Karlsruhe Institute of Technology

Reliable, Testable and Efficient Mapping of Hyperdimensional Computing for Resource-constrained Edge AI

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation
von
M. Sc. Mahboobe Sadeghipourrudsari
geboren am 16.08.1994 in Rudsar, Iran

Tag der mündlichen Prüfung: 15. Juli 2026

1. Referent: Prof. Dr. Mehdi B. Tahoori
Karlsruhe Institute of Technology (KIT)

2. Referent: Prof. Maksim Jenihhin
Tallinn University of Technology

Acknowledgments

Reflecting on this journey, I feel incredibly fortunate to have been surrounded by extraordinary individuals whose invaluable support made this dissertation possible.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Dr. Mehdi Tahoori. I could not have asked for a better mentor throughout my PhD journey. His insightful feedback, invaluable discussions, and unwavering support were especially meaningful during challenging times.

My sincere gratitude also goes to my wonderful colleagues at the Chair of Dependable Nano Computing (CDNC). In particular, I would like to thank our secretary, Ms. Iris Schröder-Piepka, for her exceptional ability to keep everything running smoothly. I am also deeply grateful to Dr.-Ing. Dennis Gnad, Dr.-Ing. Jonas Krautter, and Dr.-Ing. Mahta Mayahinia for the many valuable discussions and for their kindness in answering my questions, even after they had moved on to new professional paths outside KIT. Beyond being supportive colleagues, they were also true friends throughout my PhD journey, offering encouragement and support during both the challenging and rewarding moments.

I would also like to thank my colleagues Shanmukha Mangadahalli Siddaramu, Christopher Keßler, Haneen Seyam, Sergej Meschkov, Zhe Zhang, Huashuangyang Xu, Simon Pankner, Florentia Afentaki, Priyanjana Pal, Tara Gheshlaghi, Dina Moussa, Soyed Tuhin Ahmed, Surendra Hemaram, Ali Nezhadi Khelejani, Johannes Reibold, Polykarpos Vergos, and Theofanis Vergos for making our shared time at CDNC both enjoyable and memorable.

My heartfelt thanks also go to my friends Saba Yousefzade, Rahele Mirhashemi, Katayoon Bashrkhah, Nooshin Nosrati, Samira Ahmadi, Gürol Saglam, Hadi Mousanejad, Sepide Saeedi, Bahareh Taghavi, and Alireza Maleki for their constant encouragement and support through the ups and downs of life. Their friendship has meant a great deal to me.

Finally, I would like to express my deepest gratitude to my parents and my siblings, Bahare, Mostapha, and Mahdie, along with my cousins Mahnaz, Mahsa, Mahna, Mina, and Elham. Despite the physical distance between us, their unwavering love, encouragement, and support have always been a tremendous source of strength and comfort.

At a time when the world continues to face uncertainty and conflict, I sincerely hope for greater peace, understanding, and compassion among people, and for a better future for all.

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen – die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Karlsruhe, 20. März 2026
Mahboobe Sadeghipourrudsari

Abstract

Edge Artificial Intelligence (AI) systems require machine learning models that operate under strict power and memory constraints, which limits the practicality of conventional deep learning models on many Internet of Things (IoT) devices. Hyperdimensional Computing (HDC) is a brain-inspired learning paradigm that encodes data as high-dimensional binary hypervectors and performs classification using simple associative operations, making it suitable for energy-efficient edge AI systems. However, practical hardware implementations of HDC still face several challenges, including substantial memory requirements for storing hypervector representations, excessive data movement between memory and logic, limited scalability, and susceptibility to faults and noisy inputs.

This thesis addresses key limitations of existing HDC implementations by proposing a scalable and hardware-efficient framework for edge deployment, with a memory-efficient encoding method that eliminates the need to store large hypervector memories. An on-the-fly hypervector generation mechanism dynamically constructs encoding hypervectors using cyclic digital circuits, removing the requirement for static parameter storage and significantly reducing on-chip memory compared to conventional HDC implementations. Additionally, an optimized pipelined Field-Programmable Gate Array (FPGA) architecture is developed to improve computational efficiency. By combining dynamic hypervector generation, hardware sharing, and pruning techniques, the proposed design reduces resource utilization and increases throughput while maintaining classification accuracy compared to other state-of-the-art HDC accelerators.

To support practical deployment, the proposed architecture is integrated into an open-source end-to-end toolchain that maps Python-based HDC models directly to FPGA implementations without requiring high-level synthesis. The framework supports multiple operating modes, including memoryless encoding, conventional encoding, and pruned HDC models, enabling flexible design exploration, efficient hardware generation, and adaptive edge learning.

Furthermore, to improve system reliability, lightweight fault-tolerance mechanisms are introduced for HDC hardware accelerators. An absolute-summation-based sensitivity analysis identifies vulnerable regions in Class Hypervector (CHV) memory, enabling selective fault correction with reduced hardware overhead compared to existing approaches. Additionally, a memory-free fault correction technique eliminates the need to store check bits while maintaining robustness under high fault rates. Lightweight concurrent error detection mechanisms are also proposed to ensure functional safety by detecting both hardware faults and anomalous inputs, including noisy and out-of-distribution samples, with negligible hardware overhead and no additional latency.

Overall, this thesis advances HDC for practical edge-AI deployment by introducing hardware-aware techniques that improve memory efficiency, scalability, reliability, and functional safety. Memory efficiency is enhanced by eliminating the need for large encoder memories, while scalability is improved through pipelining, hardware sharing, and pruning techniques. Runtime reliability is strengthened using edge-oriented error-correction mechanisms, and functional safety is supported through concurrent detection of hardware faults and anomalous inputs. By preserving the lightweight and highly parallel characteristics of HDC while integrating low-overhead reliability mechanisms, this work enables more efficient, scalable, and dependable HDC accelerators for resource-constrained and safety-critical edge applications.

Zusammenfassung

Edge-Systeme für künstliche Intelligenz (KI) erfordern Modelle des maschinellen Lernens, die unter strengen Leistungs- und Speicherbeschränkungen arbeiten können. Dies schränkt die praktische Nutzung konventioneller Deep-Learning-Modelle auf vielen Geräten des Internets der Dinge (IoT) erheblich ein. *Hyperdimensional Computing (HDC)* ist ein vom Gehirn inspiriertes Lernparadigma, das Daten als hochdimensionale binäre Hypervektoren kodiert und Klassifikation mittels einfacher assoziativer Operationen durchführt. Dadurch eignet sich HDC besonders für energieeffiziente Edge-KI-Systeme. Praktische Hardware-Implementierungen von HDC stehen jedoch weiterhin vor mehreren Herausforderungen, darunter ein erheblicher Speicherbedarf zur Ablage von Hypervektor-Repräsentationen, ein hoher Datentransfer zwischen Speicher und Logik, eine begrenzte Skalierbarkeit sowie eine Anfälligkeit gegenüber Fehlern und verrauschten Eingaben.

Diese Dissertation adressiert zentrale Einschränkungen bisheriger HDC-Implementierungen durch die Entwicklung eines skalierbaren und hardwareeffizienten Frameworks für den Edge-Computing Einsatz. Dabei wird ein speichereffizientes Kodierungsverfahren vorgeschlagen, das die Speicherung großer Hypervektor-Speicher überflüssig macht. Ein Mechanismus zur Laufzeitgenerierung von Hypervektoren konstruiert die Kodierungs-Hypervektoren dynamisch mithilfe zyklischer digitaler Schaltungen. Dadurch entfällt die Notwendigkeit einer statischen Parameterspeicherung, und der On-Chip-Speicherbedarf wird im Vergleich zu konventionellen HDC-Implementierungen erheblich reduziert. Darüber hinaus wird eine optimierte, Pipeline-basierte Architektur auf Basis eines *Field-Programmable Gate Arrays (FPGA)* entwickelt, um die Recheneffizienz zu erhöhen. Durch die Kombination aus dynamischer Hypervektorgenerierung, gemeinsamer Nutzung von Hardware-Ressourcen und Pruning-Techniken reduziert der vorgeschlagene Entwurf die Ressourcennutzung und erhöht den Durchsatz, während die Klassifikationsgenauigkeit im Vergleich zu anderen modernen HDC-Beschleunigern beibehalten wird.

Zur Unterstützung eines praktischen Einsatzes wird die vorgeschlagene Architektur in eine quelloffene End-to-End-Werkzeugkette integriert, die Python-basierte HDC-Modelle direkt auf FPGA-Implementierungen abbildet, ohne High-Level-Synthese zu erfordern. Das Framework unterstützt mehrere Betriebsmodi, darunter speicherlose Kodierung, konventionelle Kodierung und geprunte HDC-Modelle. Dadurch ermöglicht es eine flexible Entwurfsraumerkundung, eine effiziente Hardwaregenerierung sowie adaptives Lernen am Edge.

Darüber hinaus werden leichtgewichtige Fehlertoleranzmechanismen für HDC-Hardwarebeschleuniger eingeführt, um die Systemzuverlässigkeit zu verbessern. Eine auf absoluter Summation basierende Sensitivitätsanalyse identifiziert anfällige Bereiche im Klassen-Hypervektorspeicher und ermöglicht dadurch eine selektive Fehlerkorrektur mit geringerem Hardware-Overhead im Vergleich zu bestehenden Ansätzen. Zusätzlich wird eine speicherfreie Fehlerkorrekturtechnik vorgestellt, die die Speicherung von Prüfbits überflüssig macht und gleichzeitig die Robustheit bei hohen Fehlerraten erhält. Weiterhin werden leichtgewichtige Mechanismen zur nebenläufigen Fehlererkennung vorgeschlagen, um die funktionale Sicherheit zu gewährleisten. Diese Mechanismen erkennen sowohl Hardwarefehler als auch anomale Eingaben, einschließlich verrauschter und außerhalb der Verteilung liegender Eingaben, mit vernachlässigbarem Hardware-Overhead und ohne zusätzliche Latenz.

Insgesamt erweitert diese Dissertation HDC für den praktischen Einsatz in Edge-KI-Systemen durch hardwarebewusste Techniken, welche die Speichereffizienz, die Hardwareskalierbarkeit, die Laufzeitzuverlässigkeit und die funktionale Sicherheit verbessern. Die vorgeschlagenen Beiträge eliminieren große Encoder-Speicher, reduzieren die Nutzung von Hardware-Ressourcen und erhöhen die Robustheit

gegenüber Speicherfehlern, Logikfehlern und unsicheren Eingaben. Durch die Beibehaltung der leichtgewichtigen und hochgradig parallelen Eigenschaften von HDC bei gleichzeitiger Ergänzung zuverlässigkeitssteigernder Mechanismen mit geringem Overhead ermöglicht diese Arbeit effizientere und zuverlässigere HDC-Beschleuniger für ressourcenbeschränkte und sicherheitskritische Edge-Anwendungen.

Contents

Abstract . . . iii

Zusammenfassung . . . v

List of own citations included in this thesis . . . xi

List of other publications not included in this thesis . . . xiii

I. Preliminaries 1

1. Introduction . . . 3
1.1. Contributions and Research Directions . . . 5
1.1.1. Memory-Efficient and Hardware-Friendly HDC Framework . . . 5
1.1.1.1. Memoryless Encoder . . . 5
1.1.1.2. HDC End-to-End Framework . . . 6
1.1.2. Reliability and Functional Safety Solutions . . . 6
1.1.2.1. Memory Fault Correction . . . 6
1.1.2.2. Functional Safety and Fault Detection . . . 6
1.2. Thesis Structure . . . 7

2. Background . . . 9
2.1. Hyperdimensional Computing for Edge Computing . . . 9
2.1.1. Encoding Concept and Hypervectors . . . 10
2.1.2. Classification Concept and Runtime Model . . . 12
2.1.2.1. Learning Phase . . . 12
2.1.2.2. Inference Phase . . . 12
2.2. Runtime Reliability Under Hardware and Data Uncertainty . . . 13
2.2.1. Error-Correcting Codes . . . 13
2.2.2. Out-of-Distribution Data . . . 15
2.3. Related Work . . . 16
2.3.1. Accuracy-Oriented Training and Encoding Methods . . . 16
2.3.2. Hardware-Efficient HDC Implementations . . . 18
2.3.3. Robustness and Reliability in HDC . . . 21
2.4. Experimental Setup . . . 24

II. Contributions 27

3. End-to-End HDC Framework . . . 29
3.1. Motivation and Problem Definition . . . 29
3.2. OTFGEncoder: Memory to Logic . . . 30
3.2.1. ID-level Generation . . . 30

