally, work in [45] proposed a new HD encoding based on random indexing for recognizing a text's language by generating and comparing text hypervectors. Work in [47] proposed an encoding method to map and classify biosignal sensory data in high dimensional space. Work in [11, 33] proposed a general encoding module that maps feature vectors into high-dimensional space while keeping most of the original data. Prior work also accelerated HD computing by binarizing the class hypervectors [31, 38], removing dimensions of the class hypervectors [17], or compressing the HD model [36]. [14] extended the idea of binarizing to instead use a ternary model to achieve higher accuracies. Work in [37] also proposed a dynamic dimensionality model to improve energy efficiency.

Prior work also tried to design hardware acceleration for HD computing by mapping its operations into hardware, e.g., in-memory architecture [48, 49, 18, 50, 51, 12, 52, 53], and tried to accelerate HD computing in hardware by binarizing the class hypervectors [31, 38] or removing dimensions of the class hypervectors [17, 17]. Work in [54] designed an FPGA

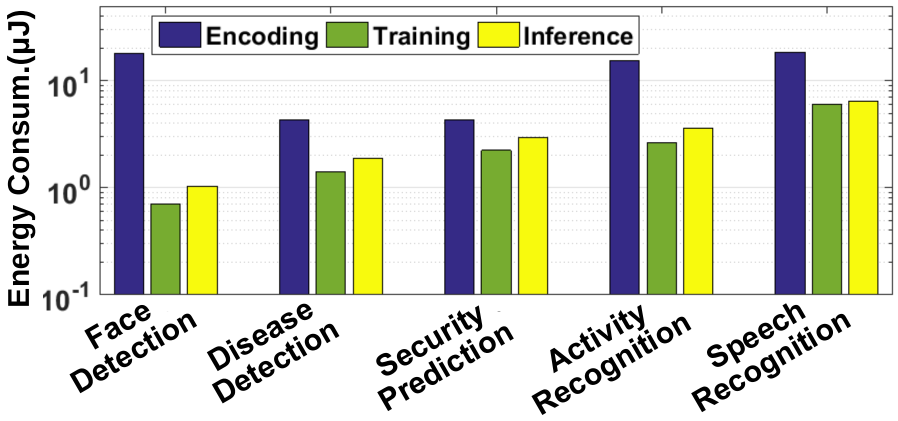


**Figure 3.1.** Energy consumption of HD encoding, training, and inference.

implementation to accelerate HD computation in the binary domain. However, the application of these approaches is limited to simple classification problems such as language recognition [45]. To provide acceptable classification accuracy, all of these approaches have to train the model using non-binary (integer) vectors. However, using non-binary vectors requires a large memory footprint and computation cost in both training and inference.

Model quantization is a widely used technique in machine learning applications to improve energy efficiency. For instance, Google's TPU for performing inference on DNNs utilizes reduced bit representations [27]. Furthermore, [28] proposes a quantization method for SVMs. Model quantization has also been used to reduce the memory requirement for a more efficient hardware design [29, 55]. Work has also been done to adaptively change the precision of the model to reduce the accuracy loss online [56]. [57] proposes a method to use multiple precision levels during inference to achieve a balance between efficiency and accuracy loss. [58] tries to alleviate accuracy loss from quantization by compensating for computational errors. Other methods such as model compression have also been used to improve the energy efficiency of neural networks [30].

In this work, we observe that the existing encoding modules are algorithmically and computationally inefficient. In addition, to get high accuracy, the encoding needs to map data into vectors with integer values which significantly increases the data size [33, 34]. This large memory

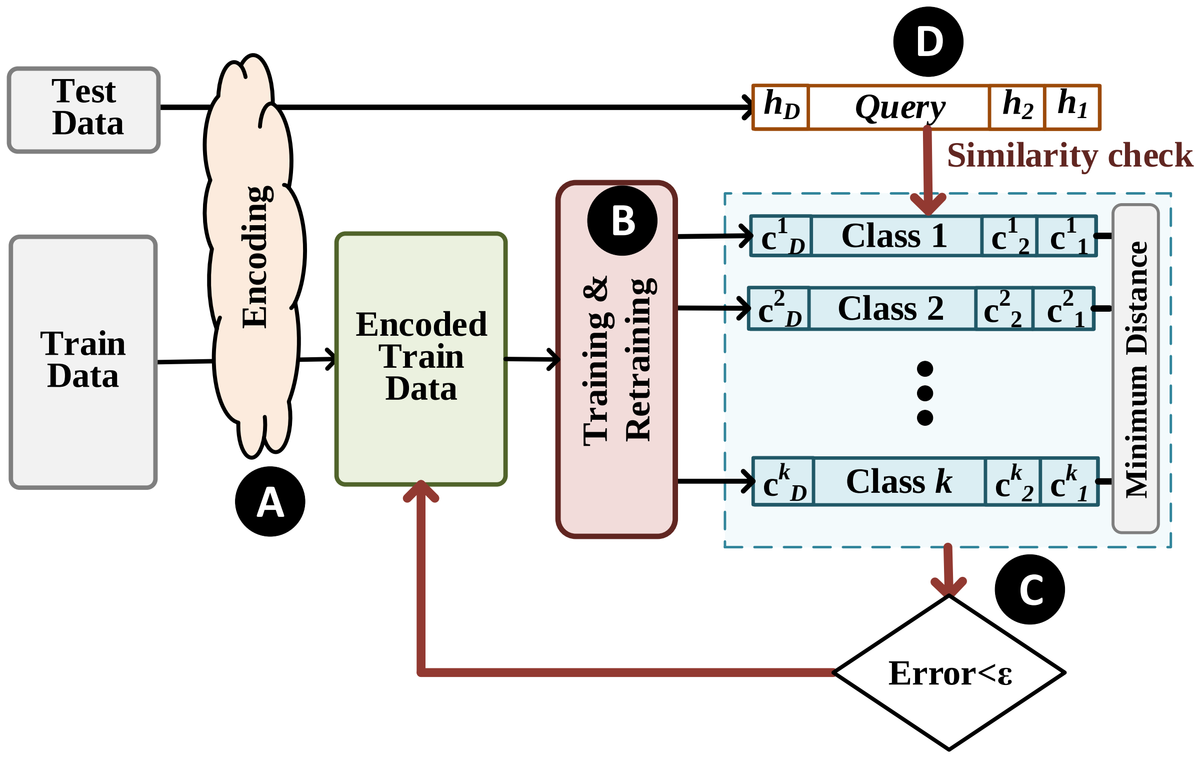


**Figure 3.2.** Overview of how ReHD is constructed and how ReHD performs inference.

requirement is often not available on embedded devices with limited resources. Figure 3.1 shows the energy consumption of encoding, training, and inference (associative search) when running a single data point on five practical applications. Our evaluation shows that the encoding module on average takes 4.7× and 3.8× higher energy than HD training and inference. In this work, we propose a novel encoding approach that (i) significantly reduces the encoding computation cost by introducing computation locality and (ii) provides high classification accuracy while mapping data into binary vectors with much lower dimensionality than existing algorithms.

## 3.3 Encoding with ReHD

In this chapter, we propose ReHD, a novel hardware friendly framework for efficient classification. ReHD consists of three main modules shown in Figure 3.2: encoding, training, and inference. The encoding module maps each data point to binary high-dimensional space. Our encoding has been designed to map the maximum amount of information to high dimensional space with the minimum computation cost. ReHD accumulates every encoded binary training hypervector to create an integer model. This integer model is then used to create a quantized model. ReHD accordingly proposes a training approach that enables the values to stay quantized during training. During inference, cosine similarity has been used as the similarity metric in prior work to achieve the best accuracy in HD computing applications [12]. Quantizing the model enables ReHD inference to be supported using a more efficient n-bit cosine similarity rather than full 32-bit precision. In the following, we explain the details of ReHD functionality.

Figure 3.2*A* shows where ReHD performs the encoding task. Previous encoding schemes are inefficient due to consistent random memory accesses to find the corresponding level hypervector for each feature value. In addition, the amount of computations needed is large and does not take advantage of hardware optimizations like data sparsity[13].

## 3.3.1 Random Projection

We desire a fast and hardware-friendly algorithm that can take a vector of real-valued data and generate a binary code such that the encoding preserves the cosine similarity. Let us assume $\mathbf{A},\mathbf{B} \in \mathbb{R}^n$ are two feature vectors in the original domain with real values. We wish to define an encoding operation $\lambda(*)$ such that:

$$\{\mathbf{X} = \lambda(\mathbf{A}), \mathbf{Y} = \lambda(\mathbf{B})\ ,\ \mathbf{X},\mathbf{Y} \in \{1,-1\}^D\}$$

$$\delta(\mathbf{A},\mathbf{B}) = \delta(\mathbf{X},\mathbf{Y})$$

where $\delta(*)$ is the cosine similarity. Since the cosine angle of binary vectors is determined by how many bits match, the cosine angle and Hamming distance are proportional. This type of encoding can be performed using Locality Sensitive Hash algorithms, such as Random Projection [59]. Let us assume a feature vector $\mathbf{F} = \{f_1,\ f_2, \ldots,\ f_n\}$, with $n$ features ($f_i \in \mathbb{N}$) in original domain. The goal of random projection is to map this feature vector to a $D$ dimensional space vector: $\mathbf{H} = \{h_1,\ h_2, \ldots,\ h_D\}$. As Figure 3.3a shows, random projection generates $D$ dense bipolar vectors with the same dimensionality as original domain, $\{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_D\}$, where $\mathbf{P}_i \in \{-1, 1\}^n$. The inner product of a feature vector with each randomly generated vector gives us a single dimension of a hypervector in high-dimensional space. For example, we can compute the $i-th$ dimension of the encoded data as:

$$h_i = sign(\mathbf{P}_i \cdot \mathbf{F})$$

where *sign* is a sign function which maps the result of the dot product to +1 or -1 values. This type of hashing involves a large amount of multiplications/additions which is inefficient in hardware. For example, to map a feature vector from $n$ to $D$ dimensions, this encoding involves $n \times D$ multiplication and addition operations.

### 3.3.2 Sparse Random Encoding

The efficiency of random projection can be improved by sparsifying each projection vector. Instead of generating dense projection vectors, we can generate sparse projection vectors(Figure 3.3b). Consider $s$ as a sparsity of each projection vector. Then, for each sparse projection vector, only $s\%$ of the vector's elements are randomly generated and the rest are set to zero. For example, if $s = 5\%$, each projection vector only has $0.05 \times n$ non-zero elements. Therefore, each dimension of the encoded hypervector can be computed with only $0.05 \times n$ multiplication/addition operations. Therefore, encoding a single hypervector takes $s \times n \times D$ multiplication/addition operations, compared to $n \times D$ multiplication/addition operations with dense projection vectors. Although the sparsity significantly reduces the number of arithmetic operations, it introduces random accesses to the algorithm, which is hard on the cache and slows down the computation.

### 3.3.3 Locality-based Sparse Random Projection

Here we propose a novel approach that keeps the advantages of a sparse projection matrix, i.e., fewer operations while removing random accesses to make the algorithm more hardware friendly. We propose a locality-based random projection encoding that uses a predictable access pattern. Instead of selecting s% random indices of the projection matrix to be non-zero, we approximate sparse random projection by selecting pre-defined indices to be non-zero. Figure 3.3c shows the structure of the locality-based matrix. Our approach selects the first $s \times n$ of the $\mathbf{P}_1$ vector to be non-zero (indices $[1...s \times n]$). Similarly, $P_2$ projection vector only has $s \times n$ non-zero elements on indices $[2...s \times n - 1]$. Finally, $\mathbf{P}_D$ contains non-zero elements on the last $s \times n$ dimensions. This creates a clear spacial locality pattern that hardware accelerators can take advantage of.

Figure 3.4 shows the overview of ReHD encoding mapping each $n$ dimensional feature vector to a $D$ dimensional binary hypervector. ReHD simplifies the projection matrix to a single dense random projection vector with $D$ bipolar values. Our approach first replicates the feature vector, $\mathbf{F}$, such that it extends to $D$ dimensions, the same as our desired high-dimensional vector. For example, to encode a feature vector with $n = 500$ features to $D = 4,000$ dimensions, we need to concatenate 8 copies of a feature vector together. Then, it generates a random $D$ dimensional projection vector, $P$, next to the extended feature vector (as shown in Figure 3.4). To compute the dimensions of the high-dimensional vector, ReHD takes the dot product of the extended feature vector with each projection vector in an $N$-gram window. The first $N$-gram calculates the dot product of the first $N$ features and $N$ projection vector elements:

$$h_1 = sign(f_1 * p_1 + f_2 * p_2 + ... + f_N * p_N)$$

Similarly, the $N$-gram window shifts by a single position to generate the next feature values. So, we can compute the $i^{th}$ dimension of an encoded hypervector using:

$$h_i = sign(f_i * p_i + f_{i+1} * p_{i+1} + ... + f_{i+N} * p_{i+N})$$

Each step of the $N$-gram window corresponds to a multiplication with a sparse projection vector in the projection matrix. Although this encoding has the same number of computations as

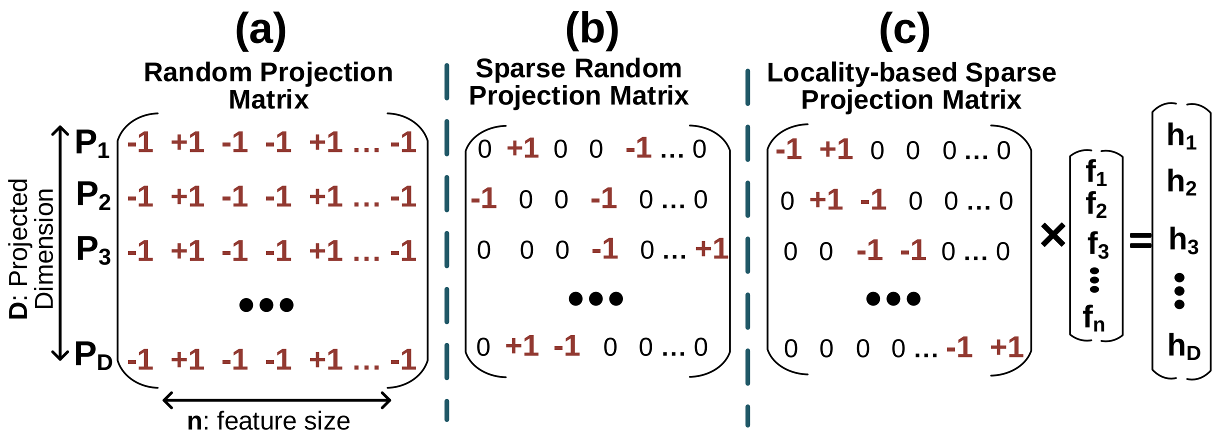


**Figure 3.3.** Random projection encoding using dense, sparse, and locality-based projection matrix.

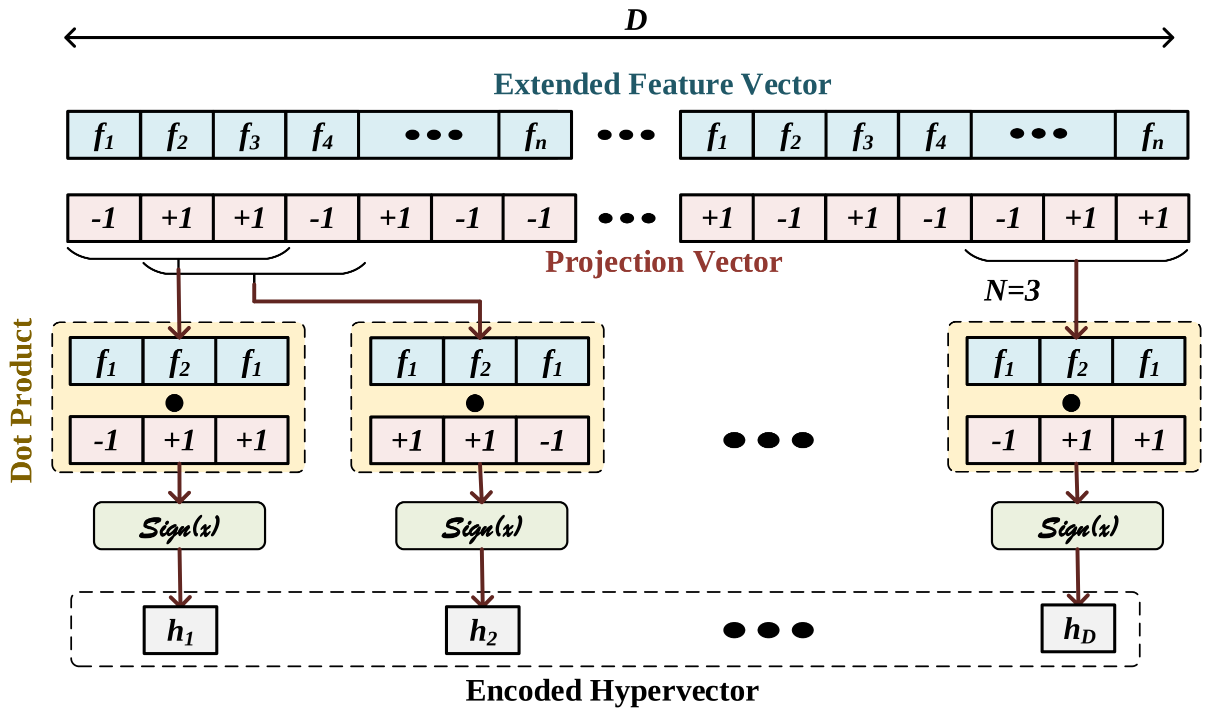


**Figure 3.4.** Locality-based random projection encoding.

sparse random projection, it provides the following advantages: (i) it removes random accesses from the feature selection by introducing spacial locality, which significantly reduces the cost of hardware implementation. (ii) There is an opportunity for computation reuse, as every neighboring dimension shares $N-1$ terms.

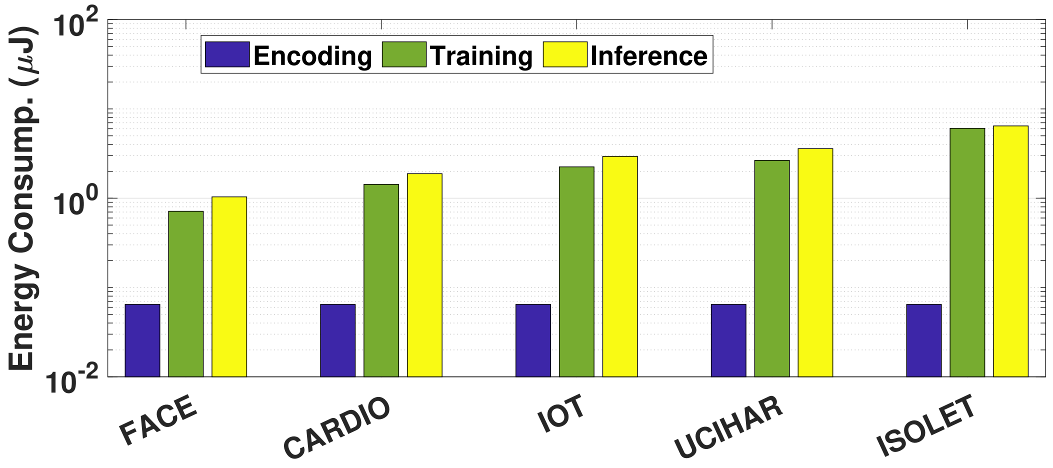


**Figure 3.5.** Energy consumption of HD encoding, training, and inference after utilizing the proposed encoding module.

## 3.4 Training in ReHD

After utilizing our new hardware-friendly encoding, we observe that training and inference are now the energy-intensive parts of the HD algorithm. Figure 3.5 shows the updated energy consumption of encoding, training, and inference (associative search) when running a single data point on five practical applications when utilizing the proposed encoding. Our evaluation shows that training and inference on average take 43% and 55% of the total energy when using the new proposed encoding. This is mainly due to the usage of full precision 32-bit HD models. In this work, we propose a novel approach which (i) allows the HD computing model hypervectors to be represented with n-bit integers, where n ranges from 1 to 32, and (ii) allows for fine-grained control between accuracy and energy efficiency compared to the previous approach of utilizing full 32-bit precision or 1-bit binary models.

.

### 3.4.1 Binary Model Quantization

Figure 3.2*B* shows the functionality of HD Computing during training. Previous work proposed quantization to a binary model for improved speed and efficiency [15].

**Initial Training:** An integer model is first initialized through element-wise addition of all encoded hypervectors in each existing class. Like in Baseline HD Computing, the result is $k$ hypervectors, each with $D$ dimensions, where $k$ is the number of classes. For example, $i^{th}$ class hypervector can be computed as $\mathbf{C_i} = \sum_{\forall j \in class_i} \mathbf{H_j}$. We then binarize each class hypervector from the integer model to create the binary model. We perform this binarization operation by taking the sign bit of each dimension from the accumulated class HVs.

**Retraining:** We train the binarized model by iterating through the training set. Throughout training, we maintain both a binary model and an integer model of the class hypervectors. In a single iteration of model adjustment, HD computing checks the similarity of all training data points, say $\mathbf{H}$, with the class hypervectors in the trained binary model. The data point is assigned to the class with which it has the closest Hamming distance. If a data point is incorrectly classified by the model, HD updates the model by (i) adding the incorrectly classified hypervector to the class the input data point belongs to ($\widetilde{\mathbf{C}}^{correct} = \mathbf{C}^{correct} + \mathbf{H}$), and (ii) subtracting it from the class to which it is wrongly matched ($\widetilde{\mathbf{C}}^{wrong} = \mathbf{C}^{wrong} - \mathbf{H}$). These changes are made to the integer model saved from training because adding to and subtracting from the binary model would drastically change the model. To update the binary model, the updated class hypervectors from the integer model are binarized via the same process described in training.

### 3.4.2 N-Bit Model Quantization

The Binary Model results in faster and more efficient training because the model is represented with integers smaller than 32 bits, but a sharp decline in accuracy often accompanies the increase in speed and efficiency. The binary model quantization, where we represent the dimensions of model hypervectors with 1 bit, maximizes efficiency but also yields the lowest classification accuracy. This forces us to choose between two extremes: low accuracy but high efficiency (binary), and high efficiency but low accuracy (32-bit). To solve the problem of having to choose between two extremes, we can achieve more granular control over this trade-off by representing dimensions with n bits, where n ranges from 1 to 32. Hence, we no longer have to choose between 1-bit and 32-bits. As we represent dimensions with more bits, we increase the accuracy but make classification less efficient.

**Initial Training:** The initial training for model quantization is very similar to the initial training for the binary model, as the integer model is created through the same process. The training process for model quantization diverges from that of past work after the initial addition, as rather than an adjacent binary model, we create an adjacent n-bit model. To represent the dimensions with n-bits, we utilize the integer model and clip all dimensions that fall outside of the range of integer values we can represent with n bits. For an n-bit model quantization, we can represent the range $[-2^n, 2^n - 1]$. Therefore, for all elements of class hypervectors, we discard any overflow beyond this range.

**Retraining:** The retraining process for model quantization is also similar to the retraining process for the binary model. Throughout training, we store both an integer model and an n-bit representation model of the class hypervectors. Model quantization performs model adjustment by iterating through the training dataset, making changes to the integer model, and reflecting those changes to the n-bit representation model similar to the initial training process.

### 3.4.3 Model Quantization Inference

After training and retraining, the HD model can now be used for inference (Figure 3.2*D*). The input data is encoded as a binary *query* hypervector. Model quantization then computes the similarity between the binary *query* hypervector and each n-bit class hypervector. 1-bit model quantization computes similarity using Hamming distance and n-bit model quantization using cosine similarity over n bits. The input data is classified into the class whose hypervector it is most similar to. As the number of bits used to represent dimensions increases, so does the inference accuracy, but the training, retraining, and inference processes become more complex.

## 3.5 Online Dimension Reduction

The gradient descent during retraining gives equal weight to all features when the data is binarized. This includes noisy, low strength features as well as features with high intra-class differences. In fact, gradient descent moves the hyperplane in the direction of these features with equal strength as the important features, which results in possible overfitting. The challenge is to amplify the learning rate of ”significant” dimensions, while not amplifying the learning rate of ”insignificant” or ”noisy” features. Online dimension reduction attempts to remove insignificant ”noisy” dimensions from the model to improve energy efficiency. We can define insignificant dimensions using either high absolute values or low variance as a metric. We define $s$ as the sparsity level denoting what percentage of the dimensions will be removed, regardless of which metric is used, dropping the $s\%$ most insignificant dimensions from the model, results in an efficiency improvement of approximately $s\%$.