3.2.2. BHV Generation . . . 32
3.3. Generic HDC Architecture . . . 34
3.3.1. Encoding . . . 35
3.3.2. Classification . . . 35
3.3.3. Pipeline . . . 36
3.3.4. Hardware Optimization . . . 36
3.4. End-to-end Software-Hardware Platform . . . 37
3.5. Evaluation . . . 38
3.5.1. Impact on Accuracy . . . 39
3.5.2. Hardware Resource Requirements . . . 39
3.5.3. Comparison with Neural Network FPGA Accelerators . . . 41
3.6. Conclusion . . . 42

4. Non-Uniform Error Correction for Reliable HDC . . . 43
4.1. Motivation and Problem Definition . . . 43
4.2. Proposed Strategy . . . 45
4.2.1. Offline Fault Sensitivity Analysis . . . 45
4.2.2. ECC Encoding Process . . . 47
4.2.3. Runtime Error Correction in HDC . . . 48
4.3. Experimental Results . . . 48
4.3.1. Performance Evaluation . . . 49
4.3.1.1. Inference Accuracy Analyses . . . 49
4.3.1.2. Hardware Implementation . . . 50
4.4. Conclusion . . . 52

5. Memory-Free Error Correction for Edge HDC . . . 53
5.1. Motivation and Problem Definition . . . 53
5.2. Proposed Memory-Free Fault Correction . . . 54
5.2.1. On-the-Fly Memory Generation . . . 55
5.2.2. Offline Processing . . . 57
5.2.3. ECC Check-bit Generator and Decoding Hardware . . . 57
5.3. Experimental Results . . . 58
5.3.1. Performance Evaluation with Faults . . . 58
5.3.1.1. Inference Accuracy Analyses . . . 58
5.3.1.2. Hardware Implementation . . . 60
5.4. Conclusion . . . 62

6. OOD Detection in HDC Accelerators . . . 63
6.1. Motivation and Problem Definition . . . 63
6.2. OOD Detection in HDC Accelerators . . . 64
6.2.1. OOD Detection Methodology . . . 64
6.2.2. Evaluation Metrics . . . 65
6.2.3. OOD Detection Robustness . . . 67
6.3. Experiments and Implementation Results . . . 70
6.3.1. Software Detection Results . . . 70
6.3.2. Hardware Implementation . . . 71
6.3.3. Discussion . . . 72
6.4. Conclusion . . . 72

7. Runtime Uncertainty Detection in HDC . . . **75**
7.1. Motivation and Problem Definition . . . 75
7.2. Reliability Monitoring Framework for Hyperdimensional Computing . . . 76
7.2.1. Fault Detection Methodology . . . 76
7.2.1.1. Working Hypothesis . . . 76
7.2.1.2. Origins of the Score Shift . . . 77
7.2.2. Threshold-Based Reliability Detection . . . 78
7.2.3. Hardware Implementation . . . 79
7.2.4. Evaluation Metrics . . . 80
7.2.5. Reliability Under Parameter Variations . . . 81
7.3. Experiments and Implementation Results . . . 83
7.3.1. Simulation-Based Detection Results . . . 83
7.3.2. Hardware Implementation . . . 86
7.3.3. Discussion . . . 86
7.4. Conclusion . . . 87

8. CHV Fault Detection in HDC . . . **89**
8.1. Motivation and Problem Definition . . . 89
8.2. Proposed CED Method . . . 90
8.2.1. Offline Fault Analysis . . . 91
8.2.2. CED HDC Hardware Implementation . . . 94
8.2.2.1. Configuration . . . 94
8.2.2.2. Metrics calculation . . . 94
8.2.2.3. Fault detection . . . 95
8.3. Experimental Results . . . 95
8.3.1. Fault Detection . . . 96
8.3.2. Hardware Implementation . . . 97
8.3.3. Discussion . . . 98
8.4. Conclusion . . . 98

9. Conclusion and Perspectives . . . **101**
9.1. Conclusion . . . 101
9.2. Future Perspectives . . . 102

References . . . **103**

List of Figures . . . **111**

List of Tables . . . **113**

List of own citations included in this thesis

[1] M. S. Roodsari, J. Krautter, and M. Tahoori, "Otfgencoder-hdc: Hardware-efficient encoding techniques for hyperdimensional computing", in *2024 IEEE Design, Automation and Test in Europe Conference (DATE)*, IEEE, 2024, pp. 1–2.

[2] M. S. Roodsari, J. Krautter, V. Meyers, and M. Tahoori, "E 3 hdc: Energy efficient encoding for hyper-dimensional computing on edge devices", in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2024, pp. 274–280.

[3] "Rcehdc", Accessed: 2025. [Online]. Available: `https://github.com/ms-pr/RCEHDC`.

[4] M. S. Roodsari, S. Hemaram, and M. Tahoori, "Non-uniform error correction for hyperdimensional computing edge accelerators", in *2025 30th IEEE European Test Symposium (ETS)*, IEEE, 2025.

[5] M. S. Roodsari, M. Mayahinia, and M. Tahoori, "Mf-ecc: Memory-free error correction for hyperdimensional computing edge accelerators", in *2026 31st Asia and South Pacific Design Automation Conference (ASPDAC)*, IEEE, 2026.

[6] M. S. Roodsari, V. Meyers, and M. Tahoori, "Ced-hdc: Lightweight concurrent error detection for reliable hyperdimensional computing", in *2025 43rd IEEE VLSI Test Symposium (VTS)*, IEEE, 2025.

[7] M. S. Roodsari, V. Meyers, and M. Tahoori, "Lightweight concurrent out-of-distribution detection in hyperdimensional computing hardware", in *2025 31st IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, IEEE, 2025.

[8] M. S. Roodsari, V. Meyers, and M. Tahoori, "Safehdc: Concurrent uncertainty and fault detection in hyperdimensional computing", *IEEE Transactions on Device and Materials Reliability*, 2026.

List of other publications not included in this thesis

[1] V. Meyers, M. Hefenbrock, M. Sadeghipourrudsari, D. Gnad, and M. Tahoori, "Towards functional safety of neural network hardware accelerators: Concurrent out-of-distribution detection in hardware using power side-channel analysis", in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, 2025, pp. 1413–1419.

[2] B. Sapui, M. Sadeghipourrudsari, and M. B. Tahoori, "Collide & conquer: Side-channel attack on hyper-dimensional computing (hdc) accelerators", in *2025 IEEE International Test Conference in Asia (ITC-Asia)*, IEEE, 2025, pp. 60–65.

[3] V. Meyers, M. S. Roodsari, and M. Tahoori, "Concurrent fault detection for binary neural network accelerators via on-chip voltage monitoring", in *2026 IEEE Design, Automation and Test in Europe Conference (DATE)*, IEEE, 2026.

[4] M. Tahoori et al., "Special sessions-hardware-software co-design for machine learning systems made open-source", in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2025, pp. 23–32.

Part I.

Preliminaries

1. Introduction

The rapid growth of edge devices and Internet of Things (IoT) systems has fundamentally transformed modern computing [5]. A wide range of applications, including wearable health monitoring, industrial sensing, autonomous systems, and smart infrastructure, require real-time Machine Learning (ML) processing capabilities with low-latency inference and privacy-aware processing without relying on cloud-based computation. These requirements drive the deployment of computation closer to the data source and on the device. In this context, edge computing has emerged as a key paradigm for delivering responsive and efficient intelligent services directly at the edge, through localized execution. However, unlike cloud platforms, edge devices operate under strict constraints in power, memory, and computational resources, often limited to milliwatts of power and kilobytes of on-chip memory. These limitations present significant challenges for deploying modern machine learning algorithms efficiently and reliably [6–8].

Conventional ML methods, particularly Deep Neural Networks (DNNs), have achieved remarkable accuracy across diverse domains. However, deploying them on resource-constrained edge platforms remains highly challenging. Although techniques such as quantization (e.g., FP8 or INT8) and model compression have reduced computational and memory demands, these models still require considerable resources and data movement. Consequently, their energy consumption and hardware complexity can limit efficient deployment in ultra-low-power edge environments.

In addition to efficiency challenges, reliability has become a critical concern in modern embedded and edge-computing systems. Operating at reduced supply voltage, a widely adopted energy-saving strategy, significantly increases the probability of memory bit-flips by reducing noise margins and increasing the likelihood of timing violations [9]. Moreover, aggressive technology scaling amplifies the impact of process variations and device-level imperfections [10]. Additional factors such as device aging, thermal stress, electromigration, manufacturing variability, and environmental disturbances introduce further sources of system uncertainty and increase the likelihood of hardware faults [11]. These faults, whether transient or permanent, can directly compromise system functionality and degrade the reliability of deployed models. As a result, the traditional abstraction of near error-free hardware is becoming increasingly difficult to sustain, and machine learning systems must be designed to operate reliably under non-ideal conditions.

Hyperdimensional Computing (HDC) [12] has emerged as a promising alternative to conventional ML paradigms for edge intelligence. Inference computation in HDC can be divided into two parts: the encoding part, which transforms the input vector into a hyperdimensional space of a few thousand dimensions called Query Hypervector (QHV), and the classification part, which finds the most similar

CHV to the encoded QHV. The CHVs are generated during training of the HDC model by bundling the HVs of each class independently over all inputs in the training dataset. Unlike neural networks, which rely on multiply-accumulate operations, HDC uses hardware-friendly primitives such as XOR and Population Count (PopCount), enabling highly parallel and energy-efficient implementations. HDC represents data using high-dimensional hypervectors and distributes information across thousands of dimensions. This distributed representation provides inherent robustness, as information is redundantly encoded and can tolerate noise and random bit errors. In contrast to many conventional machine learning methods, hyperdimensional computing supports single-pass learning, allowing models to achieve high accuracy after only one iteration over the training data. These properties make HDC particularly attractive for data-efficient and resource-constrained edge scenarios.

Despite these advantages, several challenges hinder the practical deployment of HDC systems on edge hardware. Conventional HDC encoding requires storing large hypervector parameters, resulting in a high memory footprint and increased energy consumption, with memory access becoming a key scalability bottleneck. Although HDC is robust to random noise, its hardware implementations remain vulnerable to structured faults, particularly in classification memory, where errors directly degrade accuracy. Voltage scaling further increases fault rates, while traditional mitigation techniques such as Error-Correcting Codes (ECCs) introduce significant memory overhead, which conflicts with the lightweight nature of HDC.

Existing fault-tolerance approaches based on encoding modifications or redundancy often lack generality or incur high overhead. Moreover, they primarily rely on passive error tolerance and do not provide effective runtime detection or correction mechanisms. Current error detection techniques are often misaligned with HDC characteristics or rely on costly computation and memory overhead, leaving a gap in lightweight runtime fault detection. Beyond hardware-induced faults, reliability is further challenged by uncertainties in the input data. In particular, Out-of-Distribution (OOD) inputs caused by noise or environmental variations remain largely unexplored in HDC and are difficult to detect due to the similarity-based nature of inference.

Despite recent research efforts on accuracy improvement, memory efficiency, hardware acceleration, and reliability in HDC systems, existing solutions remain fragmented. Memory- and hardware-efficient designs, lightweight fault-tolerance techniques, and functional safety mechanisms are often addressed separately and are not always compatible with the strict constraints of edge deployment. This thesis addresses these challenges through a set of hardware-aware HDC contributions tailored for resource-constrained edge devices. First, it introduces a memory-efficient hypervector generation method that eliminates the main memory bottleneck in HDC encoding and enables scalable deployment on constrained hardware. Second, it develops an end-to-end hardware-software implementation flow for mapping HDC models to target FPGA platforms. Third, it proposes lightweight runtime mechanisms for correcting and mitigating faults in classification memory. Finally, it introduces functional safety mechanisms that identify unreliable outputs and unsafe operating conditions under both hardware faults and input uncertainties.

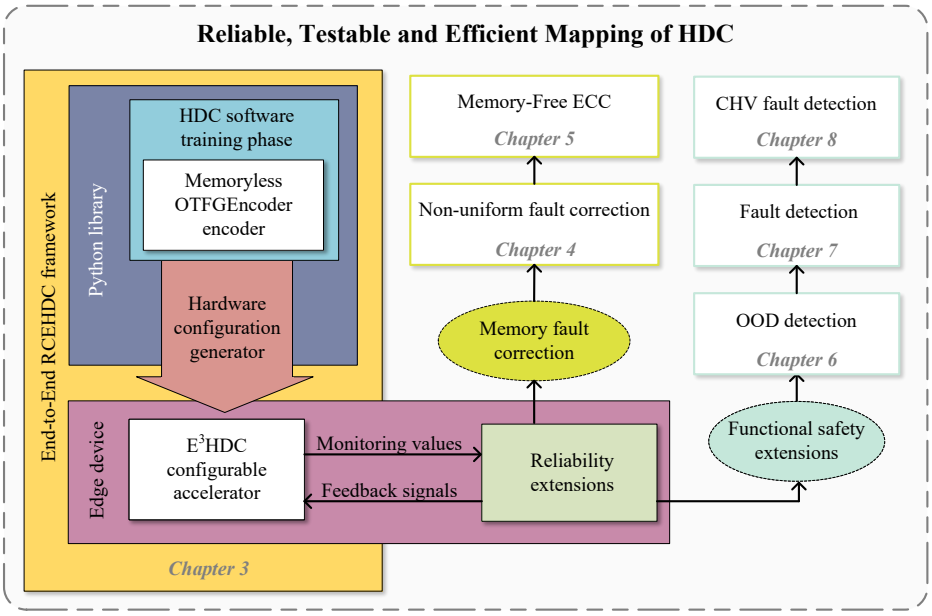


Figure 1.1.: Conceptual illustration of the thesis structure and objectives, highlighting the development of a hardware-aware HDC framework, the reduction of memory dependency, the improvement of computational efficiency, and the enhancement of robust runtime behavior on resource-constrained edge platforms.