We drop the $s\%$ most insignificant dimensions from the model rather than using a thresholding technique because the range of values varies between datasets, as it depends on how many samples there are. Datasets with larger amounts of samples result in significantly larger accumulated dimensions compared to those with fewer samples. This is because of how the initial model is created by accumulating all the encoded samples. Therefore, with more samples, the dimensions that agree across all samples will accumulate much higher values. However, we can account for this difference in datasets by removing a proportion rather than an absolute threshold.

To use high absolute values as a metric of insignificance, we first compute the element-wise addition of all binarized sample hypervectors and examine the sum of each dimension. Because all training hypervectors are initially binary with +1 or -1, dimensions with a very high sum indicate that most training instances have a +1 for that dimension, and dimensions with a very low sum indicate that most training instances have a -1 for that dimension. Such dimensions have low differentiation between training instance data points and low differentiation between classes, so we declare dimensions with high absolute value sums to be ”insignificant”, as Figure 3.6 shows. This is because to distinguish the classes from each other, we want to emphasize their differences and not their similarities.

We can choose insignificant dimensions more intelligently by using low variance as a metric of noise and low-strength. Before the encoded hypervectors are binarized by taking the sign bit, we calculate the variance of each dimension. The dimensions with low variances indicate that those dimensions contain mutual information among all the samples, and thus do not help the model differentiate between classes. Dimensions with high variance are declared ”significant’, while dimensions with low variances are ”insignificant”. As stated above, we must emphasize inter-class differences rather than similarities. This method drops the dimensions with the lowest variances from the model as shown in Figure 3.7. Comparing the distributions of the variances shown in Figure 3.7 and the distributions of absolute values in Figure 3.6, we can see that the variance metric can cluster and identify more insignificant dimensions compared to the absolute value metric. Thus, using variance as the metric to determine insignificant dimensions is able to reduce dimensionality further with less accuracy loss than using high absolute values.

## 3.6 FPGA Acceleration

ReHD can be accelerated on different platforms such as CPU, GPU, FPGA, or ASIC. FPGA is one of the best options as ReHD computation involves bitwise operations among long vector sizes. General strategies of optimizing the performance of ReHD are (i) using a pipeline and partial unrolling on low levels (dimension levels) to speed up each task and (ii) using dataflow design on a high level (task level) to build a stream processing architecture that lets different tasks run concurrently. In the following, we explain the functionality of ReHD in encoding, training, retraining, and inference phases.

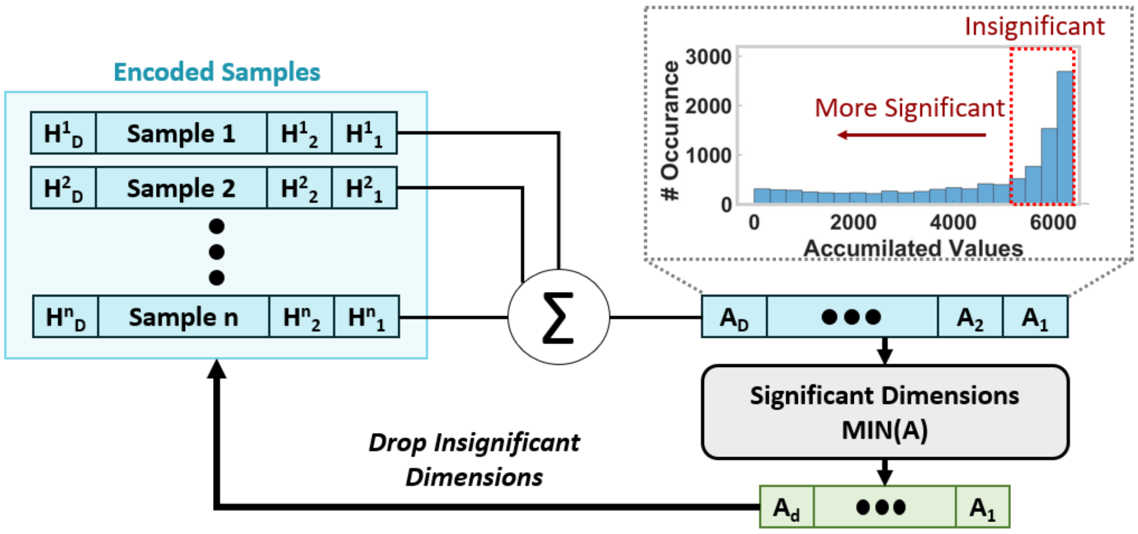


**Figure 3.6.** Online dimension reduction with absolute value.

### 3.6.1 Encoding Implementation

As we explained in Section 3.3.2, we used the locality-based random projection encoding to implement the encoding module. Due to the sequential and predictable memory access patterns as well as the abundance of binary operations, this encoding approach can be implemented efficiently on an FPGA. In the hardware implementation, we represent all $\{-1,+1\}$ values with $\{0,1\}$ respectively. This enables us to represent each element of the projection vector using a single bit. Figure 3.8a shows the hardware implementation of the ReHD encoding module. The encoding process includes reading a feature vector from off-chip DDR memory and generating a binary hypervector from them.

Calculating the inner product of a feature vector and a projection vector, $P \in \{1,-1\}^D$, can be implemented with no multiplications. Each element of the projection vector decides the sign of each dimension of the feature vector in the accumulation of the dot product. Thus, the dot product can be simplified to the addition/subtraction of the feature vector elements. Right after the encoding, the hypervectors are used for initial model training. We also need to store the encoded hypervectors for retraining. However, the FPGA does not have enough BRAM blocks

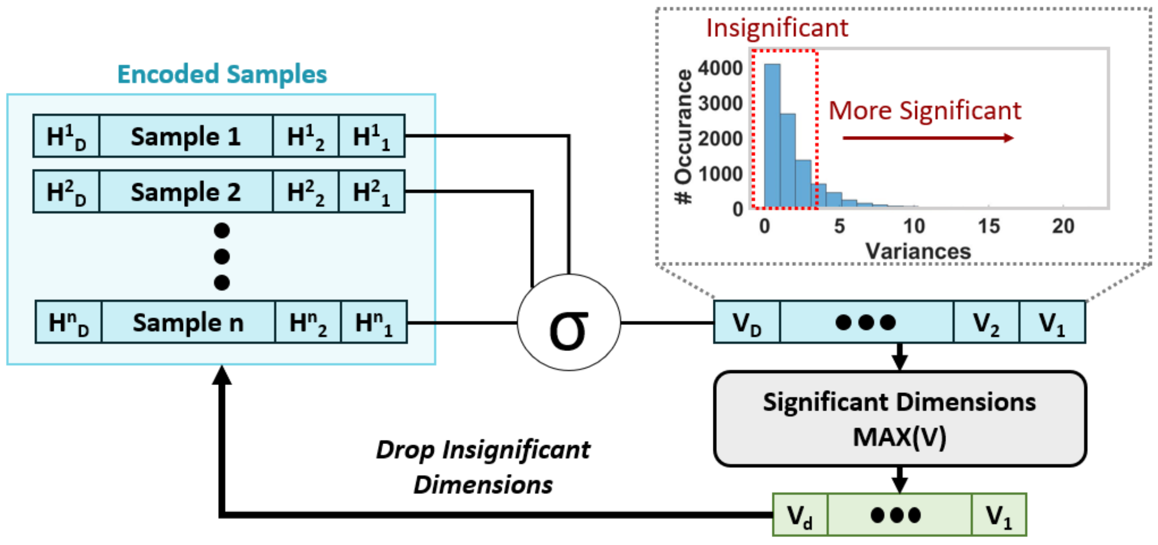


**Figure 3.7.** Online dimension reduction with variance.

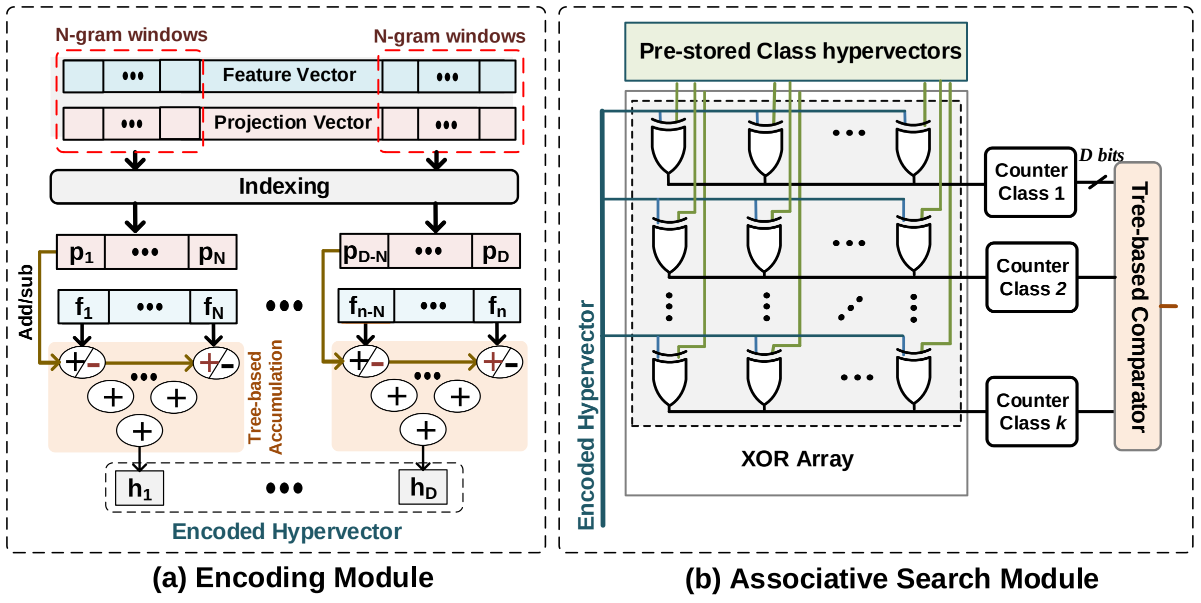


**Figure 3.8.** FPGA implementation of the encoding and associative search block.

to store all encoded hypervectors, so, our design stores them into DDR memory.

### 3.6.2 Training Implementation

**Initial Training:** Like previously, initial training for ReHD with model quantization is a single-pass process. The training module accesses the encoded hypervectors and accumulates them in order to create a hypervector representing each class. When the training module accumulates the encoded hypervector to one of the class hypervectors, the encoding module maps the next training data into high-dimensional space, improving data throughput by increasing resource utilization. After going through all of the training data, our implementation creates an n-bit quantized representation of the model. We iterate through all hypervectors in the training and test datasets, and clip values greater than $2^n$-1 to $2^n$-1 and values less than $-2^n$ to $-2^n$. Finally, the quantized n-bit model is stored in the BRAM blocks to be used for inference or retraining.

**Retraining:** The retraining phase first sequentially reads already encoded training hypervectors from the off-chip memory in batches to help hide the latency of reading from the off-chip memory. This is necessary as each read has a latency of about $15ns$, which would slow down the retraining process. Next, we check the similarity of each data point with all trained class hypervectors. Each data point gets a tag of a class in which it has the highest Hamming distance (1-bit quantized model) or cosine similarity (n-bit quantized models with $n \neq 1$). In the case of misclassification, ReHD needs to update the model by adding and subtracting a data hypervector with two class hypervectors as defined before.

### 3.6.3 Inference Implementation

After the retraining, the quantized ReHD model has a stable model that can be used in the inference phase. The encoding module is integrated with the similarity check module as the entire inference part. Each test data point is first encoded to high-dimensional space using the same encoding block explained in Section 3.6.1. Next, the quantized ReHD model checks the cosine similarity of the data point with all pre-stored class hypervectors, in order to find a class with the highest similarity. One unique advantage of our approach is its capability to enable online training during the inference phase. Our implementation stores two HD models: one with integer values used for retraining and an n-bit quantized model which is used to perform the classification task. ReHD quantizes the integer model to an n-bit model periodically to update the inference model. While the previous model computes similarity with Hamming distance, the updated quantized ReHD model computes cosine similarity. cosine similarity with n-bit quantized models may seem much more energy intensive than utilizing Hamming distance for binary models because cosine similarity involves multiplications. However, we can use the same optimization in encoding that removed the multiplications between the feature vector and a projection vector to remove the multiplications between the encoded query hypervector and n-bit quantized class hypervector. This is because each element of the encoded query hypervector is binary. Each element of the query hypervector decides the sign of each dimension of the feature vector in the accumulation of the dot product step of cosine similarity. Although Hamming distance is still faster and more computationally efficient, cosine similarity results in higher accuracy when we represent the dimensions of class and instance with hypervectors with *n* bits rather than 1-bit.

## 3.7 Evaluation

### 3.7.1 Experimental Setup

We implemented ReHD training, retraining, and inference in both software and hardware. In software, we implemented ReHD with Python. In hardware, we fully implemented ReHD using Verilog. We verify the timing and the functionality of the models by synthesizing them using Xilinx Vivado Design Suite [39]. The synthesis code has been implemented on the Kintex-7 FPGA KC705 Evaluation Kit. We evaluated the efficiency of the proposed ReHD on four practical classification problems listed below: Speech Recognition (ISOLET) [40], Activity Recognition (UCIHAR) [41], Face Detection (FACE) [42], Cardiotocography (CARDIO) [43],

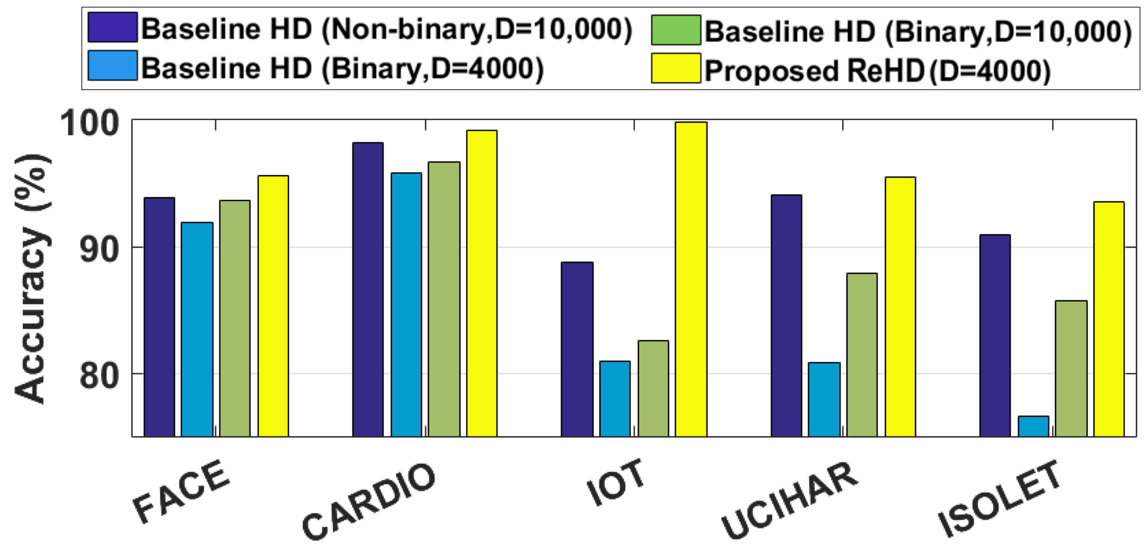


**Figure 3.9.** Classification accuracy of ReHD and the baseline HD using binary and integer models.

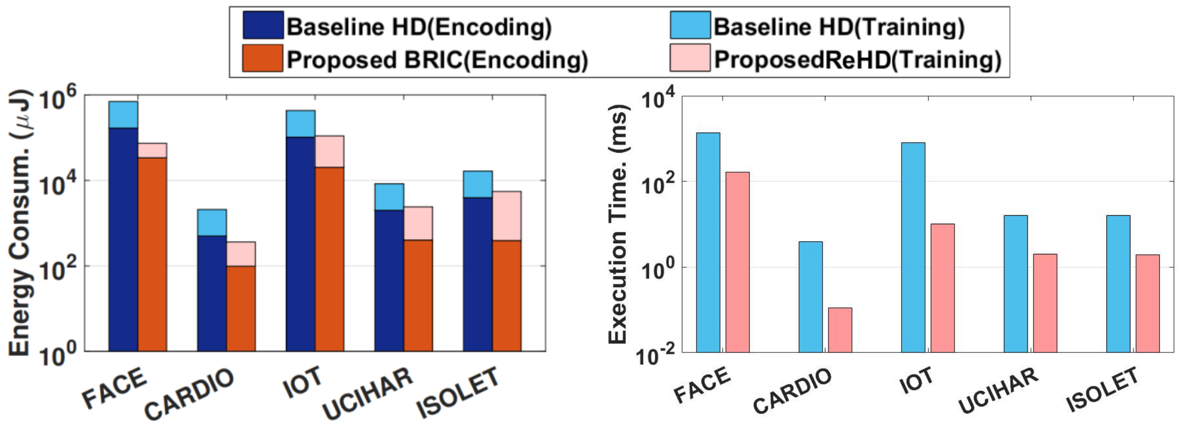


**Figure 3.10.** Energy consumption and execution time of ReHD and the baseline HD during training.

and Attack Detection in IoT systems (IoT) [44]. We compare ReHD with, baseline HD, an FPGA implementation of [11].

### 3.7.2 Comparison With Other State-of-the-Art Light-Weight Classifiers

Table 3.1 compares HD computing with other light-weight classifiers including support vector machines (SVM), gradient boosting classifiers (Boosting), perceptrons, and multi-layer perceptrons (MLP) in terms of accuracy and training/inference efficiency. All results are reported

**Table 3.1.** CPU-based comparison of HD and other classifiers.

| | **SVM** | **Perceptron** | **MLP** | **HD** |
|---|---|---|---|---|
| **Training Exe.($ms$)** | 480.3 | 320.2 | 1,229.2 | 168.3 |
| **Testing Exe.($\mu s$)** | 813.7 | 102.4 | 286.2 | 59.4 |

when applications are running on an embedded device (Raspberry Pi 3) using an ARM Cortex A53 CPU. Our evaluation shows that HD computing can provide comparable accuracy to algorithms such as SVM and MLP. In terms of efficiency, HD computing can provide much faster computation in both training and testing. For example, in a CPU implementation, HD computing is 7.3× and 4.8× (2.9× and 13.6×) faster than MLP (SVM) during training and testing respectively. These results demonstrate that HD computing is the clear choice among light-weight classifiers for low-powered energy efficient machine learning.

### 3.7.3 ReHD Accuracy and Memory Requirement

Figure 3.9 compares the impact of hypervector dimensions on the classification accuracy of ReHD and the baseline HD computing encoding [33]. As we explained, ReHD always encodes data points into $D$ binary dimensions. However, for the baseline HD computing encoding, we consider two cases when HD encodes data points to binary and integer domains. Our results in Figure 3.9 indicate that ReHD requires significantly fewer dimensions to provide the same accuracy as the baseline. For example, ReHD using $D = 4,000$ binary dimensions provides the same accuracy as the baseline with $D = 10,000$ integer dimensions. In addition, the baseline with a binarized model provides significantly lower accuracy than ReHD and the baseline with an integer model. ReHD is on average $11.5\%$ more accurate than the baseline using a binary encoding and binary model. However, as we explore in Section 3.7.5, ReHD is able to achieve even higher accuracies when utilizing n-bit quantization compared to binary quantization.

Here we compare ReHD and the baseline in terms of the training memory requirement.

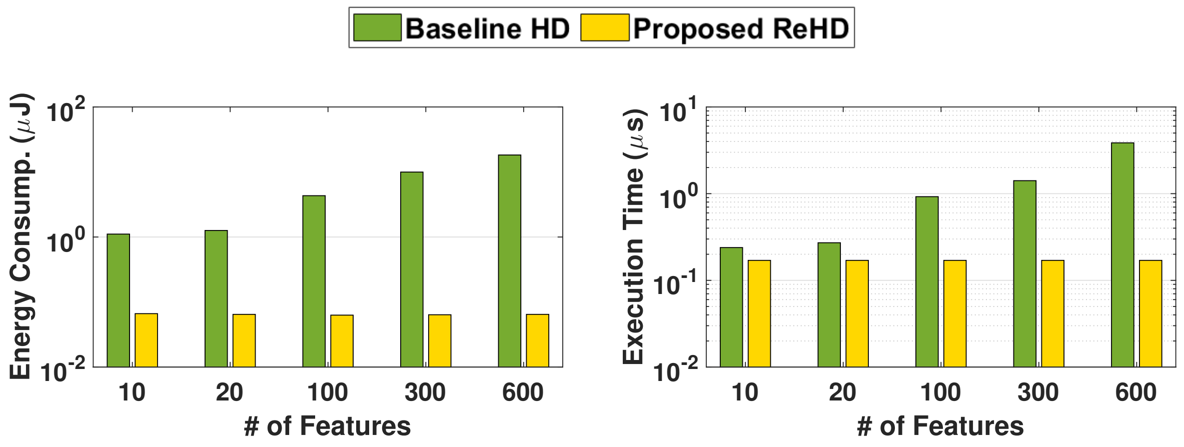


**Figure 3.11.** Scalability of the encoding module in ReHD and the baseline HD with the feature size.

As we explained in Section 3.4.1, the baseline/ReHD store all encoded training data in memory. Going into high dimensional space intuitively means increasing the data size, since we map each feature vector from $n$ into $D$ dimensional space, where $D >> n$. Let us assume a feature vector with $n = 500$ integer features. For the baseline with integer values, the data size increases by approximately 20×. Even the baseline with a binary encoding ($D = 10,000$) increases the data size by 2.5×, while it provides much lower accuracy. In contrast, the proposed ReHD encodes data points to a much lower dimensionality, e.g., $D = 4000$, in order to provide the same accuracy as the baseline. Our evaluation shows that ReHD can ensure 1:1 ratio of high-dimensional data to original data, while providing the same accuracy as baseline HD [33], proving that ReHD is more capable to run on embedded devices with limited memory.

### 3.7.4 Hardware Efficiency

We compare the efficiency of ReHD with the state-of-the-art HD computation algorithms on a Kintex-7 FPGA. To have a fair comparison, we consider an optimized implementation of the baseline [33], running on the same architecture as ReHD (explained in Section 3.6).

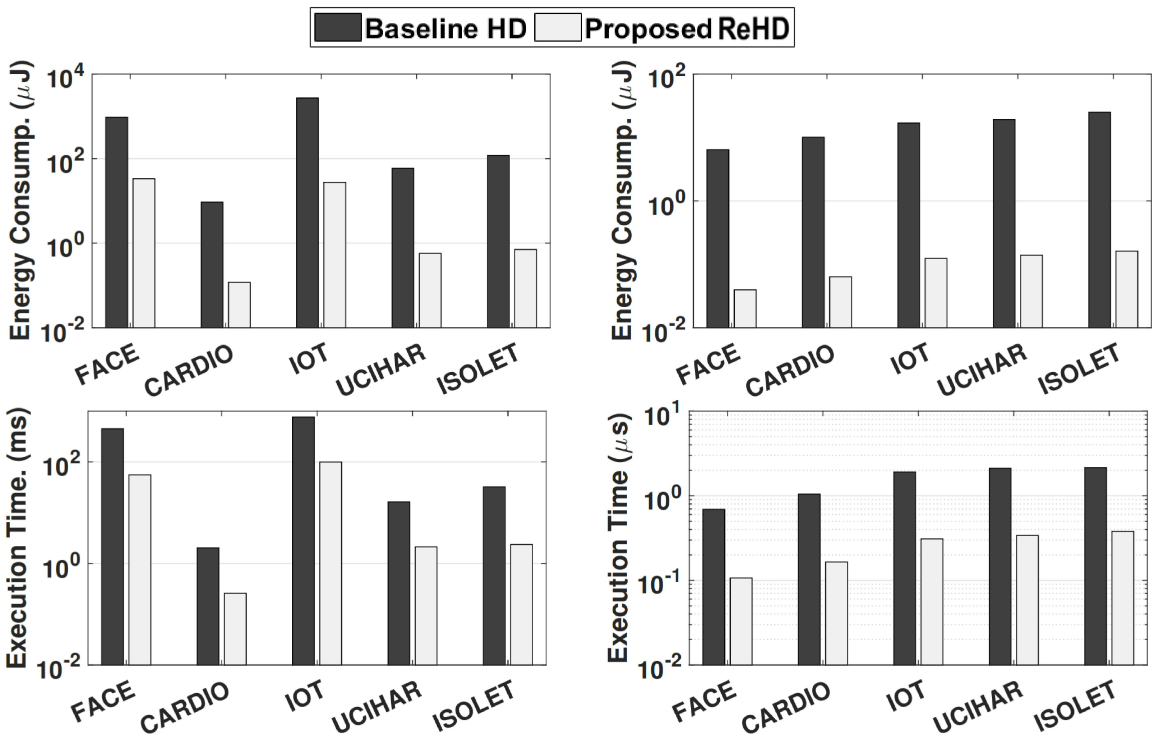
**Encoding & Training:** Due to the predictable memory access pattern and lower ReHD dimensionality, ReHD encoding can process with higher efficiency as compared to the baseline.