Together, these contributions preserve the simplicity and efficiency of HDC while improving its reliability for practical edge-AI deployment.

1.1. Contributions and Research Directions

The contributions of this thesis are summarized in Figure 1.1. The work is organized around two primary research objectives that together advance the practical and efficient deployment of HDC accelerators in edge environments. The first objective focuses on reducing memory dependency and improving hardware efficiency, while the second addresses runtime reliability, all while preserving the lightweight nature of HDC.

1.1.1. Memory-Efficient and Hardware-Friendly HDC Framework

1.1.1.1. Memoryless Encoder

The first research objective is to develop a memory-efficient, hardware-friendly HDC framework that eliminates reliance on large, static hypervector storage in the encoder. This is achieved through an on-the-fly parameter generation mechanism that dynamically produces encoding hypervectors using cyclic digital circuits, removing the need for pre-stored parameter memory [1]. The resulting memoryless encoding significantly reduces the on-chip memory footprint, improves scalability by at least 3×, and lowers power consumption by more than 31% while maintaining accuracy.

1.1.1.2. HDC End-to-End Framework

Building on this encoding strategy, a pipelined hardware architecture is designed to further improve accelerator efficiency. The architecture leverages hardware sharing, dynamic hypervector generation, and sparse computation techniques to minimize resource utilization and maximize throughput. In contrast to High-Level Synthesis (HLS), the proposed framework employs a configurable Register Transfer Level (RTL) design, enabling fine-grained control over low-level architectural aspects such as data paths, memory access patterns, and parallelism. This level of control is critical for achieving high performance and resource efficiency in edge accelerators. The result is an open-source, end-to-end flow that maps Python-based models to synthesizable FPGA bitstreams, enabling seamless hardware–software co-design and supporting real-time edge-AI applications [2, 3].

1.1.2. Reliability and Functional Safety Solutions

1.1.2.1. Memory Fault Correction

Two fault-tolerant mechanisms are introduced to efficiently address transient memory errors in HDC systems. First, a Hamming-weight-based non-uniform sensitivity metric is developed to identify the most vulnerable regions of the CHV memory based on bit-flip susceptibility. This enables targeted mitigation via selective hardware protection, eliminating the need for resource- and time-intensive fault-injection campaigns. The proposed method employs ECC principles to improve bit-flip resilience compared to uniform protection schemes and is validated on FPGA with no additional latency [4].

Although non-uniform ECC reduces memory overhead, it still conflicts with the strict memory constraints of edge devices. To address this limitation, a memory-free ECC technique is introduced that eliminates the need to store check bits, thereby removing secondary fault sources and avoiding storage overhead. This approach preserves inference accuracy under fault rates up to 12× higher than baseline designs, while using up to 27× fewer Look-Up Tables (LUTs), more than 80× fewer registers, and requiring no Digital Signal Processing (DSP), Block Random Access Memory (BRAM), without any additional latency [5]. Both fault-correction methods are released as open-source modules to support rapid integration in edge-AI deployments.

1.1.2.2. Functional Safety and Fault Detection

Functional safety is essential in edge-AI systems, especially in ultra-constrained devices where conventional fault-correction techniques may be too resource-intensive. In these scenarios, identifying unsafe operating conditions and detecting untrusted outputs is critical. To address this need, this thesis introduces lightweight concurrent detection mechanisms that leverage the similarity information already computed during HDC inference.

First, a hardware-oriented Concurrent Error Detection (CED) method is proposed for detecting faults affecting the CHV memory. The method monitors the classification similarity scores between the QHV

and the stored CHVs, detecting abnormal changes in score behavior caused by injected CHV memory faults. Instead of correcting corrupted memory contents, the proposed mechanism flags unreliable inference results before they cause significant accuracy degradation. This makes the approach suitable for resource-constrained edge accelerators, as it introduces negligible hardware overhead and no additional inference latency [6].

Building on this foundation, this thesis also investigates runtime uncertainty detection in HDC accelerators. The proposed SafeHDC framework extends similarity-based monitoring to identify unreliable predictions caused by both input-induced uncertainty, such as noisy or OOD samples, and hardware-induced perturbations evaluated in the corresponding fault scenarios. SafeHDC operates without retraining or auxiliary learning-based detectors and adds only minimal hardware cost, requiring 2 LUTs and 1 register in the evaluated implementation. By flagging outputs likely to be unsafe or untrusted, SafeHDC provides a practical, lightweight safety layer for HDC accelerators deployed in resource-limited and safety-critical edge applications [7, 8].

1.2. Thesis Structure

This thesis is organized into two main parts. Part I provides the necessary background and introduces the main concepts used throughout the thesis. Part II presents the contributions of this work, beginning with the hardware-efficient implementation of HDC and then addressing reliability and runtime safety.

Chapter 2 provides the theoretical and technical background required for the thesis. It introduces the basic principles of HDC, including encoding, training, and similarity-based classification, and reviews the reliability concepts used throughout the thesis, such as error-correcting codes, runtime hardware faults, and out-of-distribution detection. It also surveys related work on accuracy-oriented HDC methods, hardware-efficient HDC implementations, and robustness and reliability techniques. Finally, the chapter briefly describes the common experimental setup used in the thesis.

Chapter 3 presents the proposed end-to-end HDC framework. This chapter focuses on the encoder-memory bottleneck and an FPGA-based software-hardware implementation flow.

Chapter 4 examines the reliability of HDC CHV memory under faults. It introduces a non-uniform error-correction method that provides stronger protection for more sensitive memory regions, reducing overhead compared to uniform ECC protection.

Chapter 5 extends this approach with a memory-free error-correction technique. Instead of storing ECC check bits, the necessary correction information is generated at runtime, reducing memory overhead while maintaining fault resilience.

Chapter 6 addresses the problem of out-of-distribution inputs in HDC accelerators. It presents a lightweight detection method that uses the similarity scores already computed during HDC classification to identify unreliable inputs.

Chapter 7 presents a runtime monitoring framework for detecting unreliable HDC predictions. The method addresses both input uncertainty and hardware-induced faults using simple similarity-based indicators.

Chapter 8 focuses on concurrent error detection for CHV memory. It monitors classification behavior during inference and detects faulty outputs with minimal hardware overhead.

Finally, chapter 9 summarizes the main findings of the thesis and discusses possible directions for future research.

2. Background

Edge computing has become an important computing paradigm for modern intelligent systems, particularly in IoT environments where large volumes of data are generated close to sensors and actuators. In many practical scenarios, continuously transmitting raw data to centralized cloud infrastructures is impractical because of latency, privacy, security, bandwidth, and energy constraints [13]. As a result, ML models are increasingly expected to operate directly on edge devices to support real-time and autonomous decision-making. However, deploying ML at the edge remains challenging because such platforms are constrained by limited power budgets, memory capacity, hardware resources, and computational throughput.

Conventional deep learning models have achieved high accuracy across many application domains, but their computational and memory requirements often conflict with the constraints of resource-limited edge platforms. DNNs typically require large memory footprints, many multiply–accumulate operations, and costly training or inference procedures. These requirements limit their deployment in low-power embedded systems, especially in real-time or safety-critical applications. Therefore, alternative learning paradigms are needed that can provide useful classification capability while reducing implementation complexity and hardware cost.

Table 2.1 provides a qualitative comparison of representative learning paradigms for edge deployment, including conventional machine-learning models such as support vector machines (SVM) and random forests (RF), DNNs, and Hyperdimensional Computing (HDC). The comparison focuses on properties that are important for edge systems: data efficiency, computational cost, robustness to uncertainty, and classification accuracy. The table should be interpreted as a high-level comparison of typical trends rather than a universal ranking across all datasets and implementations. Among these paradigms, HDC provides a favorable trade-off between efficiency, robustness, and implementation simplicity, which makes it a promising candidate for resource-constrained edge systems.

2.1. Hyperdimensional Computing for Edge Computing

Hyperdimensional Computing (HDC) [12] is a brain-inspired computing paradigm that represents information using very high-dimensional vectors, commonly called Hypervectors (HVs). Unlike conventional neural networks, which rely heavily on arithmetic operations such as multiplication and nonlinear activation functions, HDC mainly uses hardware-friendly operations such as bitwise XOR, accumulation, and PopCount. Information in HDC is distributed across thousands of dimensions.

Table 2.1.: Qualitative comparison of machine-learning paradigms for edge deployment.

Features	Conventional ML (e.g., SVM, RF)	Deep Neural Networks	Hyperdimensional Computing (HDC)
Data efficiency	Moderate	Low	High
Computational cost	Low	High	Low
Error robustness	Low	Moderate	High
Accuracy	Moderate	High	Moderate

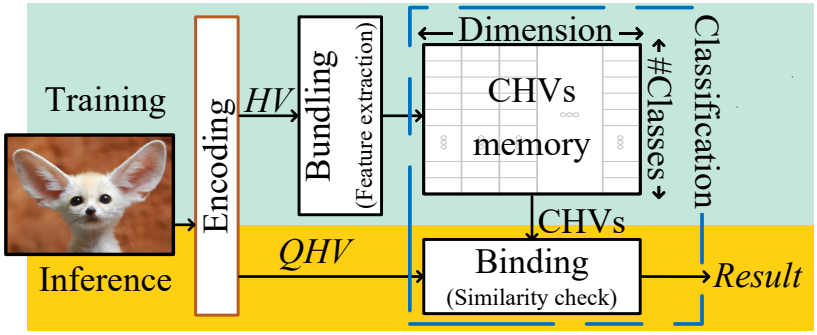


Figure 2.1.: Overview of Hyperdimensional Computing (HDC), comprising two main sections: encoding and classification. The input is first encoded, and then the most similar class to the encoded input is selected as the classification result.
HV: Hypervector, CHV: Class Hypervector, QHV: Query Hypervector

This distributed representation makes HDC naturally tolerant to noise and partial corruption, because perturbations in a subset of dimensions usually affect only part of the representation and may not change the final classification decision.

This property is useful for edge systems, where noisy operating conditions, limited training data, and strict resource constraints must be considered simultaneously. Compared with many deep-learning models, HDC can often be trained with fewer samples and implemented using lightweight operations such as bitwise logic and simple accumulation [14]. These characteristics make HDC attractive for hardware-oriented implementations on constrained platforms.

As shown in Figure 2.1, a typical HDC classifier consists of two main stages: encoding and classification. The encoding stage maps input data into the hyperdimensional space and is used during both training and inference. The classification stage then compares the encoded query hypervector with stored CHVs and selects the most similar class. The following subsections describe these two stages and their implications for edge-oriented hardware design.

2.1.1. Encoding Concept and Hypervectors

The encoding stage maps low-dimensional input data into the hyperdimensional space. Each feature or feature value is represented by a hypervector, and these hypervectors are combined to form a single representation of the input sample. The choice of encoding method directly affects memory usage, computational cost, and hardware complexity. Three common encoding approaches are random projection [15], base-level encoding [16], and permutation-based encoding [14].

Random Projection Encoding: Random projection encoding maps input features directly into the hyperdimensional space using a fixed random projection matrix, often referred to as the Base Hypervector (BHV) matrix. The input vector is projected into a high-dimensional binary or bipolar representation through matrix multiplication, followed by binarization or thresholding. The rows or columns of the projection matrix are typically generated randomly and are quasi-orthogonal with high probability, which helps preserve the independence of different input features in the hyperdimensional representation. Although this approach is conceptually simple and compact, its reliance on matrix multiplication can make it costly for hardware-oriented implementations, especially on resource-constrained platforms.

Base-Level Encoding: Base-level encoding separates value encoding from positional encoding. First, each quantized feature value is mapped to a representative hypervector using an ID-level matrix. These hypervectors are constructed so that nearby input values produce similar hypervectors, while distant values become increasingly dissimilar and ideally approach orthogonality.

Positional information is then incorporated using a BHV matrix. Each value hypervector is bound to its corresponding positional hypervector, commonly through element-wise multiplication or XOR depending on the representation. The resulting bound hypervectors are accumulated to form the final encoded representation. This structure is well suited to hardware implementation because it mainly relies on simple bitwise operations and PopCount-based accumulation. However, it requires storing both the ID-level matrix and the BHV matrix, which can become a memory bottleneck on edge devices.

Permutation-Based Encoding: Permutation-based encoding reduces memory requirements by avoiding the need to store an explicit positional BHV matrix. Instead, positional information is introduced through permutation operations, commonly implemented as circular shifts over hypervectors. After feature values are mapped to hypervectors using an ID-level matrix, a position-dependent permutation is applied according to the feature index. The permuted hypervectors are then accumulated to generate the final encoded representation.

This approach reduces memory overhead compared with base-level encoding, but it introduces additional hardware complexity. Permutation units, such as barrel shifters, can increase routing complexity, area cost, critical-path delay, or latency depending on the implementation. Thus, permutation-based encoding trades memory savings for additional datapath complexity.

A comparison of the encoding methods is shown in Table 2.2. The most suitable encoding method depends on the target application and hardware constraints. Base-level encoding can be attractive when simple bitwise operations are preferred, but it requires larger lookup memories. Permutation-based encoding reduces memory usage but may increase logic and routing cost. Random projection uses less stored hypervector memory, but the projection operation can be expensive if implemented directly in hardware.

Table 2.2.: Comparison of HDC encoding methods in terms of memory requirements and computational operations.

Encoding method	ID matrix	BHV matrix	Required memory (bits)	Operations
Permutation	✓	×	$2^Q \times D$	Shift + PopCount
Base-level	✓	✓	$2^Q \times D + F \times D$	XOR + PopCount
Random projection	×	✓	$F \times D$	Matrix multiplication

F: feature size D: hypervector dimension Q: number of input feature quantization bits

2.1.2. Classification Concept and Runtime Model

After an input sample is encoded into the hyperdimensional space, learning and inference follow a common processing structure. The encoder determines how input hypervectors are generated, while the classification stage compares encoded samples with stored class representations. In most HDC classifiers, learning relies on bundling operations, whereas inference relies on similarity or distance computation between a query hypervector and the stored CHVs.

2.1.2.1. Learning Phase

During learning, each training sample is first encoded into a hypervector and then accumulated into the corresponding Class Hypervector (CHV). For each class, the encoded hypervectors of all training samples belonging to that class are combined to form a representative class prototype. This operation is commonly referred to as bundling.

Bundling typically consists of element-wise addition followed by normalization or thresholding. In its simplest form, this process can be completed in a single pass over the training dataset, which gives HDC low training complexity compared with iterative optimization methods.

In practice, CHVs may be stored using multi-bit values to preserve richer accumulated information. However, binary CHVs are often preferred for hardware-oriented implementations because they reduce memory usage and enable efficient bitwise processing. For this reason, this thesis focuses on binary CHVs, which align well with the constraints of edge-device implementation.

2.1.2.2. Inference Phase

During inference, each input sample is encoded into a query hypervector, denoted as the QHV. The QHV is compared against all stored CHVs, and the class with the highest similarity is selected as the predicted label. For binary CHVs, Hamming distance is commonly used, whereas cosine similarity is more common for multi-bit or bipolar representations.

In binary HDC, Hamming distance is particularly suitable because it directly counts the number of differing bits between two hypervectors. It can be implemented efficiently using XOR followed by PopCount. The class with the smallest Hamming distance, or equivalently the highest similarity, is selected as the output class.

Because comparisons between the QHV and different CHVs are independent, the classification stage is highly parallelizable. This property allows inference to be implemented using simple digital logic and makes HDC suitable for hardware acceleration on constrained edge platforms.

2.2. Runtime Reliability Under Hardware and Data Uncertainty

Runtime reliability is essential when ML and HDC-based systems are deployed in real-world environments. During operation, uncertainty can arise from memory errors, transient bit flips, permanent hardware faults, device variation, or input samples that differ from the training distribution. In an HDC classifier, such uncertainty is especially important because the stored CHVs are directly used during similarity computation. A corrupted CHV bit or an unfamiliar input sample can therefore change the similarity ranking between classes and lead to an unreliable prediction.

This section discusses two mechanisms related to runtime reliability. First, Error-Correcting Codes (ECCs) are introduced as a conventional method for protecting stored data against memory faults. Second, Out-of-Distribution (OOD) detection is discussed as a mechanism for identifying inputs that fall outside the learned data distribution.

2.2.1. Error-Correcting Codes

Error-Correcting Codes (ECCs) [17] improve the reliability of memory and computing systems by adding structured redundancy to stored or transmitted data. Rather than protecting each bit independently, ECCs represent information using codewords that belong to a constrained set of valid binary patterns. This redundancy enables the detection of corrupted data and, in many cases, the reconstruction of the original information [18, 19].

A common class of ECCs is the linear block code. In a linear block code, a message consisting of k information bits is represented by a vector m and mapped to an n-bit codeword c, where $n > k$. This encoding is performed by multiplying the message vector by a generator matrix G:

$$c = mG. \tag{2.1}$$

For binary linear block codes, all operations are carried out over the finite field $\mathbb{F}_2 = \{0, 1\}$, meaning that addition and multiplication are performed modulo 2.

The extra $n - k$ bits in the codeword are parity bits. They introduce redundancy, which allows the receiver to detect and, in some cases, correct transmission errors. After transmission, the receiver obtains a vector r, which may differ from the original codeword due to errors introduced by the channel. To check whether r satisfies the code constraints, the receiver computes the syndrome using the parity-check matrix H:

$$s = rH^T. \tag{2.2}$$

The syndrome indicates whether the received vector satisfies the parity-check equations. If $s = 0$, then r belongs to the codeword space and is considered a valid codeword. If $s \neq 0$, at least one parity-check equation is violated, indicating that an error has occurred.

The error-correction capability of a linear block code is determined by its minimum Hamming distance $d_{\min}$, defined as the smallest number of bit positions in which any two valid codewords differ. A code with minimum distance $d_{\min}$ can correct all error patterns affecting up to

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor \tag{2.3}$$

bits. Within this correction radius, the received word is closer to the transmitted codeword than to any other valid codeword. A larger minimum distance also improves error-detection capability because more corrupted patterns fall outside the set of valid codewords. This distance-based view is relevant for hardware systems, where faults may include soft errors, stuck-at faults, and transient bit flips caused by radiation, voltage fluctuations, or device aging.

Hamming codes are among the earliest and most widely used linear block codes for single-error correction. For a binary Hamming code parameterized by p parity-check bits, the code length is

$$n = 2^p - 1, \tag{2.4}$$

and the number of information bits is

$$k = n - p = 2^p - p - 1. \tag{2.5}$$

The corresponding code has minimum Hamming distance $d_{\min} = 3$, which enables correction of any single-bit error. The parity-check matrix is constructed so that each bit position corresponds to a unique nonzero p-bit syndrome. Therefore, each single-bit error produces a unique syndrome, allowing the erroneous bit position to be identified and corrected.

Because of this simple syndrome-based decoding process, Hamming codes are attractive for low-cost hardware implementations where area, latency, and power consumption are tightly constrained. In memory systems, extended Hamming codes are often used to provide single-error correction and double-error detection (SEC-DED) by adding one overall parity bit [20].

Although ECCs are well-established techniques for improving runtime reliability [21], they achieve this protection by introducing redundant parity bits. For a codeword of length n protecting k information bits, the additional storage cost is $n - k$ bits, corresponding to a memory overhead ratio of $(n - k)/k$. This overhead is especially important for HDC systems, where CHVs can dominate memory usage. Therefore, applying ECCs to HDC memory requires a careful trade-off between reliability improvement and memory efficiency.

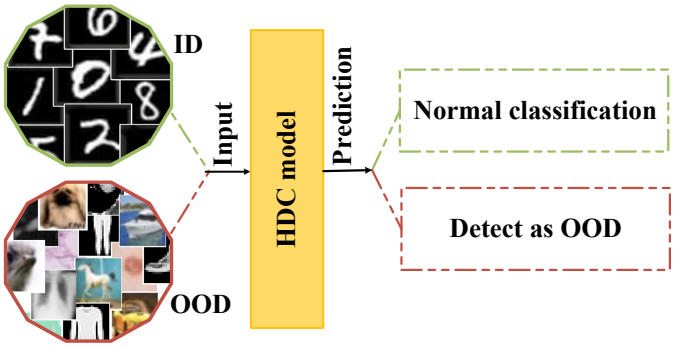


Figure 2.2.: Overview of OOD detection in an HDC-based classifier.

2.2.2. Out-of-Distribution Data

Most supervised machine-learning models are trained under the assumption that training and test samples are drawn from the same underlying distribution. In real-world deployments, this assumption is often violated. During operation, especially in edge and safety-critical systems, a model may encounter inputs that differ substantially from the data observed during training [22, 23]. These inputs are referred to as Out-of-Distribution (OOD) samples, as illustrated in Figure 2.2.

If OOD samples are not detected, the model is still forced to assign them to one of the known training classes, even though they may not belong to the learned data distribution. This can cause a silent failure: the system produces an incorrect prediction without indicating uncertainty. Such behavior is particularly problematic in safety-critical applications such as medical diagnosis, autonomous driving, and industrial monitoring, where an incorrect but confident decision can have severe consequences.

Out-of-Distribution (OOD) detection addresses this problem by allowing a model to recognize when an input lies outside its domain of competence. Once such an input is detected, the system can reject the prediction, activate a fallback mechanism, request additional sensing, or defer the decision to a human operator. This capability is important in open-world environments, where it is impossible to anticipate all possible input variations during training. Because the space of possible OOD samples is effectively unbounded, explicitly training on every anomaly is infeasible. Instead, OOD detection aims to distinguish Identifier (ID) samples from unknown inputs using information such as confidence scores, distances in representation space, uncertainty estimates, or similarity measures.

For edge devices, OOD detection is not only a model-reliability problem but also a hardware-design problem. Edge devices may not be able to use large auxiliary models, repeated inference, gradient-based preprocessing, or cloud-based fallback mechanisms. Therefore, an OOD detector for edge deployment must be lightweight enough to operate alongside the main classifier while still providing useful information about prediction trustworthiness.

Existing OOD detection methods for Neural Networks (NNs) commonly apply a scoring function to internal model quantities such as softmax probabilities, logits, pre-softmax activations, or feature representations [24–27]. Other methods modify the training procedure or use additional learning-based

components to improve OOD separability [28–30]. These approaches have been effective in neural-network settings, but many of them assume access to activations, logits, gradients, or training procedures that are not directly available or desirable in a lightweight HDC hardware implementation.

The maximum softmax probability baseline detects OOD samples using the largest class probability produced by a neural network; samples with low maximum probabilities are treated as less reliable and may be flagged as OOD [24]. A related approach uses the maximum logit value before the softmax layer, where lower maximum logits may indicate that the input is less compatible with the learned classes [25]. Entropy-based methods measure the uncertainty of the output probability distribution, with higher entropy indicating greater uncertainty [26]. Energy-based methods replace softmax confidence with an energy score, where ID samples are expected to have lower energy and OOD samples higher energy [27]. ODIN [22] further improves softmax-based detection by applying temperature scaling and a small input perturbation to increase separation between ID and OOD score distributions.

Although these methods provide useful baselines, their computational assumptions are not always compatible with constrained edge hardware and they are not specifically designed for HDC. This thesis therefore investigates a lightweight, ML-free OOD detection mechanism for HDC. Instead of introducing an auxiliary model or additional neural-network-style computations, the proposed method uses similarity information already produced during HDC associative search. To the best of our knowledge, only limited prior work has explored OOD detection directly within lightweight HDC classifiers using only the classifier's native similarity scores. The goal is to improve robustness to distributional shift and support safer runtime behavior while preserving the low-memory and low-computation properties required for edge deployment.

2.3. Related Work

This section reviews prior work on HDC with emphasis on the research directions most relevant to this thesis. Existing studies are grouped into three areas: methods that improve classification accuracy, hardware-oriented designs that reduce implementation cost, and robustness techniques that address memory faults, adversarial behavior, and runtime uncertainty. This organization clarifies how prior work improves either model quality or implementation efficiency, and it provides the basis for positioning the proposed edge-oriented and reliability-aware HDC design.

2.3.1. Accuracy-Oriented Training and Encoding Methods

The classification accuracy of HDC models is strongly affected by the hypervector dimension, the encoding scheme, and the training procedure used to construct CHVs. A direct way to improve accuracy is to increase the hypervector dimension. However, larger dimensions also increase memory consumption, arithmetic cost, and hardware complexity [31]. Therefore, several studies have focused on improving the

learning rule or adapting the encoder so that higher accuracy can be achieved without relying only on larger hypervectors.