For instance, to get maximum accuracy, the baseline needs to work with $D = 10,000$ dimensionality while ReHD can provide the same accuracy with $D = 4,000$. Figure 3.11 shows the scalability of ReHD and the baseline efficiency in terms of the feature size. Our evaluation shows that the execution time of the baseline increases with the number of features, while it takes the same time for ReHD to encode any size feature vector. For applications with 600 features, ReHD provides 282× more energy efficiency and a 22.7× speed up as compared to the baseline.

In training, to create class hypervectors, the baseline accumulates integer hypervectors, while ReHD training accumulates binary hypervectors. Figure 3.10 compares the energy consumption and execution time of ReHD and the baseline during initial training. The results are reported when both designs encode and train the model in a pipeline structure. For the baseline, encoding dominates the execution time, thus the training execution hides under the encoding module. However, in ReHD, the encoding can process faster than the training, thus the training is the bottleneck of the execution time (as it is shown in Figure 3.10). Our evaluation shows that ReHD can provide 64.1× more energy efficiency and a 9.8× speed up as compared to the baseline during training.

**Retraining/Inference Efficiency:** ReHD stores all encoded hypervectors in order to perform iterative retraining. The existing HD computing algorithms map data points to integer values, where each encoded data is around 20 times larger than the data in the original domain. During retraining, the FPGA needs to sequentially access the encoded values which are prestored on off-chip memory. The limited memory bandwidth between the off-chip memory and the FPGA BRAM blocks significantly slows down the baseline computation during retraining. In contrast, ReHD maps the training data to lower dimensions, where each dimension can be represented using a binary value. This enables ReHD to speed up the retraining by loading hypervectors faster than the baseline.

During inference and retraining, HD checks the similarity of each encoded hypervector with all existing class hypervectors. To achieve a high classification accuracy, the existing HD computing algorithms generate an integer model. Therefore, they require the use of an expensive



(a) Retraining (b) Inference

**Figure 3.12.** Energy consumption and execution time of ReHD and the baseline HD running (a) a single retraining iteration, and (b) a single query at inference.

similarity metric such as cosine to find the most similar class. In contrast, ReHD performs the similarity check with Hamming distance. Figure 3.12 shows the energy consumption and execution time of the FPGA accelerating a single retraining iteration and a single query during inference. The results show that ReHD can achieve on average a 61.6× energy efficiency improvement and a 7.9× speed up as compared to the existing HD computation algorithms. Similarly, in inference, the FPGA implementation of ReHD can achieve on average a 43.8× energy efficiency improvement and a 6.1× speed up running a single query (Figure 3.12b).

### 3.7.5 Model Quantization Trade-off

In Figure 3.13, we explore the impact of representing the HD Computing model with bit lengths ranging from 1 to 32 on quality loss. Due to significant information loss when converting to a binary hypervector, 1-bit model quantization, which computes with binary hypervectors,

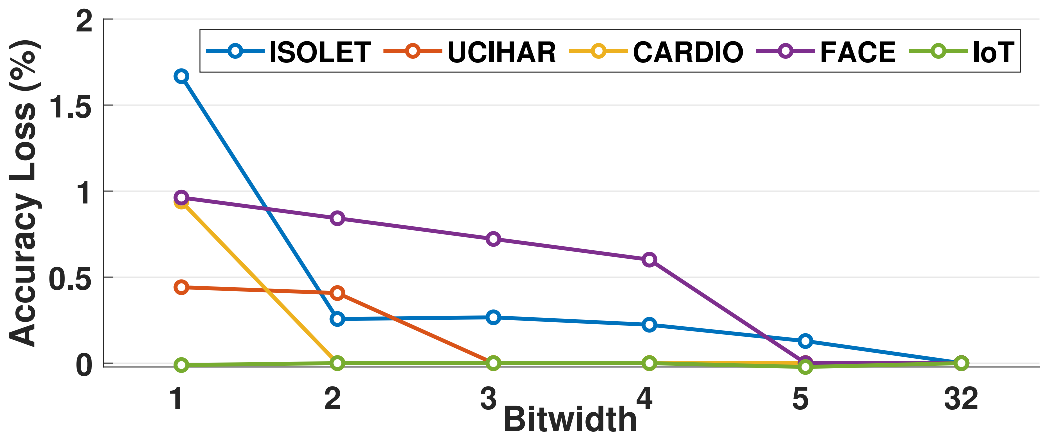


**Figure 3.13.** Accuracy loss of ReHD utilizing n-bit model quantization.

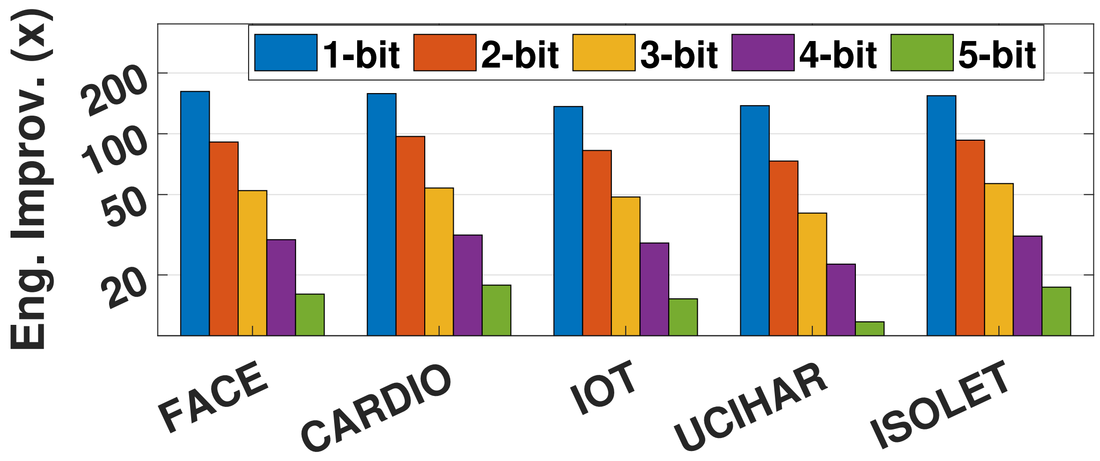


**Figure 3.14.** Energy improvement of ReHD utilizing n-bit model quantization normalized to a 32-bit integer model.

yielded the lowest inference accuracy. 1-bit quantization leads to an accuracy loss of up to 1.7%. However, because the 1-bit model quantization enables using Hamming distance as the similarity function, it is the most efficient quantized model. Figure 3.14, shows the energy improvement of n-bit model quantization over a 32-bit model. The 32-bit model uses the same encoding method as the n-bit models proposed in ReHD. The only difference is that there is no quantization during training and retraining. 1-bit model quantization results in 150$\times$ less energy consumption as compared to a 32-bit model. Therefore, 1-bit model quantization is most useful in scenarios when we primarily prioritize computational efficiency, such as on very low-resource devices. We also primarily prioritize computational efficiency when the classification task is trivial, as is the case with the IoT and UCIHAR datasets.

In scenarios where resources are constrained, but high accuracy is still required, larger bitwidth model quantization is required. By allowing for less efficiency in training and inference, higher bit models allot higher inference accuracy. Using larger bit widths, hypervector dimensions take on an exponentially larger range of values, allowing for more information to be preserved. Larger bit widths yield better inference accuracy, but at the cost of less efficiently than 1-bit model quantization. This is because we have to use cosine similarity as our similarity metric, which is much more expensive than Hamming distance. Larger bitwidth model quantization is useful for datasets that are sufficiently complex that a certain number of information needs be preserved, such as for ISOLET and FACE. On ISOLET, 1-bit model quantization achieves 1.8% lower accuracy than the full 32-bit model. However, but just increasing to a 2-bit model, we are able to reduce the quality loss to 0.25% and use 93× less energy. In our experiment, models which represented hypervectors with 5 or more bits performed with comparable accuracy to models which represented hypervectors with 32 bits. Representing hypervectors with more than 5 bits is more computationally expensive, but yields no accuracy increase. Therefore, by utilizing 5-bit model quantization, we can achieve on average, 15× less energy consumption at no accuracy loss.

### 3.7.6 Online Dimension Reduction

Table 3.2 compares the online dimension reduction techniques of (i) computing the element-wise sum of training hypervectors and removing dimensions with high absolute value sums (ABS) and (ii) computing the variance across all dimensions and removing dimensions with low variance (VARIANCE). The values compute the average quality loss(accuracy drop) over the five datasets described in Section 3.7.1. More directly, Table 3.2 shows the impact of each dimension reduction technique on classification accuracy. When using ABS as a metric of insignificance, our results indicate that dropping 20% of ”insignificant” dimensions slightly

**Table 3.2.** Average change in classification accuracy due to online dimension reduction.

| **Dimension Reduction** | **20%** | **40%** | **60%** | **70%** | **80%** | **90%** | **95%** |
|---|---|---|---|---|---|---|---|
| *ABS* | +0.38% | 0% | -0.54% | -4.1% | -5.64% | -9.6% | -14.2% |
| *VARIANCE* | +0.4% | 0% | 0% | 0% | -0.3% | -0.8% | -4.4% |

improves accuracy because we remove noisy features. As listed in Table 3.2, dropping up to 60% of ”insignificant” dimensions almost no impact on accuracy, but dropping further dimensions will lead to a decline in accuracy because we begin to drop significant dimensions.

Dropping dimensions with low intra-class differences allow for a more intelligent selection of ”insignificant” dimensions than summing all training hypervectors and dropping dimensions with high absolute values. With ABS, we were able to drop 70% dimensions before we started losing significant dimensions. But since VARIANCE selects dimensions to drop more intelligently, we can drop the 90% most insignificant dimensions with only a 0.3% average loss in accuracy as a result, meaning we improve training efficiency by 90% with only a negligible decline in accuracy. The energy efficiency improves proportionally with the dropped dimensions because operations in HD are done with hypervectors. Therefore, by reducing the dimensionality of all hypervectors, all operations reduce in complexity. When we drop more than 90% dimensions, we begin to drop too many significant dimensions and lose a significant amount of accuracy.

Figure 3.15b shows the classification probability over an image, where yellow and blue colors indicate low and high face probability respectively. The results show that ReHD working with $D = 4,000$ dimensions can perfectly detect the faces in the image. ReHD in lower dimensionality after online dimension reduction has lower quality and detects ”non-face” regions. Online dimension reduction improves ReHD efficiency linearly during both retraining and inference. For example, an 80% dimension reduction results in approximately 80% energy efficiency improvement and a 5$\times$ speed up while providing less than 0.3% quality loss as compared to ReHD with full dimensionality.

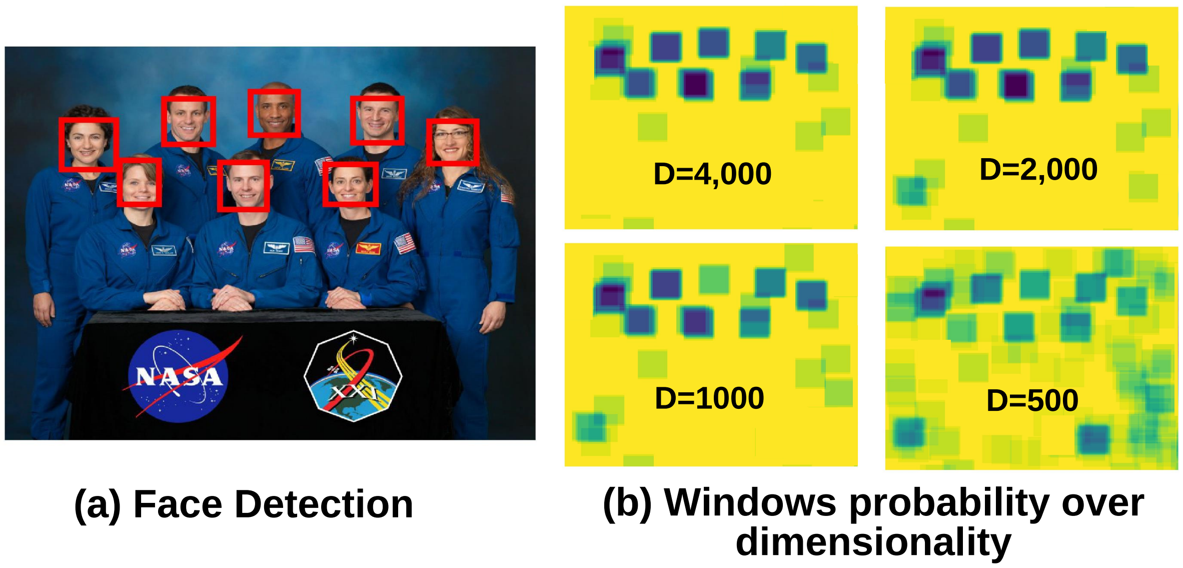


**Figure 3.15.** Visualization of ReHD face detection accuracy over different dimensionality.

## 3.8 Conclusion

In this chapter, we propose ReHD, a novel HD computing framework that significantly improves the computation efficiency of HD computing. ReHD exploits the predictable memory access of our proposed encoding to design an efficient encoding approach that maps data into binary hypervectors. ReHD enables quantized training and retraining on the encoded hypervectors and simplifies the inference similarity metric. N-bit model quantization, allows us to represent our model hypervectors with n-bits where n ranges from 1 to 32, whereas previously designs chose between 1-bit or 32-bit representations. This enables more granular control over the trade-off between model classification accuracy and efficiency. We additionally implemented a dimension reduction technique that removes unnecessary dimensions to further improve the efficiency of ReHD. We also designed a fully pipelined FPGA implementation to accelerate ReHD. Our evaluations show that ReHD can achieve 64.1$\times$ and 9.8$\times$ (43.8$\times$ and 6.1$\times$) energy efficiency and speed up as compared to the baseline during training (inference) while providing the same classification accuracy. In the next chapter, we focus on extending HDC to support multi-label classification.

## 3.9 Acknowledgements

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, in part by SRC-Global Research Collaboration grant Task No. 2988.001, and also NSF grants 1527034, 1730158, 1826967, 1830331, 1911095, 2003277, and 2003279.

Chapter 3, in part, is a reprint of the material as it appears in J. Morris, Y. Hao, R. Fernando, M. Imani, B. Aksanli , T. Rosing, “Locality-based Encoder and Model Quantization for Efficient Hyper-Dimensional Computing”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021. The dissertation author was the primary investigator and author of this material.

# Chapter 4

# Extending Hyperdimensional Computing Applications to Support Multi-Label Classification

## 4.1 Introduction

Recent work has been done to extend HDC applications beyond classification. For instance, HDC has recently been extended to support clustering [22]. Furthermore, HDC has been extended to support recommender systems for the first time [23]. There is also work on using HDC for DNA classification [24]. However, there has been no work yet on mapping HD computing to multi-label classification tasks.

While HD provides improvements in performance and energy consumption over conventional machine learning algorithms, it still involves fetching each and every data from memory/disk and processing it on CPUs/GPUs. This is exaserbated by the fact that HD expands the dimensionality of the input data into high dimensional space. This massive amount of data needed for HD cannot always fit into the memory. Recent work has introduced computing capabilities to solid-state disks (SSDs) to process data in storage [60, 61, 62, 63, 64]. This not only reduces the computation load from the processing cores but also processes raw data where it is stored. HD computing has compelling properties for efficient hardware acceleration in flash. For instance, HD is highly parallelizable with $D = 10,000$ dimensions where each dimension is independent. Furthermore, HD is comprised of simple operations such as addition, multiplication, and comparisons. With these two properties, HD computing is a prime candidate for acceleration in flash.

In this chapter, we design a new Multi-label HD computing in storage system. Our system efficiently accelerates the data-intensive steps of HD, encoding and training, in 3D storage, thus, making it possible to run multi-label classification with HD in the IoT domain. We propose two different mappings of HD to multi-label classification, Power Set HD and Multi-Model HD. Power Set HD, transforms the multi-label problem into classical classification by creating a new class for each label combination. Multi-Model HD that creates a binary classification model for each possible label. Our evaluation shows that Multi-Model HD achieves, on average, 47.8$\times$ higher energy efficiency and 47.1$\times$ faster execution time while achieving 5% higher classification accuracy as state-of-the-art light-weight multi-label classifiers such as multi-label kNNs. Power Set HD achieves 13% higher accuracy than Multi-Model HD, but is 2$\times$ slower. Using our in-3D-flash acceleration, we further improve the energy efficiency of Multi-label HD training by 228$\times$ and reduce the latency by 610$\times$.

## 4.2 Related Work

### 4.2.1 Hyperdimensional Computing

Prior work tried to apply the idea of high-dimensional computing to different classification problems such as language recognition, speech recognition, face detection, EMG gesture detection, human-computer interaction, and sensor fusion prediction [32, 34, 22, 24]. Additionally, work in [45] proposed a new HD encoding based on random indexing for recognizing a text's language by generating and comparing text hypervectors. Work in [47] proposed an encoding method to map and classify biosignal sensory data in high dimensional space. Work in [11] proposed a general encoding module that maps feature vectors into high-dimensional space while keeping most of the original data. There is no work to date that handles multi-label classification in HD.

### 4.2.2 Multi-label Classification

Prior work applied problem transformation methods to transform multi-label classification problems into multiple single-label classification problems [65, 66]. The most widely used transformation method is PT3. PT3 combines each different set of labels into a single label so that the new label set $L^{'}$ is the power set of the old label set $L$. For a dataset with three binary labels, the new label set would be 000, 001, 010, 011, 100, 101, 110, 111. This causes an exponential increase in the number of labels in the dataset. This transformation method is popular for other light-weight classifiers as their complexities mostly scale with the number of features and not with the number of labels. However, in HD computing, the complexity of inference does scale with the number of labels. Therefore, in this chapter we propose a new Multi-Model transformation method that is designed for scalable HD computing.

### 4.2.3 Hardware Acceleration

**HD Acceleration on other Platforms:** Prior work tried to design different hardware accelerators for HD computing. This includes accelerating HD computing on existing FPGA, ASIC, and processing in-memory platforms [67]. However, these solutions do not scale well with the number of classes and dimensions, primarily due to the data movement issue. Therefore, a new solution is needed that can scale with the dimensionality and number of classes. ISC is a promising acceleration architecture in this aspect. **Computing in 3D Flash:** The current 3D flash-based storage systems suffer from slow flash array read latency and storage to host I/O latency. To alleviate these issues prior work introduced in-storage computing (ISC) architectures [62]. These works exploit the embedded cores present in the SSD controller to implement ISC. Another set of work in [61, 68] used ASIC accelerators in SSDs. The work in [63] proposed a full-stack storage system to reduce the host-side I/O stack latency. While these works propose single-level computing in storage, [64] on the other hand exploited computing at flash die and in top level accelerator to provide multi-layer computing. It also allows for high parallelism in computation. In this work, we adapt the ISC design in [64] to enable multi-label HD in 3D flash.

## 4.3 Multi-label Classification with HD

Multi-label classification is the problem of finding a model that maps inputs $x$ to binary vectors $y$, and each element in $y$ is a label that is assigned a value either 0 or 1. This is in contrast to single-label classification, where y is a single value, not a vector of labels. Although HD computing performs well for single-label classification tasks, we can't directly apply it to solve multi-label classification problems, as only one label output is chosen. Therefore, we transform the multi-label problem into a single-label problem and then modify the HD computing algorithm to solve the multi-label classification problem. We propose two different mappings of HD to Multi-label HD. The first, Power Set HD, transforms the multi-label problem into single-label classification by creating a new class for each observed label combination. We additionally propose Multi-Model HD that creates a binary classification model for each possible label. By doing this, we can leverage the efficiency of HD computing to complete the multi-label classification task faster and with less energy consumption.

### 4.3.1 Problem Transformation Methods

**Power Set:** Prior work[65] mapped multi-label classification to single label classification by creating a new label set that was the power set of the multi-labels. For instance, if a multi-label problem had 3 possible labels for every sample, then prior work would transform the 3 multi-labels into 8 single labels. Where each single label represents each possible combination of the 3 individual labels. This exponential increase in the number of labels does not cause challenges for classifiers that do not scale in complexity with the number of labels. However, HD computing complexity does scale with the number of labels. We address this issue with a binary classification transformation for HD computing explained below.

**Multi-Model:** We propose Multi-Model HD, a method of building a binary classification model for each label as the problem transformation method. Suppose $[l_1...l_h]$ are the labels of the dataset, then after mapping each data point into hypervectors $[v_1...v_n]$, we build $h$ binary classification models, since each label only has a true or false value, i.e., 0 or 1. For example, if a dataset has $h = 3$, we create 3 different HD models, one for each label. Then upon inference, we feed the input data into all 3 of the models, independently checking for the existence of each label. This transformation method is better for HD in multiple ways: 1) HD model size, execution time, and energy scale with the number of classes, so when using Power Set HD, if there is a large number of possible label combinations, Power Set HD will not be as efficient as Multi-Model HD. 2) If a new label is introduced, in Multi-Model HD, we simply need to train a newly added binary classification HD model. However, with Power Set HD, or other models that use the power set transformation method, the entire model needs to be retrained to accommodate the new label combinations. The rest of Section 4.3 is mainly focused on Multi-Model HD, while we additionally provide a comparison with Power Set HD in Section 4.5. Now that the problem has been transformed into $k$ binary classification problems, we describe the algorithmic changes to HD computing blow.

### 4.3.2 Training

As stated in Section 4.3.1, since the multi-label classification problem is transformed into multiple binary classification problems with Multi-Model HD, we build two classes for each label (one for value 0 and one for value 1). As shown in Figure 4.1, after the encoding process, where we utilize the encoding proposed in [15], each data point is classified into either $Class_{label_i=0}$ or $Class_{labeli=1}$ for each label $i$ according to the values of its labels $[l_1...l_h]$. As shown in Figure 4.2$A$, for a dataset that has $h$ labels, the binary model of this dataset would contain $2h$ class HVs in total, one binary classification model for each label where each model contains 2 class HVs.

Unlike in single label classification, in Multi-Model HD, each data point is element wise

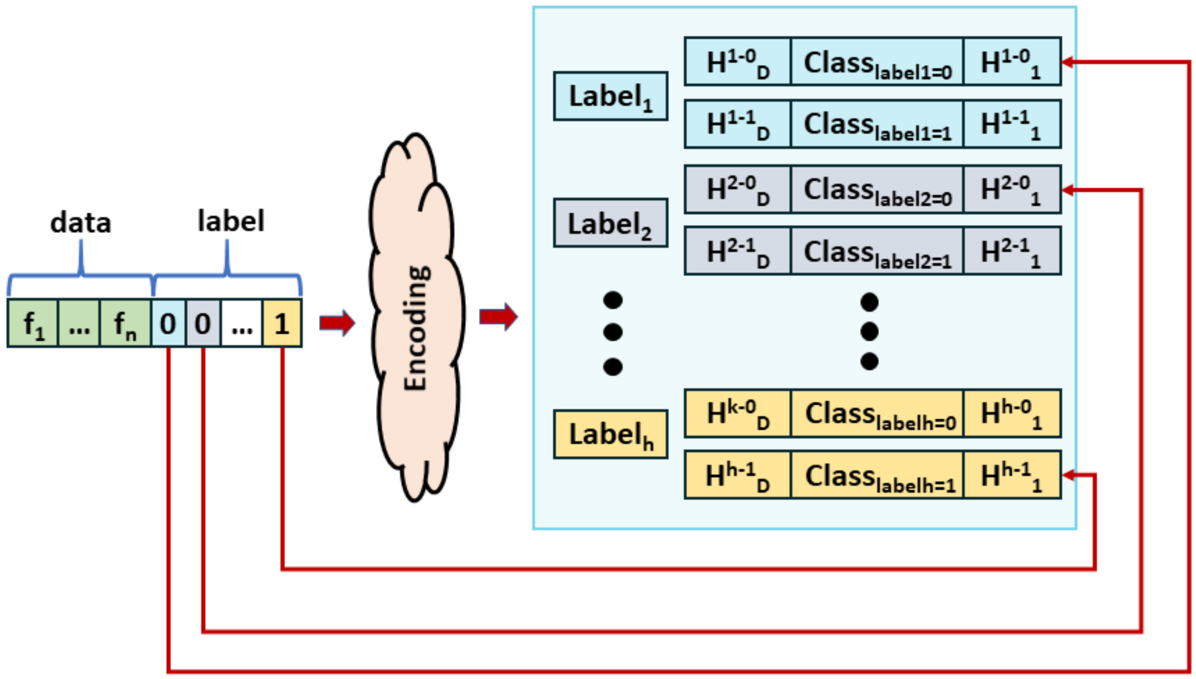


**Figure 4.1.** An example of how the Multi-Model HD model is created

added to multiple class HVs. For instance, in Figure 4.1, after the sample is encoded, it is added to the $Class_{label_1=0}$ class HV for the first label, as the first label is 0. It is then additionally added to the $Class_{label_2=0}$ class HV for the second label, as the second label is 0. This is continued for all the labels until it is added to the $Class_{label_h=1}$ class HV for the last label, as the last label is 1. After this procedure is repeated for the entire training set, we are left with $k$ classification models for each label.