OnlineHD [32] improves the baseline HDC learning procedure by introducing an adaptive online training strategy. In conventional single-pass HDC training, all encoded samples are uniformly accumulated into their corresponding CHVs. This can lead to model saturation, where frequent patterns dominate the class representation and reduce the influence of less frequent but informative samples. OnlineHD addresses this issue by updating the CHVs according to the similarity between the encoded sample and the current model.

For a sample hypervector $\mathbf{HV}_i$ with true class y_i and incorrectly predicted class $\hat{y}_i$, OnlineHD updates the corresponding CHVs as follows:

$$\mathbf{C}_{y_i} \leftarrow \mathbf{C}_{y_i} + \eta \left(1 - \rho(\mathbf{HV}_i, \mathbf{C}_{y_i})\right) \mathbf{HV}_i, \tag{2.6}$$

$$\mathbf{C}_{\hat{y}_i} \leftarrow \mathbf{C}_{\hat{y}_i} - \eta \left(1 - \rho(\mathbf{HV}_i, \mathbf{C}_{\hat{y}_i})\right) \mathbf{HV}_i, \tag{2.7}$$

where η is the learning rate and $\rho(\cdot, \cdot)$ denotes cosine similarity:

$$\rho(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}. \tag{2.8}$$

By using similarity-aware updates, OnlineHD provides a compromise between the efficiency of single-pass HDC training and the higher accuracy of iterative retraining methods. However, this improvement introduces additional arithmetic compared with simple binary accumulation.

RefineHD [33] extends this idea by using a more selective update mechanism that considers both the similarity between the sample and its CHV and the relationship between the top competing classes. For a misclassified sample, RefineHD defines a weighting factor based on the similarities of the two most similar classes:

$$w_i = 1 - \rho_{\text{top1}} - \rho_{\text{top2}}, \tag{2.9}$$

where ρ_{top1} and ρ_{top2} denote the similarities to the two most similar CHVs. The CHVs are then updated as

$$\mathbf{C}_{y_i} \leftarrow \mathbf{C}_{y_i} + \left(1 - \rho(\mathbf{HV}_i, \mathbf{C}_{y_i})\right) w_i \mathbf{HV}_i, \tag{2.10}$$

$$\mathbf{C}_{\hat{y}_i} \leftarrow \mathbf{C}_{\hat{y}_i} - \left(1 - \rho(\mathbf{HV}_i, \mathbf{C}_{\hat{y}_i})\right) w_i \mathbf{HV}_i. \tag{2.11}$$

This top-2-aware weighting allows RefineHD to incorporate inter-class relationships during learning. It also selectively updates correctly classified samples when their similarity to the corresponding class is not sufficiently high. As a result, RefineHD can preserve informative patterns while reducing unnecessary accumulation of redundant samples. The cost of this selectivity is additional similarity evaluation and control complexity.

DistHD [34] improves HDC accuracy from a different perspective by making the encoder learner-aware. While OnlineHD and RefineHD mainly modify the CHV update rule, DistHD adapts the representation space itself. Conventional HDC methods typically use static encoders that remain unchanged after initialization. In contrast, DistHD identifies hypervector dimensions that negatively affect classification and regenerates them during learning.

The main idea of DistHD is based on top-2 classification and dimension regeneration. For each training sample, the two most similar CHVs are identified using cosine similarity. The method then analyzes dimensions that contribute to confusion between the correct and competing classes. Dimensions that are closer to the incorrect class than to the true class are considered misleading and are regenerated using newly sampled random projections. This allows the encoder to evolve with the model and can improve the quality of the hyperdimensional representation. However, this benefit requires iterative refinement and encoder updates, which complicate direct deployment in simple hardware datapaths.

Overall, OnlineHD, RefineHD, and DistHD improve HDC accuracy through complementary mechanisms. OnlineHD and RefineHD modify the CHV update rule to reduce saturation and preserve informative samples, whereas DistHD adapts the encoder itself to improve representation quality. These methods demonstrate that both training and encoding have a direct effect on class separability.

However, these accuracy gains usually come with implementation costs. Compared with baseline binary HDC, which mainly relies on XOR, accumulation, and Hamming-distance comparison, adaptive and retraining-based methods may require multi-bit CHVs, weighted updates, cosine-similarity computation, iterative refinement, or encoder regeneration. These operations increase arithmetic complexity, memory usage, and control overhead. Therefore, although these methods are effective for improving accuracy, they are not always aligned with the strict area, power, and memory constraints of edge-device implementations.

2.3.2. Hardware-Efficient HDC Implementations

HDC is well suited for hardware acceleration because its core operations, including binding, bundling, and associative search, are highly parallel and can often be implemented using low-precision arithmetic or bitwise logic. However, practical HDC accelerators are frequently limited by the encoding stage, where low-dimensional input features are projected into high-dimensional hypervectors. Existing hardware-oriented work therefore focuses on reducing encoding cost, improving memory efficiency, exploiting parallelism, or moving computation closer to memory.

FPGA-based HDC accelerators are one of the most active implementation directions because they can exploit dimension-wise parallelism while allowing architectural customization. F5-HD [35] automates the generation of FPGA-based HDC accelerators. Instead of requiring manual RTL design, F5-HD generates synthesizable Verilog from high-level application and hardware parameters. The framework supports training, retraining, and inference, and explores different HDC precisions, including fixed-point,

power-of-two, and binary models. Although such automation improves design productivity, the generated architecture may not always be optimal for the strictest edge-device constraints.

Another line of FPGA work reduces the cost of encoding and similarity search. Salamat et al. [36] observe that consecutive inputs and CHVs often share intermediate computations and exploit this reuse to reduce both encoding and associative-search complexity. BRIC [15] focuses on improving the efficiency of random projection encoding. Unlike conventional random projection methods that may require dense matrix operations and irregular memory access, BRIC introduces a locality-based random projection with a predictable memory access pattern. Given an input feature vector, each hypervector dimension in BRIC is computed using a local N-gram window followed by a sign function:

$$h_i = \text{sign}\left(\sum_{j=0}^{N-1} f_{i+j} p_{i+j}\right), \tag{2.12}$$

where f_{i+j} denotes an input feature and p_{i+j} denotes the corresponding projection value. This locality-based formulation enables computation reuse and avoids irregular access to a large random projection matrix. Furthermore, BRIC operates in the binary domain, allowing training and inference to be implemented using lightweight operations.

Shear [37] is an algorithm-hardware co-design framework that improves HDC efficiency, especially in the encoding stage. It introduces approximate encoding techniques that exploit the error resilience of HDC representations to reduce hardware resource usage, such as LUT consumption, with limited accuracy loss. Shear also provides a parallel FPGA architecture and a supporting software framework for training with these approximations. This shows that controlled approximation can be useful in HDC hardware, but the resulting design still depends on the selected approximation strategy and target accuracy loss.

LookHD [38] redesigns permutation-based encoding to reduce the dominant cost of conventional HDC implementations. Traditional permutation operates over full high-dimensional hypervectors, which can lead to high power consumption and long latency. LookHD partitions the input feature vector into smaller chunks and encodes each chunk independently. It precomputes and stores all possible encoded chunk hypervectors, reducing encoding to indexed codebook access rather than repeated hypervector generation or rotation. This improves runtime efficiency, but shifts part of the cost to memory storage for the codebooks.

HD2FPGA [39] presents an end-to-end FPGA-based framework using HLS for random projection encoding. It reduces the hardware cost of encoding by decomposing the computation into smaller chunks rather than processing full high-dimensional vectors in a single cycle. The design uses a sliding-window approach in which only a subset of features and hypervector dimensions (C features $\times$ R dimensions) is processed per cycle. This allows the encoder to accumulate partial results over multiple cycles, reducing the need for wide parallel datapaths and large combinational logic. The resulting architecture

is time-multiplexed and pipelined, which improves scalability across FPGAs with different resource constraints.

SupportHDC [40] improves hardware efficiency by using sparse ternary hypervectors instead of dense binary hypervectors. The framework generates hypervectors with values in $\{-1, 0, +1\}$ through a tunable thresholding mechanism, allowing designers to trade accuracy for sparsity and reduced hardware cost. From a hardware perspective, sparsity is beneficial because operations involving zero-valued dimensions can be skipped, reducing switching activity, memory accesses, and energy consumption. SupportHDC therefore shows that controlled sparsity can serve as a practical design parameter for low-power HDC implementations.

FACH [41] reduces the computational cost of FPGA-based multi-bit HDC inference by simplifying similarity computation between query and CHVs. Realistic classification tasks may require non-binary CHVs to preserve accuracy, but comparing such hypervectors can involve costly dot-product or cosine-similarity operations. FACH clusters the values of each CHV into a small set of representative centroids. During inference, query dimensions associated with the same centroid are accumulated first, and only the accumulated result is multiplied by the centroid. This reduces the number of multiplications from the hypervector dimension to the number of centroids.

ASIC-based HDC designs have also been proposed for low-power edge and IoT systems. TinyHDC [42] implements random projection encoding in a dedicated ASIC architecture and targets datapath simplification for energy-constrained devices. Datta et al. [43] demonstrate programmable low-power HDC architectures for edge and IoT workloads. Although these works are not FPGA implementations, they provide useful architectural insights: efficient HDC hardware benefits from shallow datapaths, bit-level parallelism, compact associative memory, and configurable support for different hypervector dimensions and class counts.

In-Memory Computing (IMC) is another important direction because many HDC operations are dominated by memory-resident vector processing. In conventional von Neumann architectures, repeatedly moving high-dimensional hypervectors between memory and compute units can dominate both energy and latency. IMC architectures address this bottleneck by performing operations such as encoding and binding inside or near the memory array.

HDC-IM [44] maps HDC operations to Resistive Random Access Memory (ReRAM) crossbars and introduces block encoding and mapping to handle long hypervectors efficiently. The work also studies ReRAM stuck-at faults and shows that HDC can maintain better accuracy than ReRAM-based neural-network accelerators under device faults.

Device-level IMC implementations have also been explored. Li et al. [45] investigate HDC kernels based on 3D vertical ReRAM, showing how multiply-add-permute-style operations can be supported by emerging memory structures. Kazemi et al. [46] explore FeFET-based IMC and demonstrate multi-bit content-addressable memory operations for HDC, aiming to preserve software-level accuracy while reducing energy and latency. These works show that HDC can benefit from emerging memory

technologies, but they also require careful mapping between the algorithm and the physical memory structure.

Recent work has identified mapping inefficiencies between HDC algorithms and realistic IMC arrays. MEMHD [47] shows that conventional HDC associative memory, where each class is represented by a single high-dimensional prototype, can poorly utilize fixed-size IMC arrays and require many computation cycles. To address this issue, MEMHD proposes a multi-centroid associative memory that better matches the dimensions and column structure of IMC arrays. This result highlights that IMC-based HDC often requires algorithm-architecture co-design rather than only circuit-level acceleration.

Overall, FPGA-, ASIC-, and IMC-based HDC implementations show that hardware efficiency depends on architectural parallelism, memory organization, and algorithm-hardware co-design. The most effective designs reduce memory traffic, simplify encoding, exploit binary or low-precision operations, and adapt the dataflow to the structure of HDC computations. However, important limitations remain. Encoding may still require stored projection parameters, codebooks, or permutation hardware. Permutation can be costly in terms of routing, latency, and power. Automated frameworks based on HLS or broad design-space exploration improve usability, but they may not produce the smallest or most efficient architecture for a fixed edge target. Some designs also depend on DSP blocks or nontrivial arithmetic for encoding or similarity computation, which can limit their suitability for highly constrained devices.

This thesis targets this gap by developing an edge-oriented HDC design that reduces encoder memory and avoids expensive permutation structures. Unlike approaches that rely on stored encoder parameters, large projection memories, or codebooks, the proposed encoder removes the memory block from the encoding stage. The thesis further develops an end-to-end framework that connects an open-source PyTorch-based training library [48] to FPGA implementation files, allowing a trained HDC model to be translated into hardware. By combining pipelining and resource sharing, the proposed design aims to reduce memory cost and hardware complexity while preserving suitability for constrained edge deployment.

2.3.3. Robustness and Reliability in HDC

For edge deployment, hardware efficiency alone is not sufficient. The deployed system must also remain dependable under memory faults, device variation, noisy inputs, and distributional shift. This is particularly important for edge systems, which often operate in uncontrolled environments and have limited opportunities for maintenance, recalibration, or cloud-based correction [49]. Reliability mechanisms must therefore fit within the same memory, power, and computation constraints that motivate hardware-oriented HDC in the first place.