This training procedure also results in integer values for the dimensions of the class HVs, requiring the use of a costly cosine similarity during inference to find the best matching class HV to the query HV. We can reduce this computation to a binary operation of Hamming distance by binarizing the model, which is done by changing the class hypervector elements to +1 if they are positive and -1 if they are negative or 0. Hamming distance is desirable because it reduces each multiplication and addition in cosine similarity to a simple bitwise XOR and accumulation, which is significantly more efficient in acceleration circuits. The class with the least mismatching bits to the query is then chosen as the output.

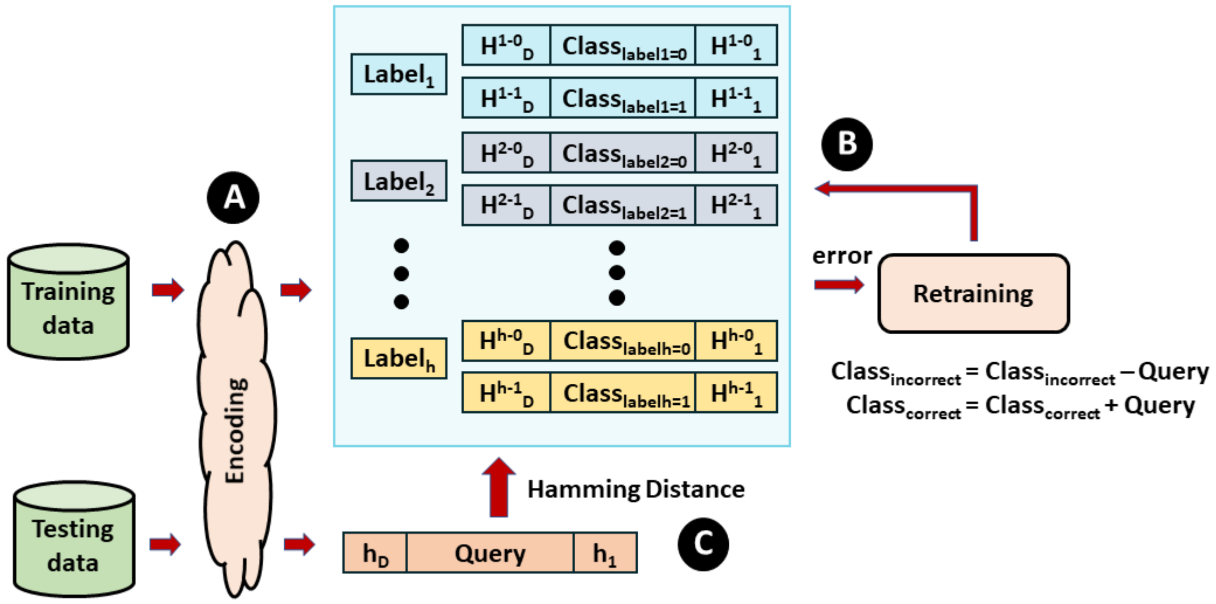

A
Training data
Testing data
Encoding
Label1
Label2
Labelh
H1-0D Classlabel1=0 H1-01
H1-1D Classlabel1=1 H1-11
H2-0D Classlabel2=0 H2-01
H2-1D Classlabel2=1 H2-11
Hh-0D Classlabelh=0 Hh-01
Hh-1D Classlabelh=1 Hh-11
B
error
Retraining
Classincorrect = Classincorrect − Query
Classcorrect = Classcorrect + Query
Hamming Distance
hD Query h1
C

**Figure 4.2.** Overview of how Multi-Model HD is constructed and how Multi-Model HD performs inference.

### 4.3.3 Inference

After training, the HD model for single-label classification can now be used for inference. Upon inference, an input data is encoded to a *query* hypervector using the same encoding module used for training. HD Computing then computes the similarity between the *query* hypervector and each class hypervector. It then uses consine similarity to find a class hypervector with the most similarity with the query hypervector.

Multi-Model HD performs inference in a similar way, however, we need to output $h$ labels instead of just 1. Figure 4.2*C* shows how inference is performed in Muti-Model HD. Upon inference, an input data is encoded to a *query* hypervector using the same encoding module used for training, just like baseline HD. However, since Multi-Model HD contains $h$ different classification models, the query HV is input into each classification model independently. For each model, if the query HV is more similar to the 0 label HV, then that label output is chosen as 0, and vice versa if the query is more similar to the 1 label HV. This generates our $h$ different labels for output in a multi-label classification problem. In Multi-label HD, inference is performed on

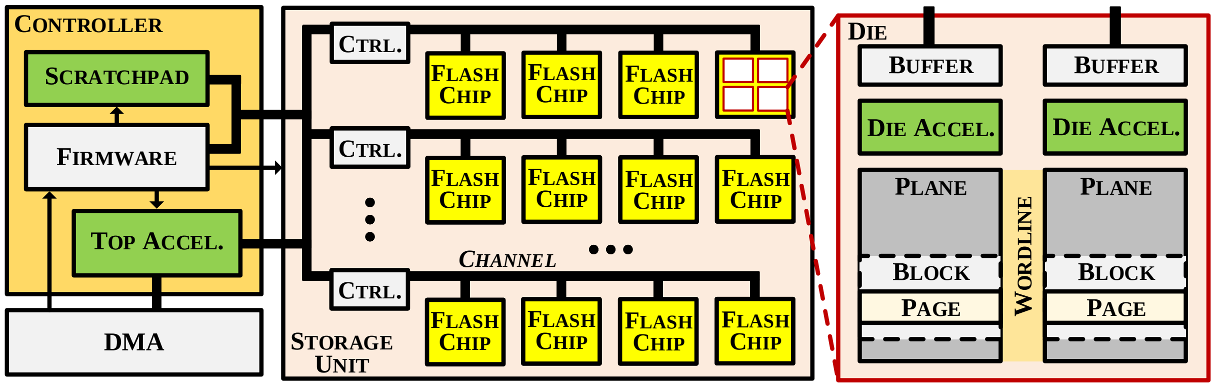


**Figure 4.3.** Overview of Multi-label HD in 3D flash-based storage. ISC enabling components of the design are shown in green.

the host CPU.

## 4.4 Acceleration with 3D NAND Flash

Here, we present an ISC design that performs Multi-label HD encoding and training completely in 3D flash. Figure 4.3 shows an overview of the SSD architecture we adopt from THRIFTY [64]. It uses a die-level accelerator (green on the right in Figure 4.3), in each plane to encode every read page into a hypervector. These hypervectors are then sent to a SSD-level FPGA, which accumulates the hypervectors in the top-level accelerator (green on bottom left in Figure 4.3) to perform training. The scratchpad (green on top left in Figure 4.3) in the controller stores the encoding projection matrix, which it receives as an application parameter from the host. The top-level accelerator is an FPGA which uses INSIDER acceleration cluster [63] to implement HDC accumulation and other operations. We utilize THRIFTY's adaptation of INSIDER's software stack to connect our ISC architecture to the rest of the system.

### 4.4.1 Encoding in 3D Flash

As shown in Figure 4.3, the flash chip may consist of several flash dies which are further divided into flash planes, each plane consisting of a group of blocks, each of which store multiple pages. Each plane has a page buffer to write the data to. Operations in SSD happen in page granularity where the size of the pages usually ranges from 2KB-16KB. Hence, we use accelerators for each flash plane to exploit the flash hierarchy. These accelerators are multiplexed to the page read path.

The die-accelerator in [64] encodes an entire page with raw data to generate a $D$ dimensional hypervector. We assume that the feature vectors are page-aligned, with each page storing one full feature vector. Multi-label HD encoding multiplies an $n$-size feature vector with a projection matrix containing $D \times n$ 1-bit elements. The accelerator calculates the dot product between the two vectors, one read from the flash array and another being a row-vector of the projection matrix. This involves element-wise multiplication of the two vectors and adding together all the elements in the product. Since the weights in the projection matrix $\in \{1, -1\}$, we map them to $\{0, 1\}$ respectively. We use 2's complement to break the multiplication into an inversion using XOR gates and then add the total number of inverted inputs to the accumulated sum of XOR outputs. With the assumption that each page consists of a maximum 1K feature elements, the accelerator consists of an array of 32K XOR gates followed by a 1024 input tree adder. It reduces 1024 inputs to 2, which is followed by a carry look ahead addition to get the final dot product. The sign bit (MSB) of the output is the value of one dimension of the encoded hypervector. Complementary to the projection matrix, the output $0 \rightarrow 1$ and $1 \rightarrow (-1)$. The accelerator is iteratively run $D$ times to generate $D$ dimensions. Each encoded hypervector is appended with the corresponding label vector. We write the output of the accelerator to the page buffer of the plane, which serves as the response to the original read request.

### 4.4.2 Training at Top-Level in Storage

The encoded hypervectors from flash chips are input into the top-level accelerator, which is implemented on an FPGA present in the SSD. During training, they are accumulated into the corresponding label hypervectors. At the end of training we obtain two output hypervector for every label ($label_i$), one each for $Class_{label_i=0}$ and $Class_{label_i=1}$.

The design in [64] utilized input queues for each class to increase parallelism between different classes. However in Multi-label HD, each encoded hypervector is added to one of the two classes of each label, i.e. 50% of the classes. Moreover, ideally an encoded hypervector has just one label as '1' while rest are '0's. Hence, all but one classes corresponding to $label_i = 0$ would receive an incoming hypervector. There is negligible parallelism in training between multiple encoded hypervectors. In this case, the input queues of [64] are an overkill. Hence, we remove input queues from the FPGA design of [64]. The label vector of an incoming hypervector is used to input it to the corresponding class ($Class_{label_i=0}$ or $Class_{label_i=1}$) of each label. The inputs to the remaining classes are set to zero. An accumulator is present for each class, which simply needs to read the input and operate on the corresponding data. The accumulators for each class operates in parallel to add an input hypervector to the corresponding class hypervector. While the computation can also be fully parallelized over all dimensions, the large size of hypervectors and the limited read ports of the memory make it impractical. Hence, we utilize the partition-based approach used in [64] to allow partial parallelism. The final class hypervectors are sent to the host.

## 4.5 Experimental Results

### 4.5.1 Experiment Setup

We tested Multi-label HD training and inference using an optimized C++ implementation. For comparison, we utilized the open source Mulan multi-label package, which is implemented in Java [69]. We compare Multi-label HD with multi-label versions of k-nearest neighbors (kNN), Sequential minimal optimization (SMO), C4.5, and Naive Bayes (NB). We also developed a simulator for Multi-label HD in flash which supports parallel read and write accesses to the flash chips. We utilized Verilog and Synopsys *Design Compiler* to implement and synthesize the die-level accelerator at 45nm and scale it down to 22nm. The top-level FPGA accelerator has been synthesized and simulated in Xilinx Vivado. For drive simulation, we assume the characteristics similar to 1TB Intel DC P4500 PCIe-3.1 SSD connected to an Intel(R) Xeon(R)

Table 4.1. Multi-label HD 3D Storage Parameters

| | | | |
|---|---|---|---|
| **Capacity** | $1TB$ | **Channels** | 32 |
| **Page Size** | $16KB$ | **Chips/Channel** | 4 |
| **External BW** | $3.2GBps$ | **Planes/Chip** | 8 |
| **BW/Channel** | $800MBps$ | **Blocks/Plane** | 512 |
| **Flash Latency** | $53us$ | **Pages/Block** | 128 |
| **FPGA** | $XCKU025$ | **Scratchpad Size** | $4MB$ |
| **Avg Power/DA** | $8mW$ | **DA Latency** | $1.02ns$ |

*DA: Die-accelerator

CPU E5-2640 v3 host. The parameters for our 3D flash implementation are shown in Table 4.1. We compare flash implementation with 6th Gen 3.2GHz Sky Lake Intel Core i5-6300HQ CPU with 8GB of RAM and a 256 GB SSD.

We tested our proposed approach on three applications:

**Genbase** (Genbase) [70]: The protein classes considered are the 27 most important protein families. The training and testing datasets are taken from the Genbase dataset. This dataset consists of 662 samples, each with 1186 attributes.

**Scene** (Scene) [71]: This dataset contains characteristics about images and their classes. One image can belong to one or more classes. The training and testing datasets are taken from the Scene dataset. This dataset contains 2407 samples, each with 294 attributes.

**Yeast** (Yeast) [72]: This database contains information about a set of Yeast cells. The task is to determine the localization site of each cell. The training and testing datasets are taken from the Yeast dataset. This dataset consists of 2417 samples, each with 103 attributes.

### 4.5.2 Multi-label HD Comparison with State-of-the-Art

**Accuracy**

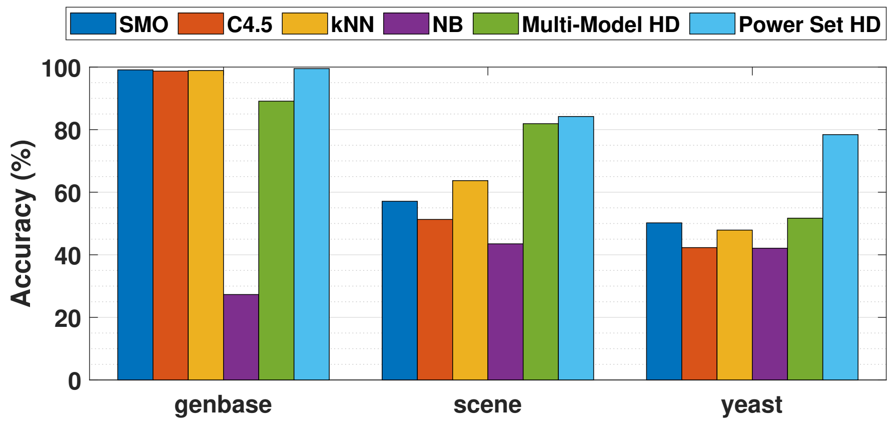


**Figure 4.4.** Classification accuracy of Multilabel HD and other multi-label classification algorithms.

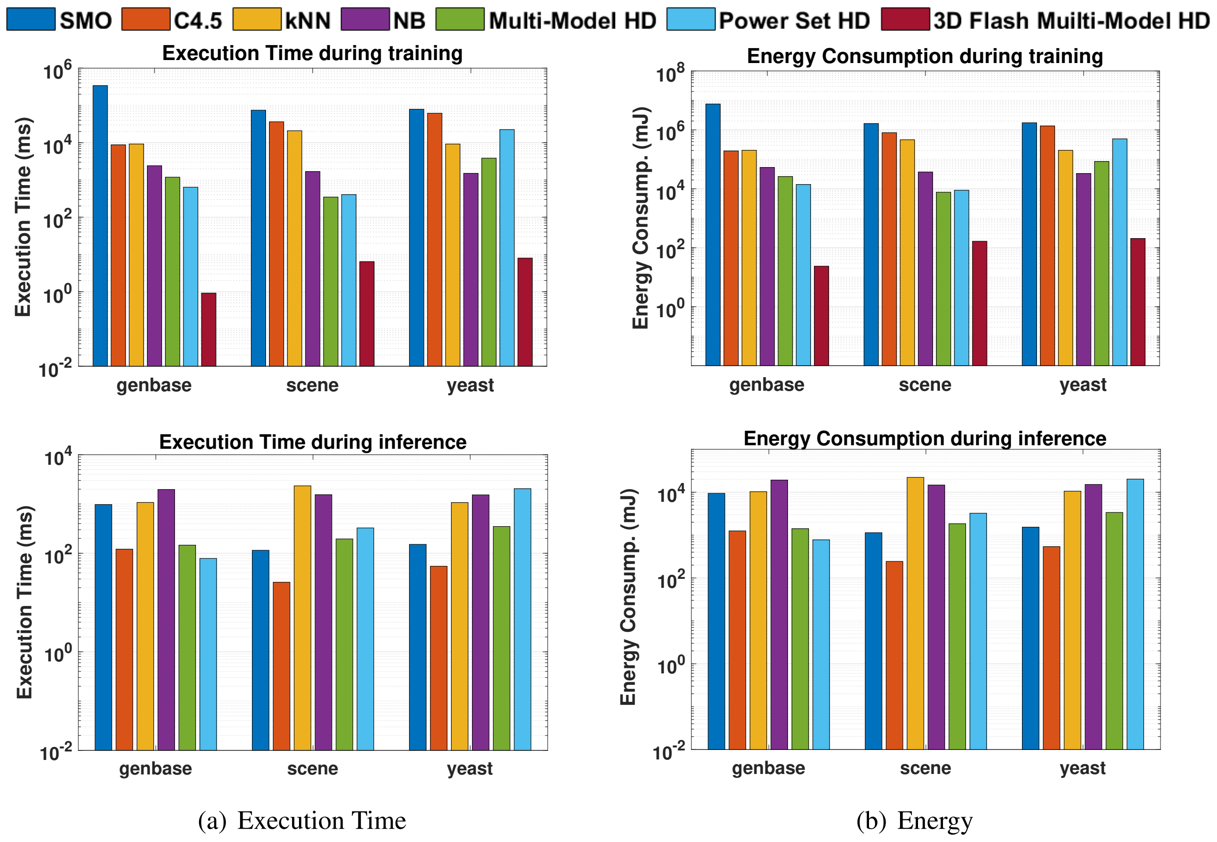


**Figure 4.5.** Energy consumption and execution time of Multi-label HD during Encoding and Training.

Figure 5.8 compares the multi-label classification accuracy of current state-of-the-art multi-label classifiers with Multi-label HD. The accuracy for multi-label is calculated by first getting the accuracy of the model on each label individually. Then to aggregate them, we average each label's accuracy together to get one overall accuracy number for each dataset. As the figure shows, Multi-label HD (Multi-Model HD and Power Set HD) are comparable in accuracy to state-of-the-art multi-label classifiers. In fact, Power Set HD is always better than the state-of-the-art on these three datasets. On the other hand, Multi-model is slightly less accurate than other multi-label classifiers on the genbase dataset by 10%. However, Multi-Model HD is able to achieve higher accuracy on the scene and yeast datasets. This could be attributed to mapping the data into HD space, offering better separability than in the low dimensional data. However, more theoretical analysis on HD Computing is necessary in order to understand why Multi-label HD is more accurate. Overall, on average, Multi-Model HD is 5% more accurate and Power Set HD is 14% more accurate than the highest accuracy state-of-the-art multi-label classifier.

Although Power Set HD achieves higher accuracy than Multi-Model HD, Figure 5.9, demonstrates that the improvement in accuracy comes at a significant cost in execution time and energy. This is because of the exponential increase in class HVs as discussed in Section 4.3.1. As mentioned before, the exception is the genbase dataset because there is only a small subset of possible combinations that appear in the dataset. On the other hand, when there is a large portion of possible combinations in the dataset, Power Set HD is 3.6× slower than Multi-Model HD. This offers a trade-off between execution time and energy efficiency vs accuracy. If an application requires the highest accuracy, Power Set HD should be used. However, if the key metric is execution time and energy efficiency, for a loss in accuracy compared to Power Set HD, but still comparable with other state-of-the-art multilabel classifiers, Multi-Model HD is the clear choice. If the dataset does not have a diverse combination of labels, such as in genbase, Power Set HD can potentially be more accurate and energy efficient compared to Multi-Model HD.

Figure 5.9 compares the execution time and energy consumption of state-of-the-art multi-label classifiers with Multi-label HD on CPU. The data demonstrates that both Multi-Model HD and Power Set HD training are significantly faster than most other multi-label classifiers. On average, Multi-label HD is 60.8× faster and 61.8× more energy efficient than other multi-label classifiers during training. The one exception is Naive Bayes on the yeast dataset, however, although Naive Bayes trains significantly faster than Multi-Model HD on the yeast dataset, Multi-Model HD is 8.6× faster and 8.7× more energy efficient than Naive Bayes during inference. Additionally, Power Set HD is only 3.5× slower than Multi-Model HD on datasets with a large portion of label combinations.

Figure 5.9 also demonstrates that Multi-Model HD is also significantly faster than kNNs and Naive Bayes multi-label models during inference. Although Multi-Model HD is comparable in execution time and energy efficiency to SMO and C4.5 during inference, Multi-Model HD is 174.4×(42.8×) faster and 178.1×(43.1×) more energy efficient than SMO(C4.5) during training. Overall, combining training and one iteration of inference, Multi-Model HD is 47.1× faster and 47.8× more energy efficient than state-of-the-art multi-label classifiers on average, while providing 5% higher classification accuracy. On the other hand, Power Set HD is 24× faster than state-of-the-art multi-label classifiers on average or approximately 2× slower than Multi-Model HD for 13% higher accuracy.

### 4.5.3 Multi-label HD in 3D Flash

Figure 5.9 also shows the latency and energy consumption of Multi-label HD when accelerated in flash. We implement Multi-label HD encoding and training in flash over the three datasets. We observe that our 3D-flash implementation of Multi-label HD is on average 610× faster and 228× more energy-efficient than CPU. Our evaluations show that the performance and energy consumption of Multi-label HD in 3D-flash increases linearly with an increase in the number of training samples. This happens because more data samples result in more huge hypervectors to generate and process. In conventional systems, this translates to a huge amount of data transfers between the core and memory. In contrast, our 3D-flash implementation generates hypervectors (encoding) while reading data out of the slow flash arrays and processes (training) them on the disk itself, reducing data movement.

## 4.6 Conclusion

In this chapter, we design the first accelerator for multi-label HD classification in 3D storage. We also propose two different transformation methods to map HD single label classification to multi-label classification: Power Set HD and Multi-Model HD. Overall, combining training and one iteration of inference, Multi-Model HD is $47.1\times$ faster and $47.8\times$ more energy efficient than state-of-the-art multi-label classifiers, while also achieving 5% higher accuracy on average. Power Set HD can achieve 13% higher accuracy than Multi-Model HD, but is $2\times$ slower. We additionally propose in-3D-flash acceleration that further improves the energy efficiency of Muilti-Model HD training by $228\times$ and speedup by $610\times$. In the next chapter, we discuss how we experimentally evaluate the robust property of HDC in a federated learning enviroment. We then exploit the robustness of HDC to create a more efficient Analog Processing-in-Memory accelerator.

## 4.7 Acknowledgements

This work was supported by NSF grants #1527034, #1730158, #1826967, and #1911095.

Chapter 4, in part, is a reprint of material as it appears in J. Morris, Y. Hao, S. Gupta, R. Ramkumar, J. Yu, M. Imani, B. Aksanli, T. Rosing, ”Multi-label HD Classification in 3D Flash”. IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC), 2020. The dissertation author was the primary investigator and author of this material.

# Chapter 5

# Evaluating and Exploiting Robustness to Create a More Efficient Analog Processing-in-Memory Accelerator for Hyperdimensional Computing Classification and Clustering

## 5.1 Introduction

In this Chapter of the dissertation we evaluate and exploit the robustness of HDC for more efficient hardware. To evaluate the robustness of HDC, we utilize a Federated learning environment where wireless communication is used and can incur errors when transmitting data. “Federated learning” [73] is a popular model for distributed model training in which a centralized model stored on a server is “cloned” to some set of devices which all collect the same features. Each device then updates its local copy of the model and periodically transmits weights to the server, which are used to update the global model via an averaging operation. Intuitively, federated learning reduces communication costs by transmitting only model weights instead of raw training data.

In “Federated learning”, Hyperdimensional (HD) computing offers three benefits [10]. First, an HD “model” is simply a collection of bitvectors which may be less burdensome for communication than other state-of-the-art methods (especially deep neural networks) where the weights are typically floating point values and are non-negligible in size [74, 33]. While a line of deep neural networks research tries to reduce the parameters of these models [75], the number of parameters are still higher than HD. Second, local training of the HD model is extremely simple and more energy efficient than many existing ML techniques [12]. Third, transmitting faulty model weights in classical ML algorithms may lead to slower training or convergence to a worse local optimum compared to HD.

The third point is particularly helpful for "Federated learning". Transmitting model parameters to the central learning system is done mostly through wireless communication. The noise in a wireless channel can incur bit-level errors in the transmitted signal and without error correction, could lead to faulty models due to the noisy data. This is especially true in urban areas where distance is not the only factor adding noise to the wireless channel, but also large buildings and multiple obstacles in the way that degrade the wireless signal.

We additionally take advantage of the simple and highly parallelizable operations in HD to create an analog PIM accelerator with adaptable model bitwidths to achieve the best energy and execution time, while maintaining high accuracy based on the SNR of the wireless channel. This characteristic has made HD the target of various hardware acceleration frameworks, particularly FPGAs [51], and PIM architectures [18, 12, 76]. Although GPUs and FPGAs provide a suitable degree of parallelism that makes them amenable to machine learning algorithms such as deep neural network [77], the complexity of their resources, e.g., floating point units or DSP blocks, is far beyond the HD requirements, making such devices inefficient for HD. Analog PIM architectures tackle this problem as they comprise memresistive arrays with intrinsically non-complex computational capability, which is sufficient for HD operations. Besides block-level parallelism, another remarkable feature of PIM is eliminating the high cost data movement in the traditional von Neumann architectures as, in PIM, data resides where computation is performed. Adding a PIM accelerator for HD computing to perform cognitive tasks provides significant speed up over utilizing the on-board CPU and saves energy with analog computations and less data movement. Our contributions in this chapter are as follows:

- We propose a PIM architecture that adaptively changes the bitwidth of the model based on the SNR of the incoming sample to maintain the accuracy of the HD model while achieving high speedup and energy efficiency. Our PIM architecture is able to achieve 255× better energy efficiency and speed up execution time by 28× compared to the baseline PIM architecture.

- We take advantage of HD Computing's robustness to errors and relax the precision of ADCs in ISAAC [78]., which introduces errors, but improves area and energy efficiency. Our architecture also utilizes quantized values to different bitwidths.

- We additionally evaluate utilizing our accelerator in a federated learning environment, by utilizing a popular network simulator – NS-3 [79] – to model the communication between devices and simulate wireless noise. We compared HyDREA with other light-weight ML algorithms in the same noisy environment. Our results demonstrate that HyDREA is 48× more robust to noise than other comparable ML algorithms. Our results indicate that our proposed system loses less than 1% Classification accuracy, even in scenarios with an SNR under $7dB$.

- We additionally evaluate HD Clustering to the same wireless communication errors and found that our proposed system also looses less than 1% in the mutual information score, even in scenarios with an SNR under $7dB$, which is 57× more robust to noise than K-means.

- Finally, we extend our architecture to support HD Clustering and our results show that our PIM architecture achieves 289× higher energy efficiency and 32× speed up compared to the baseline architecture during Clustering.

### 5.1.1 Related Work

HD computing is light-weight enough to run with acceptable performance on CPUs [31]. However, utilizing a parallel architecture can significantly speed up HD execution time. Imani et al. showed two orders of magnitude speed up when HD runs on GPU [12]. Salamat et al. proposed a framework that facilitates fast implementation of HD algorithms on FPGA [51]. Due to the bit-level operations in HD, which is more suitable for FPGAs than GPUs, they claimed up to 12× energy and 1.7× speed up over GPUs. HD requires much less memory than DNNs, but the required memory capacity is still beyond the local cache of many devices. Thus, an excessive amount of energy and time is spent moving data between these devices and their main memory (off-chip memory in the case of FPGAs).

To resolve this, prior work used PIM architectures, where processing occurs in memory, eliminating the time and energy of data movement [80, 19, 81]. In FELIX [18], a *digital* PIM architecture was proposed. However, digital PIM operations are significantly slower than equivalent analog PIM operations. Prior work accelerated the inference phase of HD computing in analog PIM with an associative memory [12]. However, the associative memory only stored the trained class hypervectors, so the input data needed to be encoded elsewhere and then moved into the associative memory, negating the benefit of less data movement. Also, the associative memory only supports inference in HD. In this chapter, we implement HD Computing in an analog PIM ReRAM architecture based on ISAAC [78]. This architecture allows us to fully implement HD Computing operations end-to-end from encoding to inference unlike prior work. Our architecture differs in that we further take advantage of HD Computing's robustness to noise and relax the precision of the ADCs. We target the ADCs as they are the highest energy overhead in the architecture [78, 26].

Several works claimed that HD signal representations are inherently robust to various forms of noise [82, 83, 13, 14]. Work in [83] investigated the robustness of HD to RTL level errors (e.g. bit-flips) during computation and found an HD-based approach tolerating an 8.8× higher probability of bit-level errors. Similar results are reported in [49].

Work in [83] presented preliminary evidence showing that HD delivered superior performance to conventional data representations in the presence of bit-level errors during processing. Similarly, bit-level errors occur during data transmission as a result of channel noise and interfer-

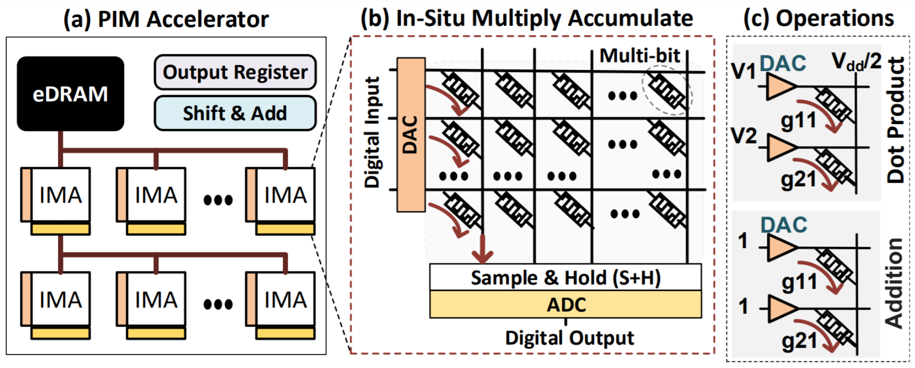


**Figure 5.1.** Overview of the PIM architecture used by HyDREA.

ence from multiple users. To the best of our knowledge, there has been no systematic empirical (or theoretical) evaluation of HD as an avenue for achieving robust learning when data must be communicated over noisy channels. This chapter compares HD computing with a “Federated learning” approach for training other ML models and proposes a new analog PIM architecture to accelerate the whole HD computing algorithm from training to inference.

## 5.2 HyDREA **Analog PIM Architecture**

Combining the energy savings by eliminating data movement and a parallel architecture suitable for dimension-wise parallelism of HD algorithms, analog PIM, with its simple arithmetic support, appears as a promising solution for HD computing. A PIM architecture needs to support three classes of in-memory operations; **(1)** dot-product for the matrix multiplication in encoding and the similarity metric in inference, i.e., the $\vec{\mathscr{H}} \cdot \vec{\mathscr{C}}^j$ part in of the cosine similarity in which each dimension of $\mathscr{H}$ and $\mathscr{C}^j$ is fixed-point (results of binary vector additions), **(2)** addition and subtraction for training and retraining where, we add $\mathscr{H}_i^j$s to produce $\mathscr{C}^j$ which denotes the final class hypervector of inputs with label $j$, and **(3)** search operation to find the best matched class in inference, by finding the maximum of cosine similarity scores between the encoded query $\mathscr{H}$ and all class hypervectors. The baseline architecture provided by ISAAC [78] is perfect for mapping HD Computing to an analog PIM architecture because it supports all 3 of the above operations. This can be seen in Figure 5.1 C.

**(1) Dot Product:** The top half shows how the dot product operation is implemented in our analog PIM crossbar. Assume each resistive cell in the first (i.e., the shown one) column is programmed to resistances $R_{11}$ and $R_{21}$ where $R_{ij}$ belongs to row $i$ and column $j$. Voltages $V_1$ and $V_2$ are applied to the first and second rows. The corresponding generated current flows through the column is $I_1$, which shows the result of dot-product. A larger $I$ shows larger number and since $I = VR$, the resistance of memristive cells need to be proportional to the *inverse* of the value they represent. For 2D vectors, $A$ and $B$, the first set of inputs, $A$, is programmed into the resistances $R_{11}$ and $R_{21}$ having the conductances of ($A_{11} = 1R_{11} = C_{11}$ and $A_{21} = 1R_{21} = C_{21}$). Afterwards, the second set of inputs, $B$, is applied as the voltages at each row ($B_{11} = V_1$ and $B_{21} = V_{21}$). As the figure shows, by applying input values as the voltages to the rows and storing values as conductances, Ohm's law dictates that the current flowing through each resistor is the product of the conductance and applied voltage. Following Kirchhoff's law, the current accumulated at each column is equal to the sum of all the currents flowing through resistors of the column. That is, the total current is $I_1 = C_{11} \cdot V_1 + C_{21} \cdot V_2$. For our design, we store the class hypervectors as the conductances of the ReRam matrix and the query HV is sent as the DAC input voltages.

**(2) Addition:** The bottom half of Fig. 5.1(c) shows how the addition is implemented in a crossbar analog PIM architecture. Addition works analogous to the dot product, except all the input voltages are set to logical 1 (i.e., $V_{high}$). This, the aggregate current of passing through the first column is $I_1 = C_{11} + C_{21}$.

**(3) Search:** Upon performing dot-product between the query hypervector with all class hypervectors, the search operation needs to find the class with the maximum similarity score. In analog PIM, search is implemented using nearest distance search, which finds the most similar value for a given reference. However, we desire a search for the maximum value (so the reference is unknown). But we know the maximum value of the cosine similarity metric is 1, hence we can implement our maximum value search with the already supported nearest distance search by searching for the value that has the highest similarity to reference 1. Hence the returned value will be maximum score. Note that similarity check returns the closest value (absolute difference) by prioritizing MSB bits.

HyDREA takes advantage of HD computing's robustness to noise to reduce computational complexity without losing a significant amount of accuracy. By reducing the bitwidth of the ADCs in analog PIM, HyDREA is able to achieve significant energy savings. However, it comes at a cost of inaccurate computations. However, HD computing is robust to hardware failures and inaccurate computations, making it a perfect candidate to be accelerated by our design. With our bitwidth reduction optimization, HyDREA is able to achieve the energy efficiency of digital PIM with the speed of analog PIM.

### 5.2.1 Architecture

Fig. 5.1(a) shows the architecture HyDREA constituting of multiple In-Situ Multiply Accumulate (IMA) blocks. In our implementation, HyDREA comprises of 24 IMA blocks. The design choice of using 24 IMA blocks was to ensure that our architecture can fit the largest dataset tested. This is critical because if all the data does not fit, data would need to be offloaded and stored off chip. The load and store operations in our ReRAM array are very costly and would incur a significant amount of latency to our design. IMA blocks are memory crossbars with the capability of performing analog addition and dot-product operations. Each IMA block consists of 8 crossbar arrays, each of which contains 128 rows and 128 columns of memory cells. There are $8 \times 128$ Digital-to-Analog (DAC) blocks per IMA, i.e., 128 per each crossbar arrays, allocated to the rows to convert the incoming digital signal (voltage) to analog (current) in order to perform computation. There is also a shared Sample and Hold (S+H) block, and shared Analog-to-Digital (ADC) blocks in each IMA. Fig. 5.1(b) shows an example of a crossbar memory array. Each bitline is connected to all the wordlines through memresistive cells, which have stored the information (e.g., values of class dimensions) by changing the resistance level of each cell. Each memresistive cell in our configuration is a 2 bit MLC, i.e., it has four resistance states to be able to represent 2 bits. Storing the HD model, i.e., the values of classes dimensions,

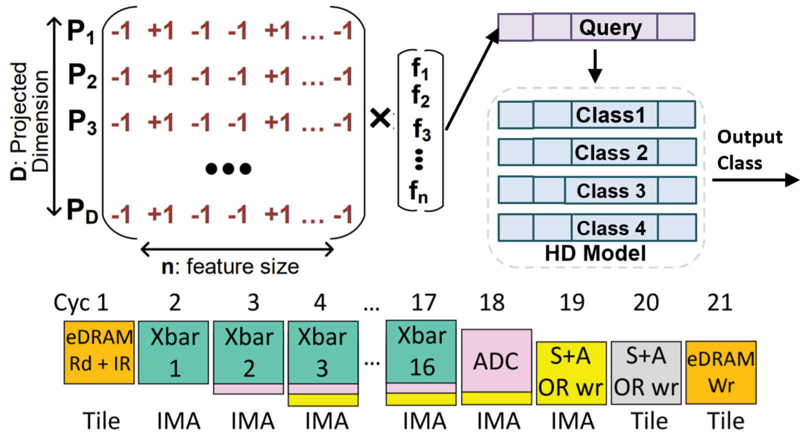


**Figure 5.2.** Example of Inference in HyDREA.

needs to program the NVMs, which is a slow write operation. However, it is only done one time before beginning the inference step, so the overhead is amortized in the entire course of inference.

Figure 5.2 shows an example of how inference is performed in HyDREA. The first step is to encode the input. The input is stored in the eDRAM buffer of the encoder tile. When a new input shows up, it allows the current input to proceed with its next operation. This operation is itself pipelined (shown in Figure 5.2). In the first cycle, an eDRAM read is performed to read the input. These values are sent over the shared bus to the IMA for the encoder and recorded in the input register (IR). After the input values have been moved, the IMA will perform the matrix multiplication during the next 16 cycles.

In the next 16 cycles, the eDRAM is ready to receive other inputs and deal with other IMAs. Over the next 16 cycles, the IR feeds 1 bit at a time for each of the input values to the crossbar arrays. The first 128 bits are sent to crossbars 0 and 1, and the next 128 bits are sent to crossbars 2 and 3. At the end of each cycle, the outputs are latched in the Sample Hold circuits. In the next cycle, these outputs are fed to the ADC units. The results of the ADCs are then fed to the shift-and-add units, where the results are merged with the output register (OR) in the IMA.

As shown in Figure 5.2, at the end of cycle 19, the OR in the IMA has its final output value. This is sent over the shared bus to the central units in the tile. The central OR contains the final results for encoding at the end of cycle 20. During this time, the IMA for the next input has already begun processing to maintain utilization. Finally, in cycle 21, contents of the central OR are written to the eDRAM that will provide the inputs for the similarity check. The similarity check is then performed with the same pipeline as it too is a matrix multiplication.

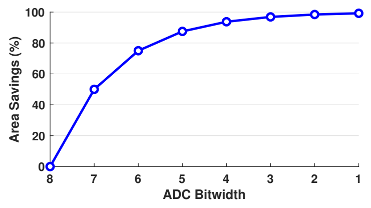
### 5.2.2 Challenges

To perform the computation in analog, PIM needs to convert the signals into analog domain. For this, it requires to employ DAC and ADC converters at the inputs and outputs, respectively. As shown in previous work, these signal domain converters contribute to a significant overhead in the residing architecture [78, 26], which reaches up to 89% of the system power consumption. However, the overhead of these converters can be significantly alleviated as it is exponentially tied in the precision of converters. This, obviously, increases the error as the signal levels are quantized. Fortunately, it is less problematic in the context of HD computing thanks to its remarkable tolerance to error, as information is spread over all the independent and identically distributed dimensions of vectors, so failing the computation on a certain portion of dimensions (bits) should not affect the overall result noticeably.

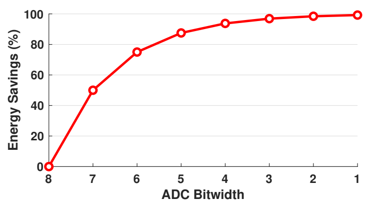
Furthermore, the addition of ADCs for conversion is the largest overhead of using analog PIM for computation. The ADCs take up a huge amount of area as with each bit of resolution added, their area doubles. Prior work tried to alleviate this by sharing the large ADC across multiple blocks [78]. This approach can slow down computation. However, in this chapter we significantly reduce this overhead by using extremely low precision ADCs (as low as 2-bits), which our application, HD Computing, can handle.

### 5.2.3 HyDREA: Analog PIM Architecture Optimiztions

**ADC Reduction:** As in Section 5.2.2, the energy overhead of conversion from the digital domain to the analog domain and back dominates the energy usage of analog PIM, and this is handled by the ADC blocks. Thus, our task to improve the energy efficiency of analog PIM



(a)



(b)

**Figure 5.3.** Area savings (a) and energy consumption savings (b) as the bitwidth of the ADC is dropped.

focuses on improving the energy efficiency of the ADC blocks. We achieve this by reducing the precision of the ADC blocks. Figure 5.3 shows the expected energy and area savings of reducing the bitwidth of an ADC. The results from the energy breakdown of ISAAC shows 89% of energy is used on ADC conversion. Then, knowing that each bit we drop from the ADC reduces ADC energy by approximately half, we can extrapolate the expected savings. As the figure shows, for each reduction in the bitwidth of an ADC, we expect the area and energy consumption to halve. This is because in order to add support for each additional bit, the amount of circuit area doubles and therefore, the energy usage approximately doubles. Instead of using 8-bit ADC blocks in analog PIM that achieve full precision conversion to the digital domain, if we reduce the bitwidth of the ADCs we can reduce the energy usage by half for every bit of the ADC we drop. This will save a significant amount of energy during the analog to digital conversion step in analog PIM. However, as mentioned our computations will lose accuracy and as we drop more bits, our computations will become more inaccurate as we sacrifice precision for energy efficiency.

We can reduce our ADC blocks from 8 bits to n bits. By doing this, we will convert the first n most significant bits and omit the $8-n$ least significant bits. For example if we use a 6 bit ADC block to convert 167 we would lose the last two bits and output 164 instead. This leads to good approximate conversions with large numbers, but very poor approximation with smaller numbers. If we use a 6 bit ADC block to convert 7 we would get 4 which is almost 50% off. Furthermore, we do not produce inaccurate conversions every time. If we convert 172 with a 6 bit ADC block, we wold get 172 because the last two bits of 172 are both 0. Therefore,

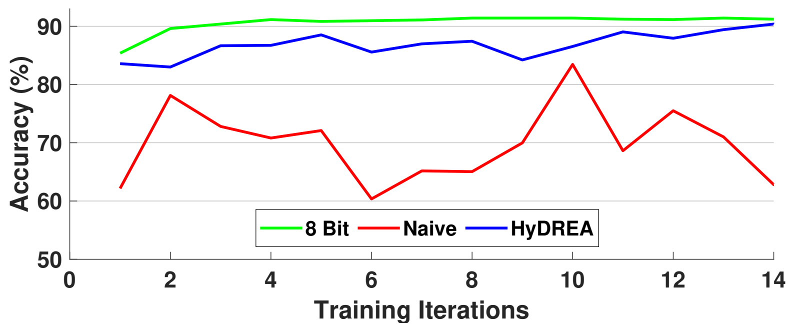


**Figure 5.4.** Impact of HyDREA using a 4 bit model on training compared to training a naive bitwidth reduction 4 bit model and training a 8 bit model.

we produce exact computations when the bits we would drop are all zero. Our ADC block conversions fall into three categories: exact conversions, slightly inaccurate conversions, and highly inaccurate conversions. Since HD computing utilizes dot product as the similarity check, the larger computations dominate the dot product operation and therefore, the highly inaccurate conversions of smaller operations do not effect the accuracy of the HD model. Therefore, we are able to take advantage of reducing the bitwidth of ADCs to create an analog PIM architecture for accelerating HD computing that does not incur a significant loss in accuracy.

**DAC Reduction:** We additionally reduce the energy and execution time overhead of analog PIM by reducing the number of DACs and IMA blocks needed. We achieve this by reducing the precision of the HD model bitwidth.

Due to HD computing's robustness to noise, we could simply reduce the bitwidth of the HD model and achieve efficiency gains without a significant drop in accuracy. When reducing the bitwidth further, training the HD model becomes unstable and the accuracy does not converge. Figure 5.4 compares training an HD model with 4 bits of precision and training the same model with a full 8 bits of precision. The details of the setup and software used to obtain these results can be found in Section 5.4. The top line shows that training an 8 bit model is much smoother and clearly improves in each iteration compared to training with reduced bitwidth. This is because, as HVs are added up and adjusted with retraining, some dimensions may saturate the available bitwidth. Any additional change to dimensions with saturated bitwidths that attempt to change the dimension in the direction of the bitwidth saturation does not improve the model further. For instance, when using a bitwidth of 4, the maximum positive value a dimension can represent is 7. If during retraining, the dimension would be increased further, it would instead stay at 7. In contrast, if the dimension is adjusted with subtraction, it would decrease normally despite any previous attempts to increase the dimension further. This causes over-adjustments in the HD model during retraining when an abnormal change is applied. This is why the accuracy does not converge during retraining with greatly reduced bitwidths. HyDREA is able to improve upon the naive design of simply reducing the bitwidths by additionally modifying the HD algorithm to complement the bitwidth reduction.

The HD model is initially trained by adding up all of encoded data points into one class HV for each class. When reducing the bitwidth of the HD model from 8 bits to 4 bits, 4 bits may not provide enough precision for model convergence during retraining, preventing the HD model from performing effectively at lower bitwidths. To subvert this problem, we propose to analyze the initial HD model to identify key dimensions that need to utilize the full bitwidth available. HyDREA then locks these dimensions to either the maximum or minimum value to ensure the the HD model does not drastically change during retraining.

We propose that the largest dimensions in both the positive and negative directions that saturate the desired bitwidth are key dimensions, as dot product is used as the similarity metric. Hence, the largest dimensions in both positive or negative direction contribute the most to the resulting dot product. Dimensions with the largest values in either direction show that most data points from that class agree in that dimension, i.e. a class HV that represents the class well should ensure these dimensions are not over-adjusted.

To support bitwidth reduction, we propose to modify the initial training algorithm of HD. To identify key dimensions in the HD model to lock, our design first performs the initial training with a full 8 bit representation. HyDREA copies the initial class HV and takes the absolute value of all the dimensions in the class HV and finds the indices of the largest $\alpha$ dimensions that would saturate the desired bitwidth. They are set to the maximum (minimum) value if they saturated in the positive (negative) direction. The other dimensions are scaled down to the desired bitwidth. This is done for all $k$ class HVs. The initial model is then loaded into our PIM architecture. The dimensions that were previously set to the maximum or minimum value are locked from changes during retraining to prevent the HD model from over adjustments. HyDREA only locks dimensions that would saturate the desired bitwidth. If the dimensions do not saturate the desired bitwidth, the bitwidth is sufficient and no change is needed. This lock is achieved by not enabling the write bits at locked dimensions.

Figure 5.4 compares training an HD model with the naive approach of simply reducing the bitwidth to 4 and training the same model with HyDREA using the same bitwidth. The graph shows how HyDREA improves upon the naive design, as during retraining the model is clearly improving and increasing in accuracy like the full 8 bit model. Meanwhile, the naive design's accuracy fluctuates greatly and does not converge.

### 5.2.4 HyDREA: Supporting HD Clustering

The HD Clustering algorithm is very similar to the popular K-means algorithm [22]. The first step of HD Clustering, like Classification is to first encode the data into high-dimensional space. HD Clustering then operates on the encoded HVs as the main datatype. HD Clustering, like K-means, then selects random centers to start. HD Clustering then iterates through all of the encoded data points while comparing them with the cluster centers using a similarity metric and assigning each point to the center it is most similar to. In K-means, that similarity metric is the Euclidean distance. In HD, we utilize cosine similarity for non-binary values, but Euclidean distance could also be used. However, HD maps data into high dimensional space, $D = 10,000$, so calculating cosine similarity is much more efficient. After all the points are labeled, the new centers are chosen and the process is repeated until convergence or the maximum number of iterations is reached. Convergence occurs when no point is assigned to a different cluster compared to the previous iteration. The main difference is that HD Clustering adds a pre-processing step to the Clustering algorithm that maps the data into high dimensional space, or hypervectors.

Here we discuss how we can use HyDREA to support HD Clustering. For HD Clustering, instead of using Euclidean distance, we use cosine similarity to measure the distance between the samples and the cluster centers. This makes mapping HD Clustering onto our existing architecture relatively simple as for Classification, HyDREA already accelerates the similarity checking part of HD inference. Additionally, we use the same encodings for Clustering and Classification, so that accelerator can be reused as well. Therefore, to map HD Clustering to HyDREA, we feed the samples in the original feature domain into our encoding block. Then, to update the distances between the samples and the cluster centers, we feed the cluster centers into the inference accelerator as the class HVs and the samples as the query HVs. This then gives us both the distance in cosine similarity between each sample and all the cluster centers as well as the cluster that each sample is most similar to. The next step of the HD Clustering algorithm, which is to chose the next cluster centers is too complex to accelerate in PIM. However, 98% of the time is spent on encoding and similarity checking. Therefore, offloading updating the cluster centers to the host CPU does not incur a significant amount of overhead.