A major advantage of HDC is its natural tolerance to noise and component-level perturbations, which comes from the distributed and holographic nature of high-dimensional representations [14]. However, this intrinsic robustness is not sufficient under stronger fault models, adversarial perturbations, or

input-distribution shifts. Recent work has therefore introduced explicit algorithm-level mechanisms for improving HDC reliability beyond the redundancy provided by high-dimensional representation alone.

One line of work improves robustness by increasing hypervector separability. HyperMetric [50] identifies the Hamming-distance margin between CHVs as a source of error resilience and uses metric learning to enlarge this margin. Instead of relying on hardware redundancy, HyperMetric optimizes the encoding and training process so that samples from the same class remain close while samples from different classes are pushed farther apart. This improves robustness through better class separation, but it also introduces an additional optimization objective.

A related group of studies focuses on hardware-induced errors, especially faults in associative memory. Zhang et al. [51] systematically evaluate how bit errors in associative memory affect HDC accuracy under different dimensions, data widths, datasets, and error rates. Although this work is mainly a characterization study, it also explores low-cost masking of corrupted bits, showing that algorithm-aware error masking can improve reliability without full hardware redundancy. Iwasaki et al. [52] address permanent faults in memristor-based HDC inference accelerators by computing checksums for CHVs and using redundancy to replace faulty memory regions.

ScaleHD [53] improves fault tolerance by scaling and clamping the elements of CHVs. The method focuses on associative memory because stored CHVs are directly used during inference and are therefore vulnerable to memory faults. By keeping CHV elements within a controlled range, ScaleHD reduces the impact of individual bit-flip errors and makes the representation more uniformly resilient across dimensions.

DependableHD [54] improves resilience by injecting noise during training and increasing decision margins, helping inference remain stable under memory failures. More recently, Mejri et al. [55] proposed checksum hypervector encoding and cross-hypervector clustering to detect and suppress corrupted dimensions in CHV memory during inference. In this case, robustness is improved at the inference-algorithm level by removing unreliable hypervector components from similarity computation instead of correcting faults only at the circuit level.

DeeperHD [56] improves error resilience by integrating a DNN-based feature extractor before the HDC encoding stage. Convolutional and pooling layers produce features that are then processed by the HDC module. Although this can improve robustness and accuracy, it also increases model complexity. Moreover, faults in the neural feature extractor can propagate to the HDC stage and affect the final prediction.

ConformalHDC [57] addresses reliability from the perspective of uncertainty-aware prediction and abstention. Instead of always assigning an input to the most similar class, the method converts similarity values into a nonconformity score that reflects both the absolute similarity to a candidate class and the relative separation from competing classes. A calibration set is then used to determine an acceptance threshold. Inputs that do not conform to the calibrated decision region, including ambiguous samples,

outliers, adversarial perturbations, or OOD inputs, can be rejected rather than forced into an unreliable prediction.

Robustness has also been studied under adversarial and poisoning attacks. Yang and Ren [58] showed that HDC classifiers can be vulnerable to small adversarial perturbations despite their tolerance to random bit errors. Later work developed more systematic testing and defense frameworks. TestHD [59] generates adversarial samples for HDC models and uses them to improve adversarial robustness through adversarial retraining. PoisonHD [60] studies data-poisoning attacks that exploit the structure of HDC associative memory and proposes HDC-based data sanitization to recover classification accuracy.

A separate line of work uses HDC as a reliability mechanism for other machine-learning models. HyperDUM [61] applies HDC to uncertainty quantification in multimodal learning systems. Rather than estimating uncertainty only at the final output, HyperDUM projects latent features into hyperdimensional prototypes and estimates feature-level epistemic uncertainty from the similarity between a query hypervector and the selected class or prototype hypervector. Other studies use HDC for OOD detection in neural networks. Wilson et al. [62] and Daniels et al. [63] project internal feature-map values from multiple layers of a neural network into hypervector space and detect OOD inputs by measuring their distance from pretrained hypervector representations.

Overall, algorithm-level robustness work in HDC can be grouped into three directions. The first direction improves class separation through robust training, retraining, or metric learning. The second direction mitigates hardware faults through error-aware inference, masking, checksums, or redundancy. The third direction evaluates prediction trustworthiness through uncertainty estimation, abstention, or OOD detection. These methods show that HDC reliability can be improved beyond its inherent high-dimensional redundancy.

However, a gap remains between reliable HDC algorithms and hardware-efficient HDC implementations. Robust training and retraining methods can improve tolerance to noise, but they do not directly detect, localize, or mitigate hardware faults that occur after deployment. Checksum-based methods and redundancy improve fault coverage, but they require extra storage and control logic. CNN-assisted methods improve feature quality, but they increase model size and computation. Adversarial retraining, calibration-based abstention, and complex uncertainty-aware inference can increase training cost or runtime overhead. Margin-enhancement methods may also require additional optimization steps or nontrivial arithmetic. These costs are difficult to justify in tightly constrained edge deployments.

This thesis addresses the unresolved intersection between hardware-efficient HDC and lightweight runtime reliability. The proposed techniques are designed to detect or mitigate memory faults and assess whether the deployed model remains functional and trustworthy, while avoiding costly redundancy, neural feature extractors, and complex runtime algorithms. The central objective is to improve reliability within the native HDC inference flow, reusing existing similarity information and lightweight hardware structures wherever possible. This approach is especially important for edge systems, where reliability must be

improved under strict memory, power, and computation constraints rather than by adding large correction mechanisms.

2.4. Experimental Setup

This section summarizes the experimental methodology used throughout the thesis. Chapter-specific settings are reported in the corresponding evaluation sections only when they differ from the common setup described here.

FPGA-based evaluations are conducted on the AMD Xilinx PYNQ-Z2 platform. For larger hypervector dimensions, the AMD Xilinx ZCU104 platform is also used when the target design exceeds the resource capacity of the PYNQ-Z2 board. Hardware utilization, timing, and power are reported after synthesis and implementation using Vivado 2022.2. The reported hardware metrics include look-up tables (LUTs), flip-flops/registers (FFs), block RAMs (BRAMs), digital signal processing blocks (DSPs), maximum operating frequency, power consumption, throughput, and latency, when applicable.

Software-level evaluations use publicly available benchmark datasets for image classification, speech recognition, human activity recognition, biomedical sensing, and medical image classification. The datasets used in this thesis are summarized in Table 2.3. Depending on the experiment, the evaluated datasets include MNIST [64], Fashion-MNIST, CIFAR-10 [65], CIFAR-3, CIFAR-100 [66], ISOLET [67], UCI-HAR [68], EMG-Hand [69], Cardiotocography [70], MedMNIST [71], and MNIST-C [72]. CIFAR-3 refers to the first three classes of CIFAR-10. Unless otherwise specified, standard training and test splits are used. Dataset-specific preprocessing and chapter-specific dataset selections are described in the corresponding chapters only when they differ from standard benchmark usage.

Table 2.3 presents the application domain, data type, number of classes, and the number of training and test samples for the main benchmark datasets. For the Cardiotocography dataset, the original class distribution is highly imbalanced; therefore, resampling techniques [73] are applied to balance the classes before training and evaluation. For MedMNIST, PneumoniaMNIST is used as a binary classification task to distinguish between normal and pneumonia chest X-ray images.

For HDC model evaluation, the main configuration parameters are the hypervector dimension D, the quantization level or bit precision Q, the encoding method, the similarity metric, and the training method. The evaluated encodings include base-level encoding and random projection encoding. The training methods include one-shot HDC training and OnlineHD when iterative retraining is required. For binary or bipolar HDC models, Hamming distance-based similarity is used. For full-precision models, dot product and cosine similarity are considered where relevant.

For reliability experiments, transient faults are modeled by randomly flipping bits in the target HDC memory structures, primarily the CHV memory and, when necessary, the encoder memories. Unless otherwise specified, injected faults are distributed uniformly across the selected memory region. Because fault injection is stochastic, Monte Carlo simulations are used, and reported results are averaged over multiple independent runs with different random seeds. Chapter-specific fault probabilities, accuracy

Table 2.3.: Dataset descriptions

Name	Subject Area	Characteristics	#Classes	#Samples
MNIST [64]	Computer Vision	Image	10	60000 / 10000
Fashion-MNIST	Computer Vision	Image	10	60000 / 10000
CIFAR-10 [65]	Computer Vision	Image	10	50000 / 10000
CIFAR-3 [65]	Computer Vision	Image	3	chapter-specific
CIFAR-100 [66]	Computer Vision	Image	100	50000 / 10000
UCI-HAR [68]	Activity Sensing	Time-Series	6	7352 / 2947
ISOLET [67]	Audio Recognition	Time-Series	26	7797 / 1559
EMG-Hand [69]	Biomedical Sensing	Time-Series	8	21000 / 9000
Cardiotocography [70]	Biomedical Sensing	Multivariate	3	3699 / 532
PneumoniaMNIST [71]	Medical Imaging	Image	2	4708 / 624
MNIST-C [72]	Computer Vision	Corrupted image	10	chapter-specific

drop targets, numbers of runs, window sizes, and detection thresholds are provided in the corresponding experimental sections.

For out-of-distribution (OOD) and uncertainty detection experiments, thresholds are calibrated using in-distribution training or validation samples and then evaluated on unseen in-distribution and out-of-distribution samples. Unless otherwise specified, the reported detection results are obtained without retraining the HDC model or adding auxiliary learning-based detectors.

Unless explicitly stated otherwise, experiments in the contribution chapters use the proposed HDC-to-FPGA mapping framework introduced in chapter 3. Software training and model generation are performed using TorchHD [48], while hardware implementations are generated from configurable RTL modules. The generated hardware includes the HDC accelerator, the CHV memory, and, depending on the experiment, the corresponding reliability, fault-correction, OOD-detection, or uncertainty-detection module.

Part II.

Contributions

3. End-to-End HDC Framework

This chapter presents the first contribution of this thesis: an end-to-end HDC framework for efficient FPGA-based edge deployment. It addresses the encoder-memory bottleneck of conventional HDC implementations by introducing On-the-Fly Generation Encoder (OTFGEncoder), which generates the required encoding hypervectors on-the-fly instead of storing large ID-level and base-hypervector memories. The proposed framework links software-level HDC training to configurable FPGA implementation and is evaluated in terms of accuracy, resource utilization, power consumption, throughput, and comparison with neural network accelerators [1–3].

3.1. Motivation and Problem Definition

In many edge-intelligence applications, data must be processed where it is produced. Sending every measurement to a remote server is often not acceptable due to latency, energy, bandwidth, and privacy constraints. This is especially true for small IoT nodes, wearable devices, and embedded sensing platforms, where decisions must be made locally and within a limited power budget. However, these platforms provide only restricted memory, logic resources, and computational capability. As a result, deploying conventional neural-network models on such devices remains difficult, even when the models are compressed or quantized.

HDC is a promising alternative for this class of systems. Its operations are simple, highly parallel, and well matched to digital hardware. During inference, an input sample is encoded into a high-dimensional query hypervector, denoted as QHV. This QHV is then compared with the stored CHVs, and the class with the highest similarity is selected as the output. Since this process mainly relies on operations such as binding and bundling HDC can avoid the costly multiply-accumulate operations that dominate neural-network accelerators.

Despite these advantages, efficient HDC deployment on edge hardware remains limited by the encoding stage. The encoder maps input features into the high-dimensional space, and its implementation can dominate the memory footprint, logic utilization, and latency of the accelerator. Existing encoding methods address this problem only partially. Random projection requires projection parameters and matrix operations. Base-level encoding is simple and hardware friendly, but it stores both ID-level and Base Hypervector (BHV) memories. Permutation-based encoding reduces some memory requirements, but introduces permutation logic, such as barrel shifters, which can increase area, routing complexity, and

delay. As summarized in Table 2.2, these methods still impose a significant cost on small FPGA-based edge platforms.

The key observation is that HDC remains lightweight only if the encoding stage is also lightweight. Otherwise, the cost saved in the classifier is simply shifted to large encoding memories or complex datapaths. Therefore, this chapter addresses the encoder as a primary hardware bottleneck and aims to reduce its cost without sacrificing accuracy or the simplicity of HDC inference.

To address this issue, this chapter introduces the On-the-Fly Generation Encoder (OTFGEncoder). The main idea is to remove the stored ID-level and BHV matrices from the encoder and generate their corresponding hypervectors directly in hardware during inference. This approach replaces repeated memory access with deterministic, lightweight generation logic. As a result, the encoder becomes less dependent on on-chip memory and scales more efficiently with the hypervector dimension.

Building on OTFGEncoder, this chapter proposes an end-to-end framework, called E^3HDC, that connects software-level HDC training to FPGA hardware generation. The framework uses a configurable pipelined architecture with parameters that can be adjusted based on the available hardware budget. This allows trained HDC models to be translated into FPGA accelerators through a practical, reproducible, and hardware-aware process.