## 5.3 Network Simulation

Figure 5.5 shows an overview of our federated learning framework and how devices communicate. There are two kinds of devices in our network edge devices and the central node. Edge devices are where local samples are generated. During training, they use a cut down version of our accelerator for HyDREA that just implements encoding to map the data into HD space. The sample is then sent to the central node, where on its way there, the encoded sample is subject to wireless communication noise. The central node's purpose is to collect all encoded samples from all of the edge nodes, train a global model, and perform inference. It too uses our accelerator, except it has full training and inference functionality. Once the global model is sufficiently trained, it can be used for inference. Upon inference, the edge device again encodes

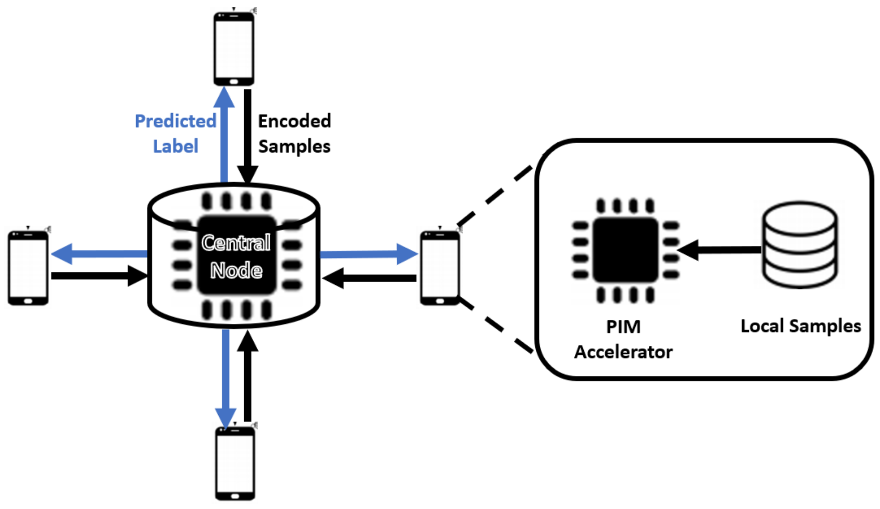


**Figure 5.5.** An Overview of our framework for communicating in the federated learning enviroment.

the input sample to HD space. The sample is then sent to the central node wirelessly incurring a varying degree of noise. The central node then performs inference on the trained HD model and sends the resulting label back to the edge device.

We evaluate the feasibility of HyDREA in a “Federated learning” environment, by utilizing a popular network simulator – ns-3 [79] – to model the communication between devices and simulate wireless noise. In the results section, we compare HyDREA with other ML algorithms in the same noisy environment. The ns-3 physical layer model calculates bit error rates (BER) taking into account the Forward Error Correction (FEC) present in WiFi standards such as IEEE 802.11a/g/n. The model first calculates the received signal–to–noise ratio (SNR) based on parameters used in the simulation model and then calculates a packet error rate (PER) based on the mode of operation (e.g. modulation, coding rate) to determine the probability of successfully receiving a frame (packet success rate - PSR). The received signal SNR depends on the following parameters:

- Transmission powers of devices: Since noise power is usually constant, increasing the transmission power results in a higher SNR, thus lower BER. However, since energy efficiency is crucial in many applications, IoT devices usually operate in low power modes,

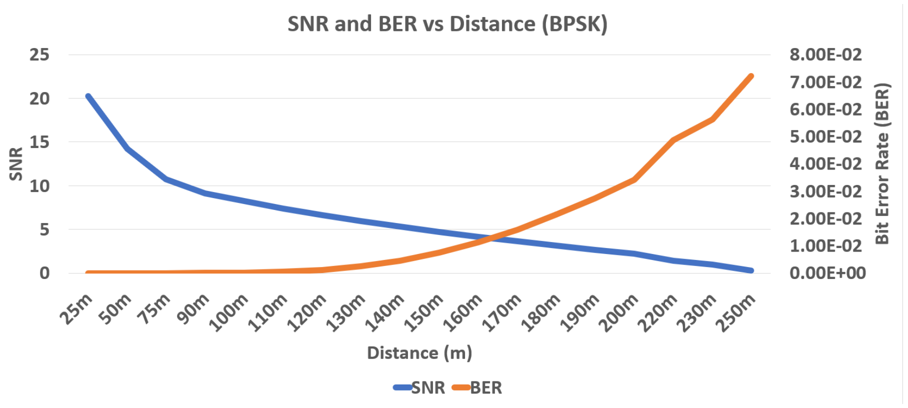


**Figure 5.6.** SNR/BER vs distance for BPSK modulation with Friis prop. loss.

resulting in low SNR.

- Distance between communicating nodes: As two communicating nodes get further away, the received signal strength decreases, resulting in low SNR.

- Propagation loss: The loss in the communication channel is different for different topologies. For example, if two devices are in the line-of-sight of each other, this scenario would incur much less loss compared to them communicating in a dense downtown with buildings blocking the view.

- Interference: When many devices communicate at the same time, each other's signals act as an interfering signal, which degrades the demodulation and decoding performance at the receiving end. In this case, we have to calculate signal-to-interference-plus-noise ratio (SINR).

We study how HD Classification and Clustering performance changes with varying transmission power levels, distance, different propagation loss scenarios, and under different number of interfering devices. Additionally, the error rate depends on the modulation, coding and error correction mechanism adopted by the WiFi technology. Ns-3 allows us to study the error rates for modulation schemes such as BPSK, QPSK, 16to1024 QAM, under binary convolutional coding for rates $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, and $\frac{5}{6}$. We can both enable or disable forward error correction (FEC)

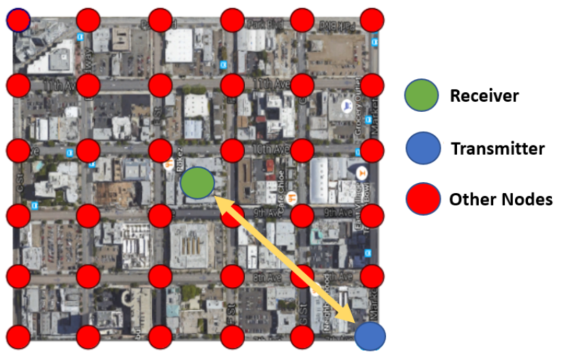


**Figure 5.7.** Model of a Downtown Topology Represented in NS-3, Where Buildings buildings have higher signal attenuation compared to open-air and they block the line-of-sight when they are placed between the transmitters (blue) and the receiver (green).

in all of these cases. Our experiments use the WiFi protocol stack (802.11n), which is the most matured communication standard implementation in ns-3. There are efforts on modeling low-rate and low-power standards for IoT, but they are not fully developed yet. Hence, we modify the 802.11 PHY and MAC layer parameters and scale data rate and power values to imitate communication in an IoT environment. The modulation techniques and coding schemes of 802.11n, namely BPSK, M-ary QAM, and Direct-Sequence Spread Spectrum (DSSS), are common with many low-power wireless protocols [84]. Different techniques have different SNR vs BER (Bit error rates) curves, but these curves are the same across protocols [85, 86, 87]. Since we adjust the parameters of 802.11n, we can simulate the characteristics of low power IoT protocols by operating at the low SNR regions of the SNR-BER curve. We vary the distance between the transmitter and the receiver to collect data at various SNRs. We evaluate with the Friis propagation loss model. Figure 5.6 shows the BER versus distance curve between transmitter and receiver. We additionally test error rates from other sources of noise. Such as a downtown scenario with buildings in between the nodes shown in Figure 5.7 or a highly congested network. We use the hybrid building propagation loss model consisting of Okumura-Hata [88], ITU-R 1411 and ITU-R 1238 [89] loss models. The model includes the multi-path fading loss through building walls for both line-of-sight (LoS) and no LoS cases. There are also random communication attempts between other nodes in the network resulting in dynamic BER and packet losses.

We compare HD with two baseline approaches. In the first, we assume that corrupted data packets are discarded and must be re-transmitted. This ensures the accuracy of the resulting model, but increases latency and energy consumption – especially in congested networks. Second, we train on the corrupted data. This eliminates the need to re-transmit packets but may slow model convergence or cause the model to converge to a worse local optimum (recall that Neural Networks are a non-convex optimization problem). Due to the robustness of HD Computing to noise, the HD model is able to learn more effectively from corrupted packets than other ML models, eliminating the need to re-transmit data while ensuring a high-quality result. Low-power networks such as LoRaWAN and LPWAN usually operate at very low SNRs [90] which can result in error rates ranging from $10^{-5}$ to $10^{-1}$. Many applications require perfect data reconstruction at the receiver, so it is often aimed for networks by design to have an error rate at upper levels of this range. We show that HD is very resilient to errors, such that one can deliberately use very low-power for communication and operate at extremely low SNRs, going beyond the error rates that of standard network configurations, while still getting acceptable accuracy for the learning tasks. This comes with large energy savings that is crucial for resource-constrained IoT devices. We additionally compare HD Computing robustness to Error Correction Codes (ECC) in wireless communication in Section 5.10.

## 5.4 Evaluation

### 5.4.1 Experimental Setup

We verified the functionality of HyDREA using both software and hardware implementations. In software, we implemented HD Classification and Clustering on an Intel Core i7 7600 CPU using an optimized C++ implementation. For the hardware implementation, we used an analog-based PIM architecture proposed in [78]. We modify the ISAAC architecture to more efficiently run for HD Computing by relaxing the bitwidth resolution of the ADCs. Our

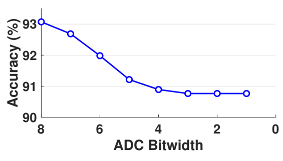
**Table 5.1.** Dataset Information

| Dataset | Type | # Classes | # Train Data | # Test Data | # Features |
|---|---|---|---|---|---|
| **UCIHAR** [41] | **Classification** | 6 | 6,213 | 1,554 | 561 |
| **CARDIO** [43] | **Classification** | 2 | 1,913 | 213 | 21 |
| **FACE** [42] | **Classification** | 2 | 22,441 | 2,494 | 608 |
| **ISOLET** [40] | **Classification and Clustering** | 26 | 6,238 | 1,559 | 617 |
| **Hepta** [91] | **Clustering** | 7 | N/A | 212 | 3 |
| **Tetra** [91] | **Clustering** | 4 | N/A | 400 | 3 |
| **Two Diamonds** [91] | **Clustering** | 2 | N/A | 800 | 2 |
| **Wingnut** [91] | **Clustering** | 2 | N/A | 1016 | 2 |
| **Iris** [91] | **Clustering** | 3 | N/A | 135 | 3 |

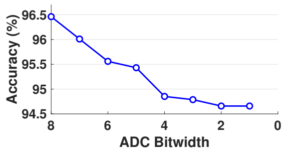
**Table 5.2.** Impact of Dimensionality and Data Representation on the Robustness of HD Computing Classification and Clustering Accuracy.

| Dimensionality | 10,000 | 8,000 | 6,000 | 4,000 | 2,000 |
|---|---|---|---|---|---|
| RP Binary (Classification) | 0.58% | 0.82% | 1.44% | 1.89% | 2.39% |
| ID-Level Binary (Classification) | 0.56% | 0.79% | 1.52% | 1.78% | 2.42% |
| RP (Clustering) | 0.58% | 2.31% | 2.65% | 2.86% | 3.24% |
| ID-Level Binary (Clustering) | 0.66% | 2.48% | 2.52% | 2.79% | 3.13% |
| ID-Level Int (Clustering) | 44.89% | 46.60% | 64.71% | 72.82% | 72.13% |
| ID-Level Float (Clustering) | 85.17% | 85.19% | 85.23% | 85.43% | 85.55% |

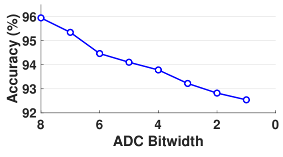
PIM design works at 1.2GHz and uses n bit ADCs, 1 bit DACs, and 128×128 arrays, where each memresistor cell stores 2 bits. To estimate the energy consumption and execution time of HyDREA, we utilize the detailed energy and execution time breakdown of an ISAAC tile found in the original ISAAC paper [78]. We then calculate the estimated execution time and energy by summing up the required operations for HD Computing. We tested our approach for HD Classification on four practical Classification applications and for HD Clustering on six datasets from the Fundamental Clustering Problem Suite [91], shown in Table 5.1.



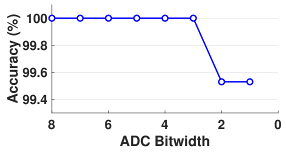
(a) ISOLET



(b) UCIHAR

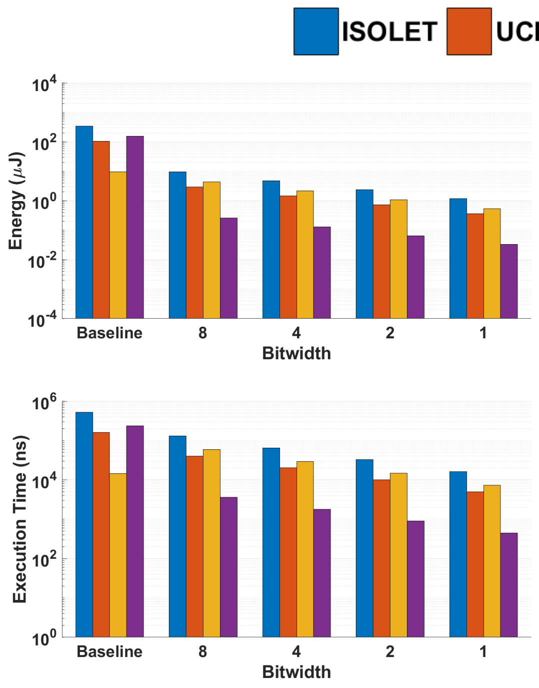


(c) FACE

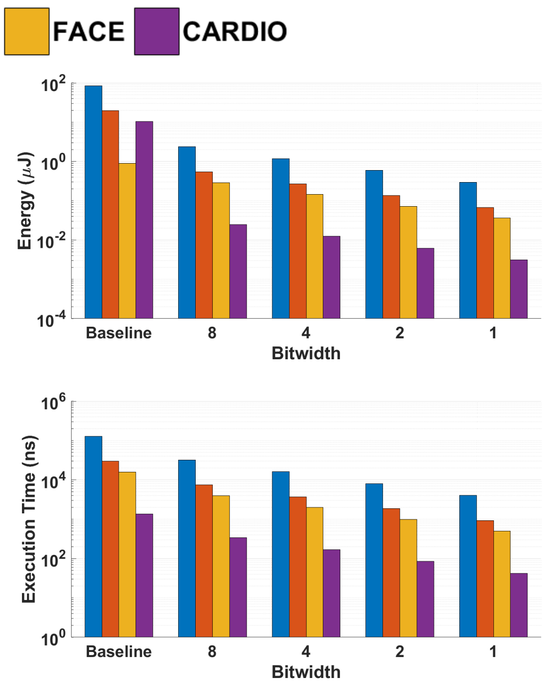


(d) CARDIO

**Figure 5.8.** Impact of bitwidth reduction on accuracy of HyDREA.



(a) Retraining



(b) Inference

**Figure 5.9.** Energy consumption and execution time of HyDREA using different model bitwidths during training and inference with an ADC bitwidth of 2.

### 5.4.2 HyDREA and Dimensionality

To test the impact of dimensionality on HD Classification and Clustering robustness, we utilized the 6.64 SNR test with all datasets. Table 5.2 summarizes the results, where each entry in the table is the average accuracy for all datasets at that dimensionality. There is a clear relationship between HD robustness to errors and dimensionality. One may think that we can achieve faster execution and lower energy consumption with lower dimensionality; but due to our PIM's highly parallel nature, as long as the HD model fits into the PIM arrays, execution

**Table 5.3.** Speedup of HyDREA over a digital PIM implementation with the same bitwidth as HyDREA with the same area.

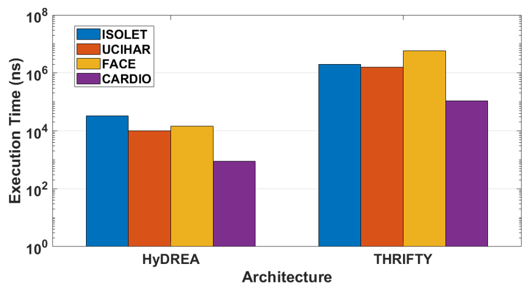
| Dataset | ISOLET | UCIHAR | CARDIO | FACE |
|---|---|---|---|---|
| **Retraining** | 110.4× | 111.8× | 105.6× | 115.2× |
| **Inference Same Bit Digital** | 128.9× | 137.3× | 139.9× | 136.1× |

time and energy does not change. Since our design requires a highly robust HD model, the rest of our tests utilize a dimensionality of $D = 10,000$. Additionally, the table shows that the data representation highly impacts the robustness of HD. Binary values are the most robust because each individual bit flip impact the correctness of the end result the same. However, with other representations such as floating point, depending on the bit flipped, the error can increase significantly. For instance, if an exponent bit is flipped, that would incur significantly more error than if a mantissa bit was flipped. For the most robust models, one should transmit binary encoded HVs.

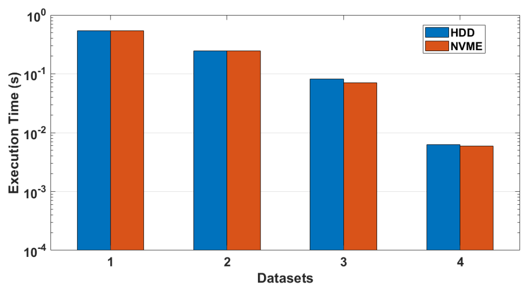
### 5.4.3 HyDREA and the Impact of our Analog PIM Architecture on HD Classification

Figure 5.8 shows the impact of ADC bitwidth reduction on HD model accuracy for four practical applications. The accuracy of each model reduces as the bitwidth drops, but not significantly. When the ADC bitwidth is 4, the average accuracy drop across all applications is 1.5%. This is because our ADC blocks provide highly accurate approximations for high value conversions, and the high value numbers dominate the dot product output. Thus, the resulting dot product closely approximates the exact version. Also, the resulting dot product does not need to be exact, owing to HD's robustness to hardware inaccuracies. Despite inaccurate results, the classes are separated enough that slight variations still result in the HD model selecting the same output class. Overall, HyDREA reduces bitwidth to 2 while only losing 1.8% in accuracy.

Figure 5.9 shows the impact of our analog PIM architecture with 2 bit ADCs and varying



(a) Comparison With THRIFTY [64]



(b) Impact of NVME on HD

**Figure 5.10.** Execution time comparison of HyDREA with THRIFTY, a processing in storage architecture for HD Computing and the impact of higher bandwidth memories such as NVME on HD Computing.

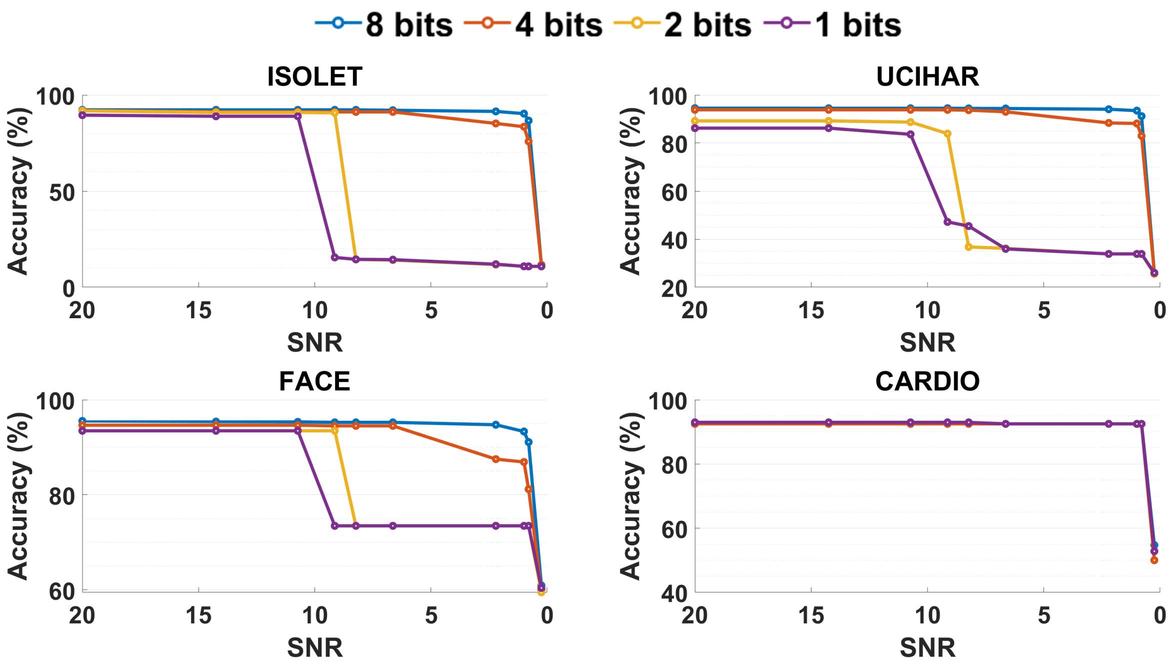


**Figure 5.11.** Accuracy of Design as the SNR varies with an ADC bitwidth of 2 and varying model bitwidth.

model bitwidths on energy consumption and execution time. Our proposed architectural changes drastically improve the energy efficiency and execution time of HD. Our proposed architecture uses 2 bit ADCs and 1 bit models, and achieves 32× (29×) speed up and 232× (267×) higher energy efficiency than the baseline architecture during inference (retraining). Also, in high SNR cases, these models achieve comparable accuracy to full precision models.

### 5.4.4 HyDREA vs Processing in Storage and Digital Processing in Memory

Figure 5.10 compares HyDREA execution time during training to THRIFTY [64]. The results show that due to the slower digital operations in THRIFTY, as well as the higher latency of computing near flash storage, HyDREA is on average 180× faster during training than in storage computing. Furthermore, Figure 5.10 also compares the impact of high bandwidth memory, or specifically NVME storage, on HD Computing latency. We perform this test on the same machine where the only difference is for HDD, we store all data on a slow spinning hard drive and for NVME, we use a PCIe generation 4.0 NVME storage drive. The results clearly show that the higher bandwidth does not impact the overall latency of HD Computing. Therefore, in storage computing solutions such as THRIFTY do not have much to gain from utilizing NVME technologies. Thus, analog processing in memory architectures such as HyDREA are more capable of delivering faster execution times than digital processing in memory architectures.

In Table 5.3 we also compare HyDREA with a FELIX [18] digital PIM based implementation of HD Computing. We compare using the same model bitwidths and memory area. Our results show that HyDREA is 111× faster than the digital PIM design during retraining and 136× faster than the digital PIM design during inference on average. because the individual operations in analog PIM are much faster than they are in digital PIM. HyDREA achieves better speed up during inference than retraining when compared to digital PIM because inference only involves the dot product operation while retraining includes addition operations to adjust the HD model. Due to relying on nor based operations in digital PIM, execution time scales quadratically for multiplications. Therefore, because analog PIM directly implements multiply and accumulate, HyDREA achieves better speed up during inference and retraining.