By solving the encoding-memory bottleneck, this chapter provides the hardware foundation for the rest of the thesis. It shows that HDC can be implemented as a complete edge accelerator, not only as an efficient classifier.

3.2. OTFGEncoder: Memory to Logic

OTFGEncoder applies a memory-to-logic transformation to the base-level encoding approach. This encoding scheme is selected because it relies on simple operations and is therefore well suited to efficient hardware implementation. In conventional base-level encoding, the value of each input feature indexes the ID-level matrix, while the position of the feature indexes the BHV matrix, as illustrated in Fig. 3.1. The two selected hypervectors are bound using XOR/XNOR operations, and the results for all features are bundled to form the QHV. The proposed method preserves this encoding behavior but replaces the ID-level and BHV memories with low-overhead generator circuits.

3.2.1. ID-level Generation

The ID-level matrix represents the possible values of an input feature in hyperdimensional space. In many HDC applications, the original input features are numerical values, such as sensor readings, pixel intensities, or extracted signal features. Before encoding, these continuous or discrete values are typically quantized into a finite number of levels. Each quantization level is then assigned a high-dimensional hypervector from the ID-level matrix.

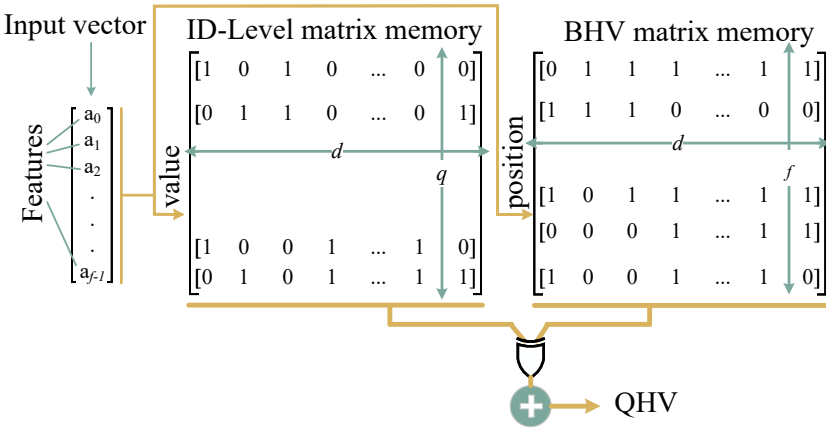


Figure 3.1.: Base-level encoding in HDC. Each input feature selects an ID-level hypervector according to its quantized value and a BHV according to its feature position. The selected hypervectors are bound and bundled to form the final QHV.

The most important property of the ID-level matrix is similarity preservation. Feature values that are close to each other should be mapped to similar hypervectors, while values that are far apart should be mapped to dissimilar hypervectors. This is important because many real-world classification tasks depend on the relative distance between feature values. For example, two nearby sensor readings should not produce completely unrelated representations. Instead, their corresponding hypervectors should have high similarity, allowing the HDC model to preserve the structure of the original input space after encoding.

Therefore, neighboring levels in the ID-level matrix are designed to differ in only a limited number of dimensions, while more distant levels differ in more dimensions. As a result, the ID-level matrix enables a gradual transition from one value level to another, allowing the encoder to capture the magnitude of each input feature and preserve meaningful relationships between values.

The standard size of the ID-level matrix is $N \times D$, where N is the number of quantization levels and D is the hypervector dimension. However, storing this matrix can introduce a significant memory overhead, especially when either the number of quantization levels or the hypervector dimension increases.

To avoid storing the ID-level matrix, the proposed *ID-Gen* module generates level hypervectors at runtime. For an input feature value x, where $x \in [0, N)$, the step size S is computed as

$$S = \left\lfloor \frac{D}{N} \right\rfloor, \tag{3.1}$$

and the remaining dimensions are given by

$$R = D - (S \times N). \tag{3.2}$$

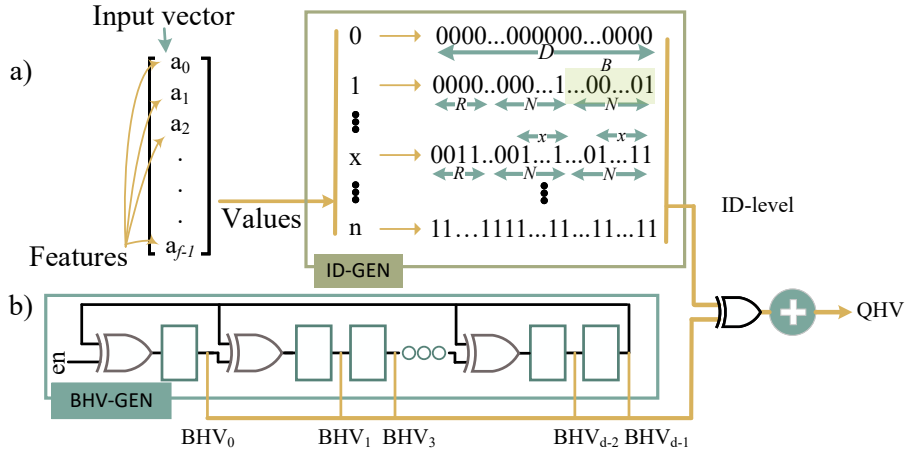


Figure 3.2.: Memory-to-logic transformation used by the proposed OTFGEncoder. Instead of retrieving ID-level and BHV hypervectors from stored matrices, the required encoding parameters are generated at runtime using lightweight deterministic logic.

As depicted in the upper section of Figure 3.2, to generate the ID-level hypervector for a feature value x, a binary sequence B is first formed in the N-level quantization space. This sequence contains x ones followed by $N - x$ zeros:

$$B = \underbrace{00\ldots0}_{N-x}\underbrace{11\ldots1}_{x}. \tag{3.3}$$

B is then repeated S times, and the repeated bit groups are concatenated to form the main part of the hypervector. Since D is not necessarily divisible by N, the remaining R dimensions are filled by concatenating the first R bits of B to the generated sequence. Therefore, the final ID-level hypervector ID_x can be expressed as

$$ID_x = \underbrace{B(R:0]}_{R}\underbrace{BB\ldots B}_{S\times N} \tag{3.4}$$

where $B(R:0]$ denotes the first R bits of B.

In this construction, the lowest feature value is represented by a sequence dominated by zeros, and the number of ones increases as the feature value increases. Consequently, nearby scalar values produce similar hypervectors, while distant scalar values produce more dissimilar hypervectors. This preserves the similarity-preserving behavior needed for HDC classification without storing the ID-level matrix.

3.2.2. BHV Generation

The base-level matrix represents the identity or position of each input feature in the hyperdimensional space. While the ID-level matrix encodes the value of a feature, the base-level matrix differentiates one feature from another. Each row of the base-level matrix corresponds to a specific feature index in the input

vector. For example, in an input vector with F features, the base-level matrix contains F feature-position hypervectors of size D.

The most important property of the base-level matrix is near-orthogonality. Hypervectors assigned to different feature positions should be nearly orthogonal, meaning that they should have very low similarity to each other. This property ensures that different input features remain distinguishable after encoding. Without this separation, values from different features could interfere with each other, making it difficult for the classifier to learn meaningful class representations.

In binary HDC, near-orthogonality usually means that two BHVs have approximately half of their bits different. This makes them statistically independent in the high-dimensional space. Because of this independence, binding a value hypervector with different BHVs produces different representations, even if the input values are the same. For example, if two input features have the same quantized value, they will select the same ID-level hypervector. However, after binding with two different base-level hypervectors, the resulting feature representations will still be distinct because each feature has its own identity hypervector.

During encoding, the selected BHVs are bound with the corresponding ID-level hypervector, usually using XOR or XNOR operations. The bound hypervectors from all features are then bundled together to generate the final query hypervector. In this way, the base-level matrix prevents ambiguity between features and allows the HDC model to combine value information and feature-position information in a single high-dimensional representation.

The proposed *BHV-Gen*, shown in the lower section of Figure 3.2, replaces the BHV memory with a cyclic digital circuit based on Linear Feedback Shift Register (LFSR) and Multiple Input Signature Register (MISR) principles [74]. A conventional LFSR is a sequential circuit that shifts its internal state and generates a single output bit at each clock cycle, while its new input bit is produced by a feedback function of selected register values. This structure allows the circuit to generate long pseudo-random bit sequences from configurable seeds. The feedback behavior of an LFSR can be described using a characteristic polynomial, for example:

$$p(x) = x^D + x^{D-3} + \cdots + x^0. \tag{3.5}$$

A MISR-style structure extends this principle by accepting parallel inputs and maintaining a parallel state representation. The proposed *BHV-Gen* combines these properties by using a chain of D registers, whose current state forms one row of the BHV matrix. Through shifting and feedback, the generator produces repeatable pseudo-random rows with low hardware overhead. A comparison between the LFSR, MISR, and the proposed *BHV-Gen* is shown in Figure 3.3.

In this way, each row of the BHV memory is generated in one clock cycle instead of being read from a stored memory array. This does not reduce the overall throughput compared with a conventional memory-based design, because the complete BHV matrix cannot be accessed from memory in a single

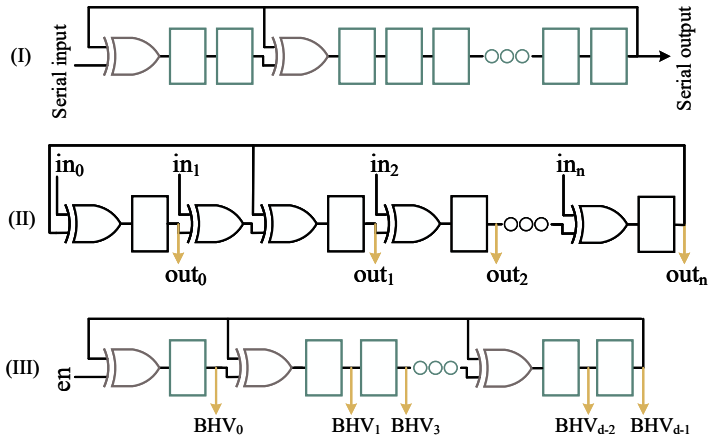


Figure 3.3.: Comparison of conventional LFSR, MISR, and the proposed BHV-Gen structure. BHV-Gen combines feedback-based pseudo-random generation with parallel state outputs, allowing BHV rows to be generated on-the-fly without storing the entire matrix.

clock cycle either. Therefore, the generator-based approach preserves the sequential row-access behavior required by the encoder while eliminating the need to store the full BHV matrix.

If higher parallelism is required, multiple *BHV-Gen* modules can be instantiated in parallel using the same generator structure with different seeds. This enables several BHV rows to be generated simultaneously, depending on the throughput and resource requirements of the target design. Empirical exploration of generator configurations confirms that the LFSR-like construction provides suitable orthogonality for HDC encoding while avoiding the power and area costs of storing and accessing high-dimensional BHVs.

3.3. Generic HDC Architecture

To evaluate our proposed encoding scheme, we also introduce a generic HDC inference hardware framework for FPGAs. This framework allows the deployment of different encoding schemes onto the FPGA to easily evaluate the impact of the proposed Energy Efficient Encoding for Hyperdimensional Computing (E^3HDC) on resource utilization, power, and maximum frequency. The overview of the HDC hardware architecture is shown in Figure 3.4, which consists of the typical two main stages: encoding and classification. Our main objectives for the hardware architecture are reconfigurability and low resource usage, without sacrificing throughput. This is achieved through highly flexible VHDL constructs and an elaborate pipeline design as illustrated in Figure 3.4-b. The following subsections provide a thorough explanation of each stage and their pipeline structure.