### 5.4.5 HyDREA and the Impact of SNR on HD Classification

Figure 5.11 shows the impact of SNR on model accuracy in our analog PIM architecture. We can load in low bitwidth models when the channel has a high SNR to achieve the best energy

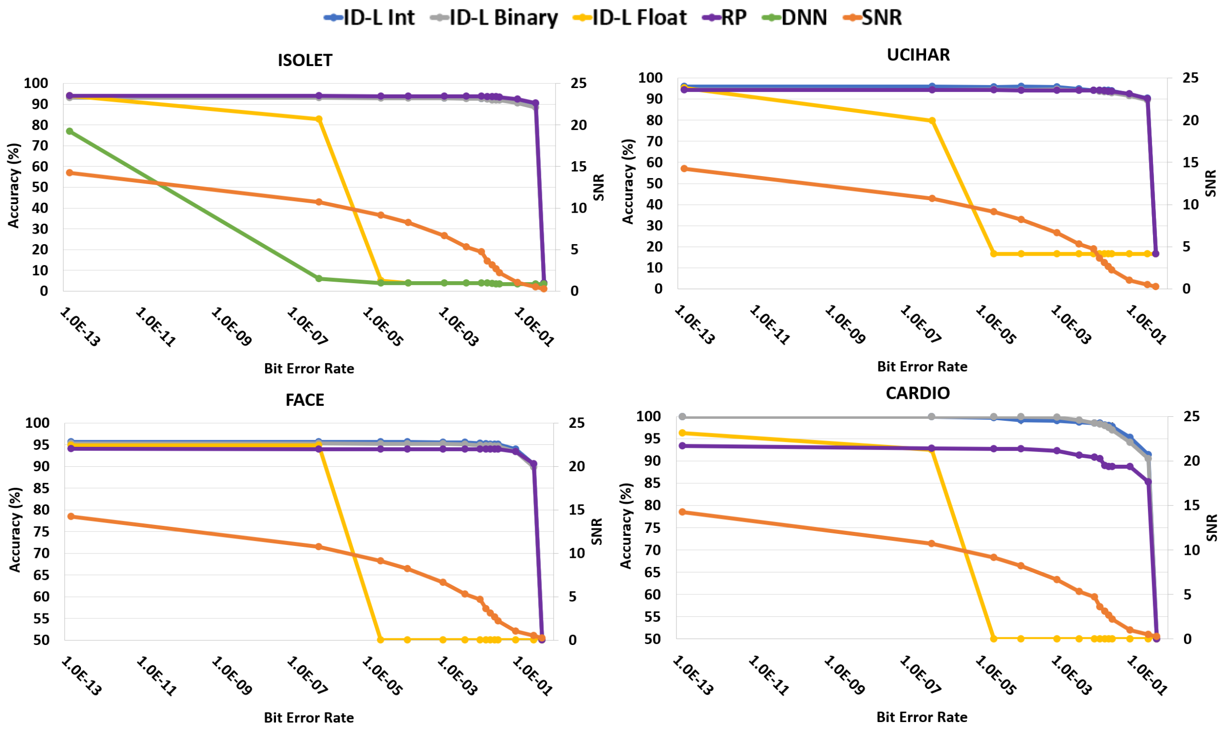


**Figure 5.12.** Accuracy of HD Classification as the SNR varies with different encodings and data representations.

consumption and execution time. However, during high network traffic, longer communication distance, or other factors that incur a high amount of noise on the wireless channel, we need to load in the higher bitwidth models to maintain accuracy. This is because our highly quantized models are taking advantage of HD's robustness to noise by effectively adding more noise to the computation. Therefore, if the environment, in this case wireless communication, is also adding noise, the robust property of HD does not hold up. However, if we adaptively switch which model is loaded based on the SNR, we can maintain high accuracy and achieve significant energy and execution time savings when possible.

Figure 5.12 shows the impact of SNR on model accuracy for two different encodings as well as different datatype representations. Results from all datasets show a similar pattern with increasing bit error rate. HD using integer and binary hypervectors is much more robust to noise as compared to floating-point representations. Since floating-point numbers are represented with mantissa and exponent, if the exponent bits are flipped because of an error, the number itself changes significantly. We additionally compare to a DNN for the ISOLET dataset [92]. The DNN model uses a 16bit floating point representation for its weights, so we can observe the same problem with robustness in DNNs. The data also demonstrates that the random projection (RP) encoding offers similar robustness to noise as the ID-level encoding with binary values. This is likely because our implementation of random projection also encodes hypervectors to binary values (through a final sign function), so both the random projection and quantized ID-level encodings lead to similarly robust binary hypervectors. Lastly, random projection achieves on average, the same accuracy as ID-level, but beats ID-level in some datasets, such as ISOLET, while loses in accuracy to others, such as CARDIO and EMG, as both of them are time-series signals, which random projection does not classify well.

### 5.4.6 HD vs. Other Classifiers

We also compared HD to state-of-the-art classifiers (Linear Regression (LR), MultiLayer Perceptron (MLP), Perceptron, Support Vector Classification (SVC)) and evaluated its robustness to noise on our 4 datasets. Figure 5.13 shows the results for 1) data with no noise, and 2) data corrupted with SNR of 2.21. We choose an SNR or 2.21 because it is the worst practical scenario in our ns-3 setup. All classifiers have comparable accuracy with no noise. While HD stays robust with a significant amount of noise, the other classifiers become very inaccurate. The high-dimensional nature of the hypervectors used in HD leads to significant redundancy in representation which improves its robustness to noise by 48× compared to other classifiers at 2.21 SNR. In other words, HD loses 48× less accuracy compared to the other classifiers. This gives us a metric where noise robustness is defined by how well the model maintains accuracy with the added wireless noise.

### 5.4.7 HyDREA vs State-of-the-Art PIM DNN Accelerator

In Table 5.4 we compare HyDREA with a State-of-the-Art DNN PIM accelerator Q-

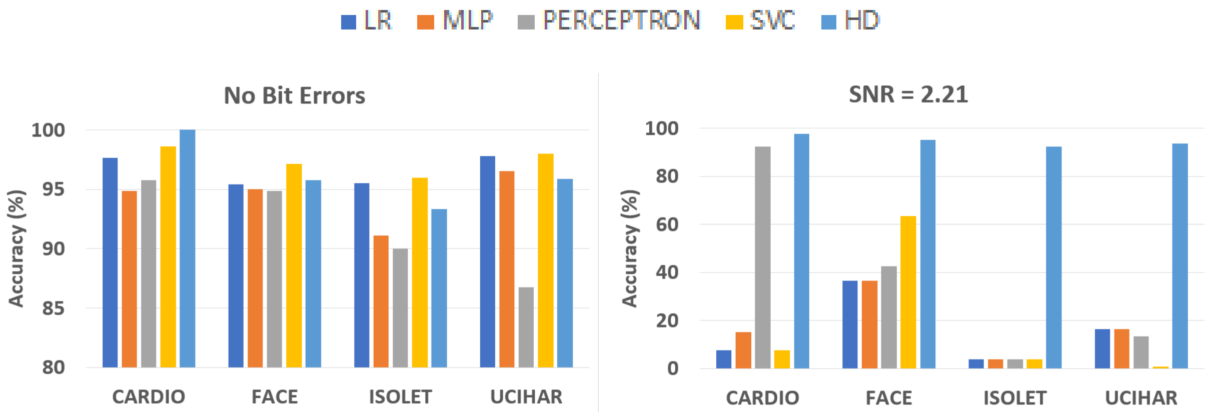
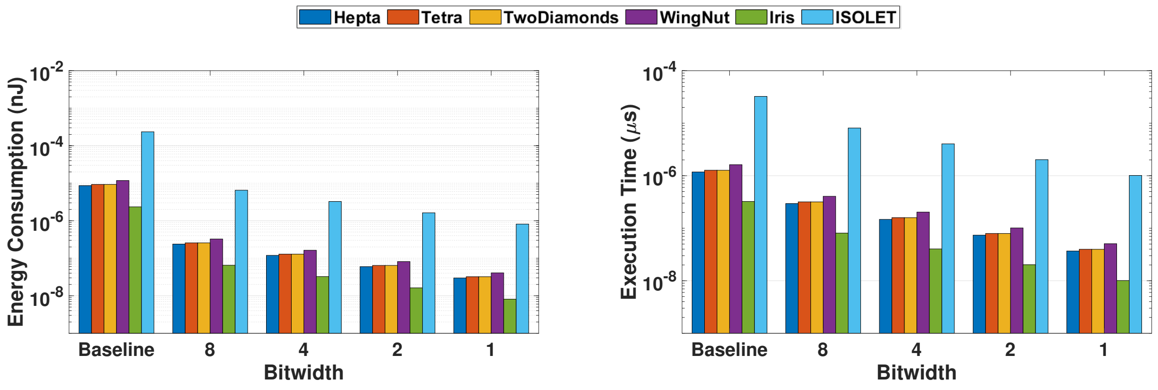


**Figure 5.13.** Comparison of the Robustness of HD to other Classifiers

**Table 5.4.** Comparison of HyDREA with the State-of-the-art DNN PIM Accelerator Q-PIM [1]

| Design | Exact Accuracy | 2.21 SNR Accuracy | Latency(s) | Energy(J) |
|---|---|---|---|---|
| HyDREA | 93.4% | 92.1% | $9.98\times10^{-6}s$ | $8.02\times10^{-7}J$ |
| Q-PIM [1] | 98.5% | 10% | $4.1\times10^{-3}s$ | $4\times10^{-4}J$ |



(a) Energy (b) Execution Time

**Figure 5.14.** Energy consumption and execution time of HyDREA for one Clustering iteration using different model bitwidths with an ADC bitwidth of 2.

PIM [1]. The results show that State-of-the-Art DNNs are able to achieve higher accuracy on more complex datasets such as MNIST. However, in the presence of wireless communication errors, HD Computing is able to maintain its accuracy, while traditional DNNs become unreliable and return random classification results. Furthermore, due to HD Computing's light weight operation, HyDREA achieves a 411× speedup and 498× energy efficiency improvement over

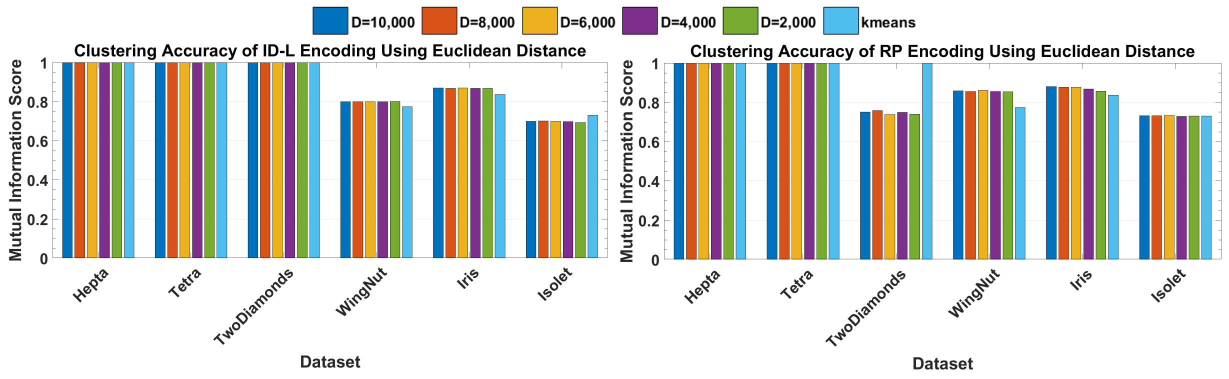


**Figure 5.15.** Comparison of HD Clustering with K-means Accuracy With no Bit Errors

Q-PIM.

### 5.4.8 HyDREA **Architecture Impact on Clustering Energy Consumption and Execution Time**

Figure 5.14 shows the impact of our analog PIM architecture with 2 bit ADCs and varying model bitwidths on energy consumption and execution time for HD Clustering. Our proposed architectural changes drastically improve the energy efficiency and execution time of HD Clustering. Our proposed architecture uses 2 bit ADCs and 1 bit models, and achieves 32× speed up and 289× higher energy efficiency than the baseline architecture during Clustering. Also, in high SNR cases, just like for Classification, these models achieve comparable accuracy to full precision models.

### 5.4.9 HD Clustering Accuracy and Robustness vs K-means

We also compared HD to a state of the art Clustering algorithm, K-means, and evaluated its robustness to noise. As can be seen from Figure 5.15, K-means has a comparable accuracy to HD when there are no bit errors in the dataset. To measure Clustering accuracy, we use a metric based on the mutual information between the cluster assignments returned by our algorithm and ground truth cluster labels. The metric is one when the predicted labels are perfectly correlated with the ground truth and zero when they are totally uncorrelated. Although accuracy is similar without errors, when we introduce errors HD Clustering is significantly more robust. Our proposed system also looses less than 1% in the mutual information score, even in scenarios

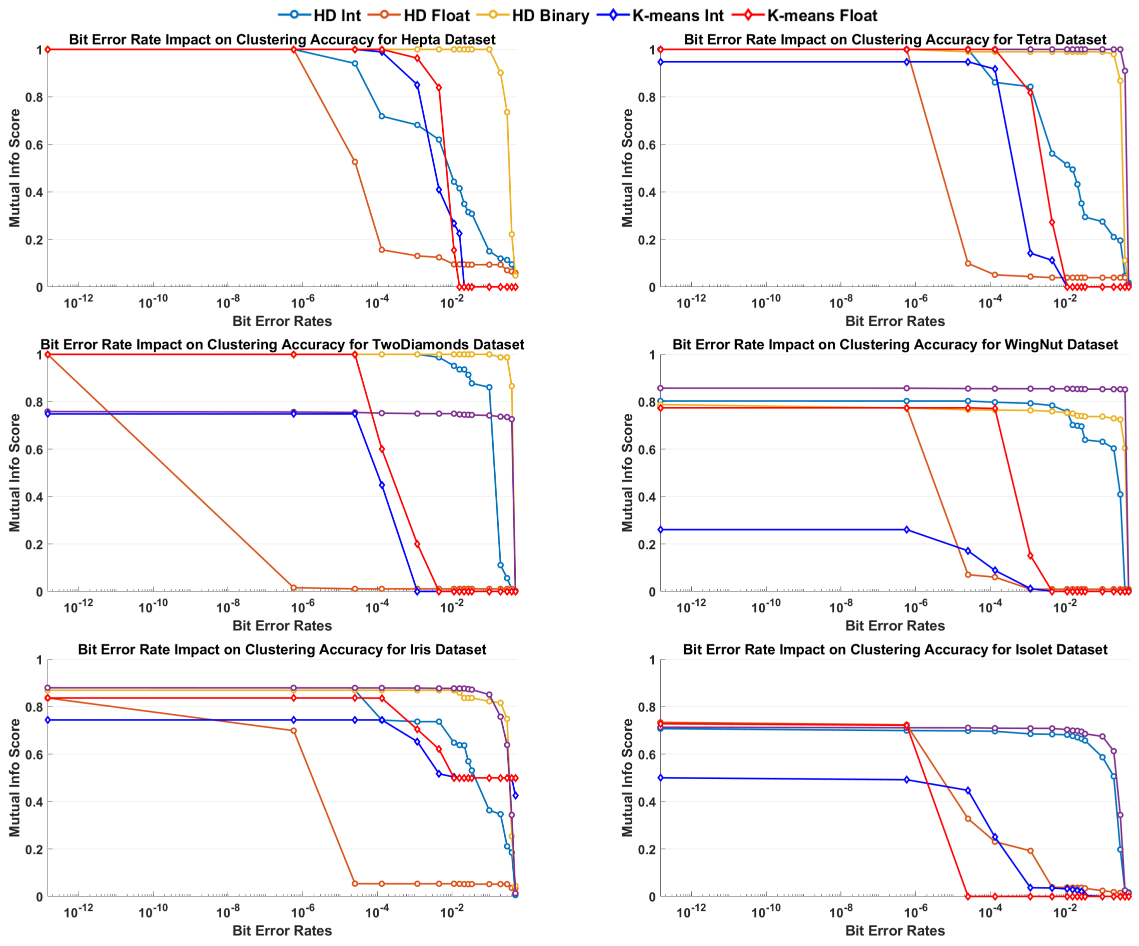


**Figure 5.16.** Accuracy of HD Clustering as the SNR varies with different encodings and data representations vs K-means.

with an SNR under $7dB$, which is $57\times$ more robust to noise than K-means.

Figure 5.16 compares HD Clustering vs K-means Robustness to bit error rates. K-means has similar robustness to bit error rates as HD using integer and floating point representations, until a breaking point around $10^{-3}$ bit error rate for most datasets. This is especially clear with the Isolet dataset, which is the biggest dataset we use. HD Clustering is able to maintain accuracy for much larger bit error rates than K-means when running Isolet. HD gains this additional robust property from the high-dimensional nature of the hypervectors used in HD computing leading to significant redundancy in the representation which improves robustness to noise similar to our Classification results.

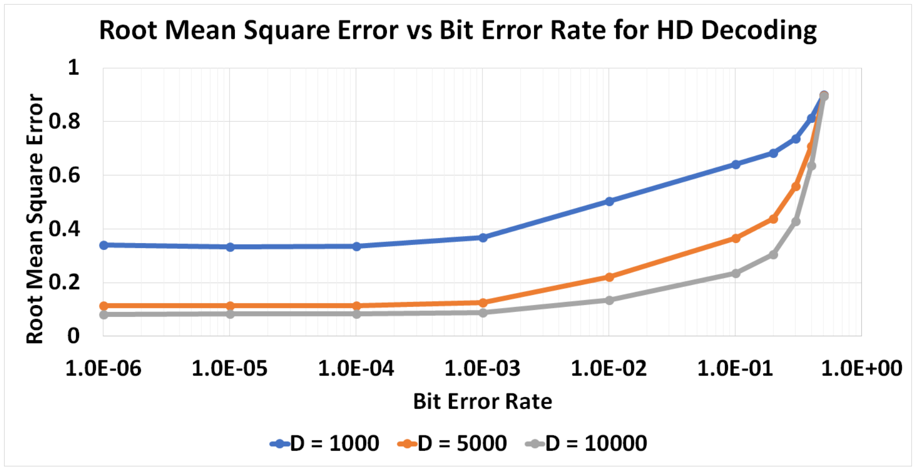


**Figure 5.17.** Impact of Dimensionality on Decoding Quality.

Additionally, similar to our Classification results, the results from all datasets show a similar pattern where HD using integer and binary hypervectors is much more robust to noise as compared to floating-point. Since floating-point numbers are represented with mantissa bits and exponent bits, if the exponent bits are flipped because of an error, the number itself changes significantly, thus incurring more noise. Integer representation performs closer to binary. Random projection provides similar accuracy to binarized ID-Level as random projection encodes hypervectors to binary values as well. Binary representation is the most robust as each individual bit flip incurs the same proportion of noise.

### 5.4.10 Impact of Bit Error Rates on Decoding

Some HD Computing encoding methods have the property where the encoded HV can be decoded back into the original feature vector. For instance, with access to the ID and LV HV banks used to encode the HV in ID-Level, one can decode the encoded HV to get back the original feature vector with some errors [35]. In Figure 5.17, we show the impact of dimensionality on the quality of the recovered feature vector using the ID-Level encoding. The y-axis shows the mean-squared-error of the original feature vector with the decoded one. We test against a range of bit error rates that could be seen in wireless communication as well as across different dimensions. The results indicate that with higher dimensionality, we are able to recover a better quality sample in the original feature space. Additionally, as the bit error rate increases, our decoding quality decreases. The decoded feature vectors become drastically different after bit error rates of around 0.001 for both $5,000$ and $10,000$ dimensions.

### 5.4.11 HD Computing vs Error Correcting Codes (ECC)

In conventional systems, the transmitter performs three steps to generate the wireless signal from data: source coding, channel coding, and modulation. First, a source encoder removes the redundancies and compresses the data. Then, to protect the compressed bitstream against the impairments introduced by the channel, a channel code is applied. The coded bitstream is finally modulated with a modulation scheme which maps the bits to complex-valued samples (symbols), transmitted over the communication link. The receiver inverts the above operations, but in the reverse order. A demodulator first maps the received complex-valued channel output to a sequence of bits. This bitstream is then decoded with a channel decoder to obtain the original compressed data; however, it might be possibly corrupted due to the channel impairments. Lastly, the source decoder provides a (usually inexact) reconstruction of the transmitted data by applying a decompression algorithm.

In this work we deal with robust learning over unreliable communication channels, so we focus only on the channel coding techniques from this pipeline for our comparison. Error correcting codes (ECC) are used in channel coding for controlling errors in data over unreliable and noisy communication channels. The central idea is the sender encodes the message with redundant information in the form of an ECC. This redundancy allows the receiver to detect a limited number of errors that may occur anywhere in the message, and often to correct these errors without retransmission. We implement the setups depicted in Fig. 5.18 and compare channel codes to our method. We refer to the framework shown in Fig. 5.5 with the evaluation setup described in Section 4, and evaluate the inference robustness of the different communcation systems. For all experiments we have an Additive White Gaussian Noise (AWGN) channel, over a range of SNR values, and the modulation type is QAM. In the first setup, there is no channel

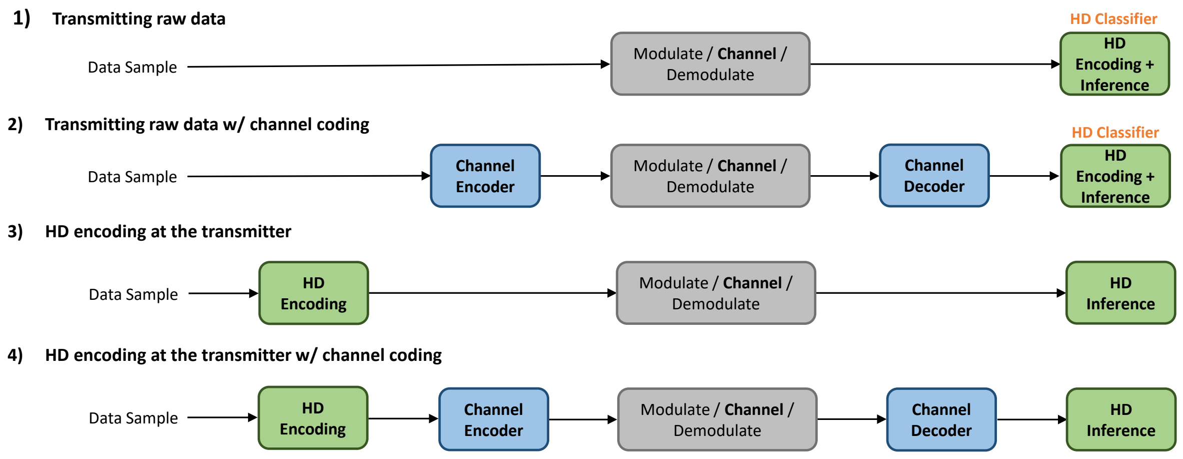

1) Transmitting raw data
Data Sample
Modulate / Channel / Demodulate
HD Classifier
HD Encoding + Inference
2) Transmitting raw data w/ channel coding
Data Sample
Channel Encoder
Modulate / Channel / Demodulate
Channel Decoder
HD Classifier
HD Encoding + Inference
3) HD encoding at the transmitter
Data Sample
HD Encoding
Modulate / Channel / Demodulate
HD Inference
4) HD encoding at the transmitter w/ channel coding
Data Sample
HD Encoding
Channel Encoder
Modulate / Channel / Demodulate
Channel Decoder
HD Inference

**Figure 5.18.** Simulated communication setups

coding and raw data samples are transmitted over the channel. The HD classifier at the receiver side uses these raw data samples corrupted by bit errors to do inference. In the second setup, we add channel coding to the configuration. In the third setup, we apply HD encoding to data at the transmitter side and transmit hypervectors. In this case we don't need to do encoding at the receiver, only a simple similarity check for HD Inference on the corrupted hypervectors suffices. In the fourth setup, we add channel coding on top of HD encoded hypervectors to further add redundancy.

In Fig. 5.19a, we compare a rate $\frac{1}{2}$ convolutional channel code with HD encoding. Viterbi decoder is used to decode the transmitted bitstreams at the centralized receiver. Both channel codes and HD encoding are applied directly to raw data samples, as illustrated in 2nd and 3rd communication setups respectively. The results show that HD encoding has better performance at similar coding rates than convolutional codes. At 35% BER, HD still has around 90% accuracy with 10k dimension hypervectors whereas convolutional code quickly loses accuracy then completely fails. In Fig. 5.19b, we compare HD encoding with high dimension hypervectors to using channel codes combined with lower dimension hypervectors. HD encoding alone performs better at the same overall coding rate, meaning that channel codes do not provide extra protection to the hypervectors. The above results can be explained by Fig. 5.19c, for which

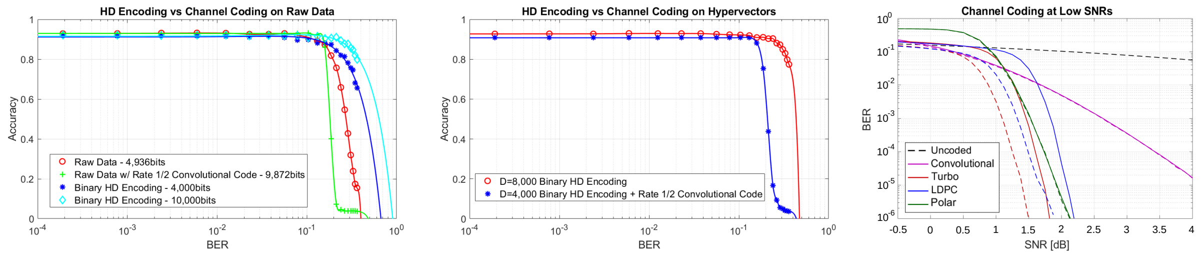


**Figure 5.19.** a) Comparison of HD encoding to channel coding (setup 1,2, and 3), b) combined HD encoding and channel coding (setup 3 and 4), c) channel coding performance at low SNRs, exact (dashed) and approximate (solid) decoding algorithms.

we refer to [93]. All the plotted coding methods are rate $\frac{1}{2}$ as the convolutional code used in the previous experiments. We show the SNR vs BER curves for both the exact (dashed) and approximate (solid) decoding algorithms of the considered methods. As implied by the plots, channel coding gains are significant at moderate to high SNRs. However, BER performance of channel coding converges to that of uncoded communication at low SNRs, for which we perform our experiments. In such cases, particularly where BER is greater than 10%, HD encoding is more robust. Moreover, channel codes aim at correcting the errors and reconstructing the original data. Since we are only interested in using the received data for classification or clustering, the exact reconstructions are not necessarily needed. HD encodings are more suitable for this purpose, as the holographic representation property allows to maintain as much information as possible when part of the data is lost.

## 5.5 Conclusion

In this chapter, we proposed HyDREA, an HD computing system that is Robust, Efficient, and Accurate. We proposed a PIM architecture that adaptively changes the bitwidth of the model based on the SNR of the incoming sample to maintain the robustness of the HD model while achieving high accuracy and energy efficiency. Our results indicate that our proposed system loses less than 1% Classification accuracy even in scenarios with an SNR under $7dB$. Our PIM architecture is also able to achieve 255× better energy efficiency and speed up execution time by 28× compared to the baseline PIM architecture. We evaluated the feasibility of HyDREA in a “Federated learning” environment, by utilizing a popular network simulator, NS-3, to model the communication between devices and simulate wireless noise. We compared HyDREA with other light-weight ML algorithms in the same noisy environment. Our results demonstrated that HyDREA is 48× more robust to noise than other comparable ML algorithms. We additionally tested the robustness of HD Clustering in the same network simulation scenarios and found that our proposed system also looses less than 1% in the mutual information score, even in scenarios with an SNR under $7dB$, which is 57× more robust to noise than K-means. Finally, we extended our PIM architecture to support Clustering and our results show that we are able to achieves 289× higher energy efficiency and 32× speed up compared to the baseline architecture during Clustering. In the next Chapter, we summarize the contributions of this dissertation and go over projects for future work.

## 5.6 Acknowledgements

This work was supported in part by CRISP, one of six centers in JUMP (an SRC program sponsored by DARPA), SRC Global Research Collaboration (GRC) grant, and NSF grants #1911095, #2003279, #2003277, #2100237, and #2120019.

Chapter 5, in part, is a reprint of material as may appear in J. Morris, K. Ergun, B. Khaleghi, M. Imani, B. Aksanli , T. Rosing, ”HyDREA: Utilizing Hyperdimensional Computing For A More Robust and Efficient Machine Learning System.” *ACM Transactions on Embedded Computing Systems (TECS)*, 2022. The dissertation author was the primary investigator and author of this material.

# Chapter 6

# Summary and Future Work

The Internet of Things (IoT), created a network of billions of devices that are generating massive data streams demanding services that pose huge technical challenges due to limited device resources. One of those services is Machine Learning (ML). However, the power and computation capability of many of these IoT or embedded devices do not match the requirements of running ML models. Therefore, instead of running the ML models on these devices, they send their data to the cloud and the ML models are deployed at a data center level. However, sending data to the cloud for processing is not scalable, cannot guarantee the real-time response, uses megawatts of power, and is often not desirable due to privacy and security concerns. To alleviate these concerns, much of IoT data processing will need to run at least partly on devices at the edge of the internet.

## 6.1 Thesis Summary

In order to achieve real-time learning in IoT systems, we need to rethink the algorithms we use for machine learning. We need to redesign the algorithms themselves using strategies that more closely model the ultimate efficient learning machine: the human brain. To address the issue, we propose utilizing Hyperdimensional Computing (HDC) [10]. HDC mimics several desirable properties of the human brain, including: robustness to noise and hardware failure and single-pass learning where training happens in one-shot without storing the training data points or using complex gradient-based algorithms. These features make HDC a promising solution for today's embedded devices with limited storage, battery, and resources, and the potential for noise and variability. Due to these properties, HDC has been demonstrated to be orders of magnitude more efficient than other ML models such as DNNs [18]. HDC research has continued to push the boundaries of efficient machine learning focusing on four main topics: Algorithmic improvements, Hardware/Software Co-Design, Exploiting Robustness, Extending Applications, and HDC Theory, its connection to neuroscience and mathematics. In this thesis, our contributions covered 4 of these 5 topics: (1) the first adaptive model quantization for HDC [2], (2) a hardware-friendly locaclity based encoding for HDC [3], (3) extending the HDC algorithm to support multi-label classification [4], and (4) evaluating and exploiting the robustness of HDC for more efficient hardware [5].

### 6.1.1 Adaptive Model Quantization for Hyperdimensional Computing

Research on HDC algorithmic changes has extended ideas from other ML algorithms to HDC by adding a learning rate, utilizing model quantization, and introducing sparsity. All previous work on model quantization for HDC has been static quantization. This lead to two extremes in the energy and accuracy trade-off curve. Works focused on highly accurate models with less aggressive quantization and more energy consumption, or highly efficient models with aggressive quantization, such as binary models, but at the cost of accuracy loss. In this dissertation, we proposed a new model quantization method that adapatively changes the effective bitwidth for every sample [2]. This enables our design to achieve the energy efficiency and execution time comparable a binary model, while also achieving a similar accuracy to the full precision model.

### 6.1.2 A Rework of the Hyperdimensional Computing Pipeline and Acceleration on FPGA

There is a multitude of works on mapping HDC to different hardware platforms and using hardware/software co-design to achieve more efficient solutions. One problem previous work does not solve though is encoding to HD space. Previous encoding algorithms require random memory accesses and a massive amount of element wise multiplications and additions. All of these operations occurring on hypervectors with 10,000 dimensions. These encodings are inefficient to map to hardware designs. In this dissertation, we propose a new hardware friendly encoding that removes random memory accesses and replaces them with a fixed memory access pattern with sparsity and a subsequent FPGA architecture that takes advantage of the changes [3]. Overall, our new architecture achieves 64× energy efficiency and 10× faster execution time than the previous state of the art FPGA implementation of HDC.

### 6.1.3 Extending Hyperdimensional Computing Applications to Support Multi-Label Classification

Prior work has implemented HDC to support other ML applications beyond classification. However, there were no works on extending HDC to support multi-label classification. In this dissertation, we presented our work on extending HDC to support multi-label classification [4]. Prior work on other classifiers map to multi-label by simply taking the power set of the possible labels and creating new labels for each combination. This reduces the multi-label problem to a single label problem. This works well for other classifiers as they don't scale as poorly with an increase in classes. However, HDC scales linearly with the number of classes, so it scales exponentially with the number of labels in a multi-label problem, when using the power set idea. Therefore, instead of using the power set to reduce the problem to a single label classification problem. We propose a binary classification model for each label. Overall, with this idea, we achieve 47× faster execution time, 48× better energy efficiency, and 5% higher accuracy than other multi-label classifiers.

### 6.1.4 Evaluating and Exploiting Robustness to Create a More Efficient Analog Processing-in-Memory Accelerator for Hyperdimensional Computing

Works in the HDC space often cite that HDC is robust to noise and hardware errors. They also often take advantage of that property in their designs. However, before our work in [5] there were no empirical results to back up the claims of robustness or comparisons with other ML models. We demonstrate experimentally that HDC is 48$\times$ more robust to noise than other machine learning models. We furthermore show how to exploit this property with more efficient hardware. Previous analog PIM designs have a vital flaw where up to 90% of energy is used in the analog to digital conversion (ADC) [26]. However, we alleviate this issue when mapping HDC to analog PIM by reducing the ADC bitwidth. This reduces ADC energy consumption by half for every bit dropped, but results in inaccurate conversions. However, HDC is robust to these errors, and is able to tolerate the inaccurate operations up to a point. Overall, our design achieves 289$\times$ better energy efficiency than simply mapping HDC onto the existing architecture.

## 6.2 Future Work

There are multiple paths for future work on HDC on algorithms, architectures, extending applications, and theory. Our future plan is to continue our work making HDC a compelling light-weight classifier for devices on the edge. One current disadvantage of HDC is that it does not have the same separability capabilities of state-of-the-art Deep Neural Networks. Additionally, although approximate computing has been widely explored in related fields such as DNNs, we demonstrated that HDC is significantly more robust to errors than DNNS. With this knowledge, we can take approximation further in HDC than previously done on DNNs for more efficient hardware solutions.

### 6.2.1 Feature Extraction and HDC

HDC does not currently achieve the same accuracy on more complex datasets such as images. However, DNNs have employed online feature extraction with convolutions layers in order to achieve high accuracies on image based datasets. Future work on HDC should focus on combining HDC with convolutional feature extraction to extend the applications that HDC can accuratly classify. For example, our work in [4] uses simpler feature based datasets for evaluation. We plan to extend this work to support more complex datasets such as text-based or image-based datasets by adding convolutional layers for feature extraction. Furthermore, we previously combined HDC with a single random SNN layer to enable efficient HDC classification on event-based datasets [94]. We plan to extend this work by creating a novel ASIC design that supports the convolutional SNN layer and HDC encoding, training, and inference on the same chip. Furthermore, we can evaluate if HDC is able to maintain it's robust property with errors occurring on the feature extraction model.

### 6.2.2 Exploiting and Evaluating HDC Robustness

We plan to further exploit the robustness of HDC by creating computing architectures that incur errors on the computation, in order to achieve better efficiency. We showed in this dissertation that HDC is significantly more robust (48$\times$ more robust to errors than other classifiers) and can take these architectural changes further than other classifiers [5]. Therefore, we can exploit the robustness of HDC to create even more efficient architectures than previously done with DNNs. For example, one concern with digital processing in memory is the longevity of the cells. Eventually, cells can get stuck in a state permanently holding its last value or 0 or 1. We plan on demonstrating that because HDC is robust to errors, it can still accurately classify even with a significant amount of dead memory cells. This would result in HDC digital PIM architectures having a significantly larger effective number of re-write cycles compared to a DNN solution.

We additionally plan to further evaluate our results in [5]. In the current paper, we perform all of our experiments with network simulations utilizing Ns-3. We plan on validating our experiments with real world examples. We will set up a network of IoT devices to encode data and send them to a central server over wireless communication with no error correction protocols. Then evaluate the model that is created on the central server. We will perform this experiment in multiple different challenging scenarios, such as downtown with buildings in the way, a crowded public area with wireless interference from other devices, and varying distances.

# Bibliography

[1] Y. Long, E. Lee, D. Kim, and S. Mukhopadhyay, "Q-pim: A genetic algorithm based flexible dnn quantization method and application to processing-in-memory platform," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[2] J. Morris, S. T. K. Set, G. Rosen, M. Imani, B. Aksanli, and T. Rosing, "Adaptbit-hd: Adaptive model bitwidth for hyperdimensional computing," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pp. 93–100, IEEE, 2021.

[3] J. Morris, Y. Hao, R. Fernando, M. Imani, B. Aksanli, and T. Rosing, "Locality-based encoder and model quantization for efficient hyper-dimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[4] J. Morris, Y. Hao, S. Gupta, R. Ramkumar, J. Yu, M. Imani, B. Aksanli, and T. Rosing, "Multi-label hd classification in 3d flash," in *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, pp. 10–15, IEEE, 2020.

[5] J. Morris, K. Ergun, B. Khaleghi, M. Imani, B. Aksanli, and T. Rosing, "Hydrea: Towards more robust and efficient machine learning systems with hyperdimensional computing," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 723–728, IEEE, 2021.

[6] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, "Predicting parameters in deep learning," *Advances in neural information processing systems*, vol. 26, 2013.

[7] A. Zaslavsky, C. Perera, and D. Georgakopoulos, "Sensing as a service and big data," *arXiv preprint arXiv:1301.0159*, 2013.

[8] Y. Sun, H. Song, A. J. Jara, and R. Bie, "Internet of things and big data analytics for smart and connected communities," *IEEE Access*, vol. 4, pp. 766–773, 2016.

[9] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: the internet of things architecture, possible applications and key challenges," in *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, pp. 257–260, IEEE, 2012.

[10] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.

[11] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *International Conference on Rebooting Computing (ICRC)*, pp. 1–6, IEEE, 2017.

[12] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 445–456, IEEE, 2017.

[13] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2017.

[14] M. Imani, S. Bosch, S. Datta, S. Ramakrishna, S. Salamat, J. M. Rabaey, and T. Rosing, "Quanthd: A quantization framework for hyperdimensional computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[15] M. Imani, J. Morris, J. Messerly, H. Shu, Y. Deng, and T. Rosing, "Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[16] M. Imani, J. Morris, S. Bosch, H. Shu, G. De Micheli, and T. Rosing, "Adapthd: Adaptive efficient training for brain-inspired hyperdimensional computing," in *2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 1–4, IEEE, 2019.

[17] M. Imani, S. Salamat, B. Khaleghi, M. Samragh, F. Koushanfar, and T. Rosing, "Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 190–198, IEEE, 2019.

[18] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, IEEE, 2018.

[19] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 802–815, IEEE, 2019.

[20] Y. Hao, S. Gupta, J. Morris, B. Khaleghi, B. Aksanli, and T. Rosing, "Stochastic-hd: Leveraging stochastic computing on hyper-dimensional computing," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pp. 321–325, IEEE, 2021.

[21] B. Khaleghi, H. Xu, J. Morris, and T. Š. Rosing, "tiny-hd: Ultra-efficient hyperdimensional computing engine for iot applications," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 408–413, IEEE, 2021.

[22] M. Imani, Y. Kim, T. Worley, S. Gupta, and T. Rosing, "Hdcluster: An accurate clustering using brain-inspired high-dimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1591–1594, IEEE, 2019.

[23] Y. Guo, M. Imani, J. Kang, S. Salamat, J. Morris, B. Aksanli, Y. Kim, and T. Rosing, "Hyperrec: Efficient recommender systems with hyperdimensional computing," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 384–389, IEEE, 2021.

[24] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "Hdna: Energy-efficient dna sequencing using hyperdimensional computing," in *2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, pp. 271–274, IEEE, 2018.

[25] J. Morris, M. Imani, S. Bosch, A. Thomas, H. Shu, and T. Rosing, "Comphd: Efficient hyperdimensional computing using model compression," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2019.

[26] S. Ghodrati, H. Sharma, S. Kinzer, A. Yazdanbakhsh, K. Samadi, N. S. Kim, D. Burger, and H. Esmaeilzadeh, "Mixed-signal charge-domain acceleration of deep neural networks through interleaved bit-partitioned arithmetic," *arXiv preprint arXiv:1906.11915*, 2019.

[27] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[28] B. Lesser, M. Mücke, and W. N. Gansterer, "Effects of reduced precision on floating-point svm classification accuracy," *Procedia Computer Science*, vol. 4, pp. 508–517, 2011.

[29] M. Wess, S. M. P. Dinakarrao, and A. Jantsch, "Weighted quantization-regularization in dnns for weight memory minimization toward hw implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2929–2939, 2018.

[30] S. Ye, T. Zhang, K. Zhang, J. Li, J. Xie, Y. Liang, S. Liu, X. Lin, and Y. Wang, "A unified framework of dnn weight pruning and weight clustering/quantization using admm," *arXiv preprint arXiv:1811.01907*, 2018.

[31] M. Imani, J. Messerly, F. Wu, W. Pi, and T. Rosing, "A binary learning framework for hyperdimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 126–131, IEEE, 2019.

[32] O. Rasanen and J. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–12, 2015.

[33] M. Imani, C. Huang, D. Kong, and T. Rosing, "Hierarchical hyperdimensional computing for energy efficient classification," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[34] Y. Kim, M. Imani, and T. S. Rosing, “Efficient human activity recognition using hyperdimensional computing,” in *Proceedings of the 8th International Conference on the Internet of Things*, pp. 1–6, 2018.

[35] M. Imani, Y. Kim, S. Riazi, J. Messerly, P. Liu, F. Koushanfar, and T. Rosing, “A framework for collaborative learning in secure high-dimensional space,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 435–446, IEEE, 2019.

[36] J. Morris, M. Imani, S. Bosch, A. Thomas, H. Shu, and T. Rosing, “Comphd: Efficient hyperdimensional computing using model compression,” in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2019.

[37] Y.-C. Chuang, C.-Y. Chang, and A.-Y. A. Wu, “Dynamic hyperdimensional computing for improving accuracy-energy efficiency trade-offs,” in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 1–5, IEEE, 2020.

[38] M. Imani, X. Yin, J. Messerly, S. Gupta, M. Niemier, X. S. Hu, and T. Rosing, “Searchd: A memory-centric hyperdimensional computing with stochastic training,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[39] T. Feist, “Vivado design suite,” *White Paper*, vol. 5, 2012.

[40] “Uci machine learning repository: Isolet dataset.” http://archive.ics.uci.edu/ml/datasets/ISOLET, 1994.

[41] “Uci machine learning repository: Har dataset.” https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities, 2012.

[42] G. Griffin, A. Holub, and P. Perona, “Caltech-256 object category dataset,” 2007.

[43] “Uci machine learning repository: Cardiotocography dataset,” 2010.

[44] “Uci machine learning repository.” https://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT.

[45] A. Rahimi, P. Kanerva, and J. M. Rabaey, “A robust and energy-efficient classifier using brain-inspired hyperdimensional computing,” in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 64–69, ACM, 2016.

[46] P. Kanerva, J. Kristofersson, and A. Holst, “Random indexing of text samples for latent semantic analysis,” in *Proceedings of the 22nd annual conference of the cognitive science society*, vol. 1036, Citeseer, 2000.

[47] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey, “Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition,” in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, IEEE, 2016.

[48] T. F. Wu, H. Li, P.-C. Huang, A. Rahimi, J. M. Rabaey, H.-S. P. Wong, M. M. Shulaker, and S. Mitra, "Brain-inspired computing exploiting carbon nanotube fets and resistive ram: Hyperdimensional computing case study," in *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 492–494, IEEE, 2018.

[49] H. Li, T. F. Wu, A. Rahimi, K.-S. Li, M. Rusch, C.-H. Lin, J.-L. Hsu, M. M. Sabry, S. B. Eryilmaz, J. Sohn, W.-C. Chiu, M.-C. Chen, T.-T. Wu, J.-M. Shieh, W.-K. Yeh, J. M. Rabaey, S. Mitra, and H.-S. P. Wong, "Hyperdimensional computing with 3D VRRAM in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *2016 IEEE International Electron Devices Meeting (IEDM)*, pp. 16–1, IEEE, 2016.

[50] M. Imani, S. Salamat, S. Gupta, J. Huang, and T. Rosing, "Fach: Fpga-based acceleration of hyperdimensional computing by reducing computational complexity," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 493–498, 2019.

[51] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 53–62, 2019.

[52] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *Nature Electronics*, vol. 3, no. 6, pp. 327–337, 2020.

[53] S. Salamat, M. Imani, and T. Rosing, "Accelerating hyperdimensional computing on fpgas by exploiting computational reuse," *IEEE Transactions on Computers*, 2020.

[54] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 4, pp. 1–25, 2019.

[55] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 267–278, IEEE Press, 2016.

[56] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard, "Adaptive quantization for deep neural network," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[57] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8612–8620, 2019.

[58] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, "Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization

errors," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[59] T. I. Cannings and R. J. Samworth, "Random-projection ensemble classification," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 79, no. 4, pp. 959–1035, 2017.

[60] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[61] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 153–165, 2016.

[62] G. Koo, K. K. Matam, I. Te, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 219–231, IEEE, 2017.

[63] Z. Ruan, T. He, and J. Cong, "Insider: designing in-storage computing system for emerging high-performance drive," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, pp. 379–394, 2019.

[64] S. Gupta, J. Morris, M. Imani, R. Ramkumar, J. Yu, A. Tiwari, B. Aksanli, and T. Š. Rosing, "Thrifty: training with hyperdimensional computing across flash hierarchy," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2020.

[65] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 3, no. 3, pp. 1–13, 2007.

[66] T. Durand, N. Mehrasa, and G. Mori, "Learning a deep convnet for multi-label classification with partial labels," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 647–657, 2019.

[67] M. Schmuck, L. Benini, and A. Rahimi, "Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 4, pp. 1–25, 2019.

[68] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. De Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "Deepstore: In-storage acceleration for intelligent queries," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 224–238, 2019.

[69] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas, "Mulan: A java library for multi-label learning," *Journal of Machine Learning Research*, vol. 12, pp. 2411–2414, 2011.

[70] S. Diplaris, G. Tsoumakas, P. A. Mitkas, and I. Vlahavas, "Protein classification with multiple algorithms," in *Panhellenic Conference on Informatics*, pp. 448–456, Springer, 2005.

[71] M. R. Boutell, J. Luo, X. Shen, and C. M. Brown, "Learning multi-label scene classification," *Pattern recognition*, vol. 37, no. 9, pp. 1757–1771, 2004.

[72] A. Elisseeff and J. Weston, "A kernel method for multi-labelled classification," in *Advances in neural information processing systems*, pp. 681–687, 2002.

[73] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[74] M. Imani, Z. Zou, S. Bosch, S. A. Rao, S. Salamat, V. Kumar, Y. Kim, and T. Rosing, "Revisiting hyperdimensional learning for fpga and low-power architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 221–234, IEEE, 2021.

[75] M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, "Genecai: Genetic evolution for acquiring compact ai," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, p. 350–358, Association for Computing Machinery, 2020.

[76] M. Imani, S. Pampana, S. Gupta, M. Zhou, , Y. Kim, and T. Rosing, "Dual: Acceleration of clustering algorithms using digital-based processing in-memory," in *Proceedings of the International Symposium on Microarchitecture*, IEE/ACM, 2020.

[77] M. Samragh, M. Javaheripi, and F. Koushanfar, "Encodeep: Realizing bit-flexible encoding for deep neural networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 6, pp. 1–29, 2020.

[78] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[79] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[80] C. Li, F. Müller, T. Ali, R. Olivo, M. Imani, S. Deng, C. Zhuo, T. Kämpfe, X. Yin, and K. Ni, "A scalable design of multi-bit ferroelectric content addressable memory for data-centric computing," in *2020 IEEE International Electron Devices Meeting (IEDM)*, pp. 29–3, IEEE, 2020.

[81] M. Imani, M. S. Razlighi, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, “Deep learning acceleration with neuron-to-memory transformation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–14, IEEE, 2020.

[82] A. Rahimi, P. Kanerva, L. Benini, and J. M. Rabaey, “Efficient biosignal processing using hyperdimensional computing: Network templates for combined learning and classification of exg signals,” *Proceedings of the IEEE*, vol. 107, no. 1, pp. 123–143, 2018.

[83] A. Rahimi, P. Kanerva, and J. M. Rabaey, “A robust and energy-efficient classifier using brain-inspired hyperdimensional computing,” in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 64–69, ACM, 2016.

[84] U. Raza, P. Kulkarni, and M. Sooriyabandara, “Low power wide area networks: An overview,” *ieee communications surveys & tutorials*, vol. 19, no. 2, pp. 855–873, 2017.

[85] “IEEE 802.11n-2009 - IEEE Standard for Information technology– Local and metropolitan area networks.” https://standards.ieee.org/standard/802_11n-2009.html.

[86] “IEEE 802.15.4-2020 - IEEE Standard for Low-Rate Wireless Networks.” https://standards.ieee.org/standard/802_15_4-2020.html.

[87] T. S. Rappaport, *Wireless communications: principles and practice*, vol. 2. prentice hall PTR New Jersey, 1996.

[88] A. Medeisis and A. Kajackas, “On the use of the universal okumura-hata propagation prediction model in rural areas,” in *VTC2000-Spring. 2000 IEEE 51st Vehicular Technology Conference Proceedings (Cat. No. 00CH37026)*, vol. 3, pp. 1815–1818, IEEE, 2000.

[89] “International Telecommunication Union.” https://www.itu.int//.

[90] O. Afisiadis, M. Cotting, A. Burg, and A. Balatsoukas-Stimming, “On the error rate of the lora modulation with interference,” *IEEE Transactions on Wireless Communications*, vol. 19, no. 2, pp. 1292–1304, 2019.

[91] A. Ultsch, “U* c: Self-organized clustering with emergent feature maps.,” in *LWA*, pp. 240–244, Citeseer, 2005.

[92] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, “Customizing neural networks for efficient fpga implementation,” in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 85–92, IEEE, 2017.

[93] B. Tahir, S. Schwarz, and M. Rupp, “Ber comparison between convolutional, turbo, ldpc, and polar codes,” in *2017 24th international conference on telecommunications (ICT)*, pp. 1–7, IEEE, 2017.

[94] J. Morris, H. W. Lui, K. Stewart, B. Khaleghi, A. Thomas, T. Marback, B. Aksanli, E. Neftci, and T. Rosing, “Hyperspike: Hyperdimensional computing for more efficient and robust spiking neural networks,” in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022.