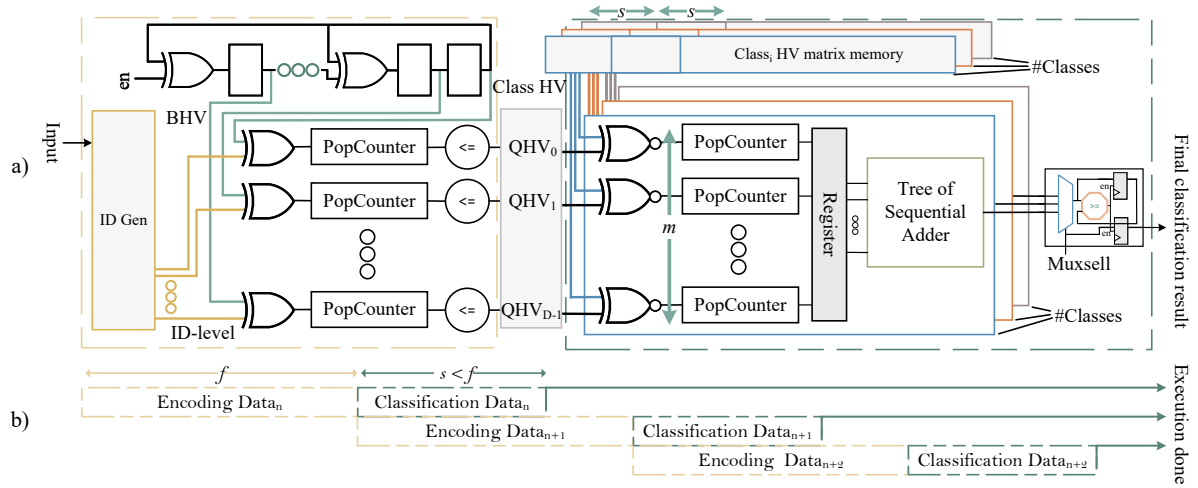


Figure 3.4.: Overview of the proposed E^3HDC accelerator architecture and execution pipeline. (a) The accelerator consists of an on-the-fly encoding stage, CHV memory, parallel similarity computation units, and final classification logic. The ID-level and BHV generators replace stored encoding matrices and provide the hypervectors required for QHV generation. (b) The two-stage pipeline overlaps encoding and classification so that, after one input sample is encoded, its QHV is classified while the next input sample is processed by the encoder.

3.3.1. Encoding

During encoding, the dimension of the input is transformed into hyperspace, which is done feature by feature in this architecture. During the *bundling* part of encoding, we generate an HV for each input feature per clock cycle, which in this case of E^3HDC is done with the constructs described in section 3.2.

In the *bundling* phase, the feature-dependent HVs are then fed directly into an array of sequential PopCount modules. After iterating over all features in the input, the results of the PopCounts are binarized and stored in the registers as the encoded QHV for classification.

Thus, the encoding in our architecture requires F clock cycles, where F is the number of features in the input vector. Note that the dimension size impacts resource utilization (as shown in subsection 3.3.3), but not latency or throughput.

3.3.2. Classification

The classification stage consists of two parts, computing the similarities between QHV and CHVs, and comparing the similarity values. The generated QHV from the encoding stage is now compared with all pre-stored CHVs which is done with an array of XNORs and PopCounts for all classes simultaneously. The results of the similarity calculations between QHV and CHVs are stored in registers and compared one by one sequentially.

To improve the efficiency of our design, we design a two-stage pipeline, which means the classification stage must be synchronized with the previous stage. The following subsection discusses the design of this pipeline and the timing of the operations of the two stages.

3.3.3. Pipeline

The proposed two-stage pipeline shown in Figure 3.4-b is adjusted based on the number of features (F). In the encoding stage, the number of XNOR and sequential PopCount modules in the encoding part is equal to the dimension size or D and they need to iterate feature size (F) times to cover all features of the input. The rest of the pipeline is adjusted based on these two parameters as well. When all input features are fully processed after F clock cycles, the QHV result is stored in a register file after encoding to be processed by classification, and at the same time, the encoding module starts processing the next input data in the queue.

To form a pipeline structure, the classification stage must operate in less than/equal to F clock cycles to iterate over all elements of the current QHV before the next input is generated. Since the size of the QHV is usually much larger than the size of the features ($F \ll D$), the QHV must be divided into equal segments of size indicated by E, where ($E \leq F$). Since all segments must operate simultaneously, it is necessary to have multiple (M) XNORs and PopCounts for each class. A central controller controls all operations for all classes and allows simultaneous access to different units of CHV with corresponding pointers, as well as for the memory of CHV, and normally the last segment of CHV requires zero padding. Therefore, to select the appropriate value for E, we must consider that E can be chosen between any power of two numbers because of the memory segments and should be less than F, with a tradeoff between the number of XNORs and PopCounts of each class (M) and the zero padding of the memory for the last segment of the class memory QHV. In other words:

$$E = 2^n, E \leq F, D \leq M \times E. \tag{3.6}$$

After counting the similarity of the segments independently, they are joined together using a tree of sequential adders that are also pipelined and operate at the same level as the similarity count. Since the number of classes is always smaller than the number of features (F), the similarity count can be performed with a single sequential comparison.

3.3.4. Hardware Optimization

To improve the efficiency of the processing hardware in HDC, we investigate the possibility of eliminating the calculations based on the classification nature in HDC. In HDC, the classification results do not depend on exact similarity scores, but on the relative similarity of classes within the CHV by detecting the most similar one. By considering the importance of relative scores over absolute scores in the classification results, it becomes notable that certain elements within all CHVs have identical effects on all categorization results.

For this purpose, an analysis of all CHVs is performed after training and quantization, and all their comparable elements are retrieved based on their respective positions. Subsequently, the *XOR+PopCount* operations associated with these elements are pruned from the encoding part of the architecture. As a

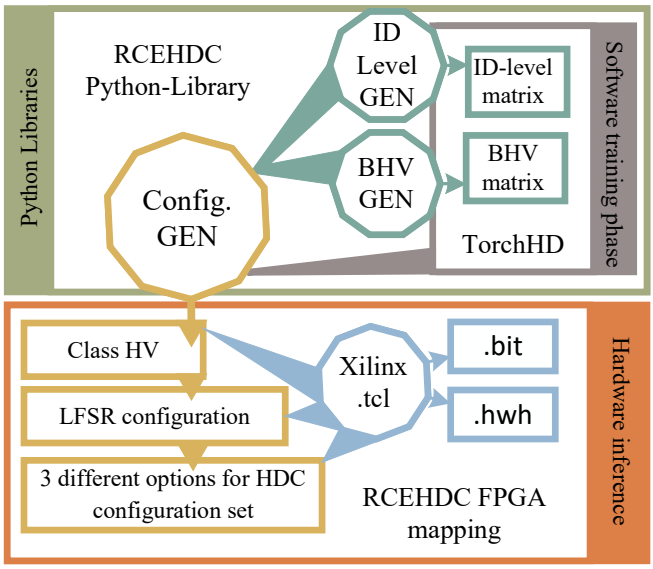


Figure 3.5.: End-to-end software-hardware mapping flow of the proposed HDC framework. During the software training phase, the ID-level and BHV representations are generated inside the Python library built on TorchHD. Based on the selected implementation setup, the configuration generator exports the required CHV memory, LFSR settings, and HDC configuration files to the hardware flow. These inputs are then used by the Xilinx toolchain to generate the FPGA implementation outputs, including the bitstream and hardware handoff files.

result, they are no longer included in the generated QHV. At the same time, all these components are also pruned from CHVs, resulting in the entire hardware functioning in a reduced dimension space. In summary, we can identify the most important dimensions and perform computations only on them to reduce the hardware, with no impact on the classification result and thus, ensure an efficient hardware utilization.

3.4. End-to-end Software-Hardware Platform

An overview of the open-source end-to-end software-hardware flow is presented in Figure 3.5. This flow is implemented for the Resource-Constrained Edge Hyperdimensional Computing (RCEHDC) framework. The goal of the framework is to provide a configurable path from software-level HDC training to hardware-level FPGA implementation, especially for resource-constrained edge and IoT devices.

The framework uses TorchHD [48] for software-level HDC training. After training, the CHVs are extracted from the trained model and passed to the hardware-generation stage. Depending on the selected operating mode, the framework either stores the encoder parameters in memory or generates the required parameters for the on-the-fly encoder. If pruning is enabled, the framework applies dimension-pruning analysis before generating the final hardware files. The remaining dimensions are then segmented according to the selected segment size E, and the required ROM initialization files are generated.

The current version of the platform supports three main operating modes. The first mode is the conventional base-level HDC implementation. In this mode, the encoder parameters are stored explicitly

in memory. More specifically, the ID-level matrix and the BHV matrix are generated during the software stage and then mapped to hardware memories. During inference, the encoder accesses these memories to retrieve the required hypervectors. This mode follows the standard base-level encoding approach and provides a reference implementation for comparison with the optimized designs.

The second mode supports the proposed *OTFGEncoder*. In this mode, the encoder parameters are not stored as full matrices. Instead, the ID-level hypervectors and BHV hypervectors are generated on-the-fly using lightweight hardware generators. This replaces the memory-based encoder with logic-based generation modules and significantly reduces the memory footprint of the accelerator. As a result, the design becomes more suitable for small FPGA devices where on-chip memory is limited.

The third mode adds dimension pruning to the generated hardware. This optimization can be applied to both the conventional memory-based encoder and the proposed *OTFGEncoder*. After training, the framework analyzes the generated CHVs and identifies dimensions that do not affect the final classification decision. These ineffective dimensions are removed from the hardware implementation. Therefore, the pruning stage reduces the size of the stored CHVs, the number of comparison operations, and the overall hardware cost, while preserving the classification result.

All hardware modules are designed to be configurable using VHDL *recursive* and *for-generate* constructs. Therefore, the same hardware description can be automatically specialized for different numbers of features, hypervector dimension sizes, and number of classes. The top-level configuration is generated from the feature size F, hypervector dimension D, number of classes, and segmentation parameter E. This removes the need for manual modification of the hardware code when moving between datasets or FPGA platforms. Finally, the generated VHDL files and memory initialization files are passed to the FPGA vendor toolflow to synthesize the final accelerator and generate the bitstream.

3.5. Evaluation

In this section, we evaluate our proposed encoding technique and hardware design and compare it with other state-of-the-art works in terms of accuracy and hardware consumption.

Table 3.1.: Impact on accuracy for different datasets.

Data set	E^3HDC	***TorchHD***	***HDCC [75]***
Dimension Size	1000	1000	10240
MNIST	82.98%	82.66%	82.8%
ISOLET	85.18%	85.05%	84.8%
EMG	97.43%	98.88%	99.3%
GraphHD	89.47%	89.47%	-

Table 3.2.: Comparison of different encoding techniques with the proposed approach without additional hardware optimization.

Platform	Dimension	Encoding method	LUT	FF	BRAM	Power	$Freq_{max}$
PYNQ-Z2	1000	base-level [16]	10840 (20.37%)	20760 (19.51%)	56 (40.00%)	0.377 W	120 MHz
		Permutation [14]	20014 (37.62%)	20729 (19.48%)	28 (20.00%)	0.672 W	70 MHz
		E^3HDC	**12883 (24.21%)**	**21759 (20.45%)**	**0.00%**	**0.259 W**	**120 MHz**
		Prior work [76]	33010 (67.20%)	33224 (31.23%)	111.5 (79.64%)	1.384 W	100 MHz
	2000	base-level [16]	21595 (40.59%)	41144 (38.66%)	112 (80.00%)	0.644 W	100 MHz
		Permutation [14]	40251 (75.65%)	41122 (38.64%)	56 (40.00%)	1.255 W	50 MHz
		E^3HDC	**26640 (50.07%)**	**43169 (40.57%)**	**0.00%**	**0.398 W**	**100 MHz**
	3000	E^3HDC	**44314 (83.29%)**	**64701 (60.80%)**	**0.00%**	**0.573 W**	**85 MHz**
Zynq ZCU104	5000	E^3HDC	**68257 (29.63%)**	**158443 (34.38%)**	**0.00%**	**1.474 W**	**100 MHz**

Table 3.3.: Comparison of FINN binary neural-network accelerators with the sparse E^3HDC implementation on PYNQ.

Dataset	ML model	Accuracy	LUT	FF	BRAM	Power	Throughput	Energy/data
ISOLET	E^3HDC **(30.6%),** D = 1000	**82.65%**	**10714 (20.13%)**	**18638 (17.51%)**	**0.00%**	**0.308 W**	**162K**	**1.90 μJ**
	1D-CNN* (FINN)	84.35%	12230 (22.99%)	13023 (12.24%)	6.43%	1.527 W	16.5K	92.55 μJ
	SFC-max** (FINN)	89.99%	27684 (52.04%)	25756 (24.21%)	99.29%	2.704 W	609K	4.435 μJ
	SFC-fix** (FINN)	89.99%	13535 (25.44%)	18275 (17.18%)	25.17%	1.762 W	169K	10.396 μJ
MNIST	E^3HDC **(64.8%),** D = 1000	**82.83%**	**5502 (10.34%)**	**12590 (11.83%)**	**0.00%**	**0.215 W**	**127.5K**	**1.68 μJ**
	2D-CNN* (FINN)	82.80%	9473 (17.81%)	11278 (10.60%)	1.79%	1.484 W	2990	0.49 mJ
	MLP-16-max** (FINN)	82.60%	22912 (43.07%)	24476 (23.00%)	1.07%	1.523 W	4.76M	0.32 μJ
	MLP-16-fix** (FINN)	82.60%	8149 (15.32%)	10641 (10.00%)	1.79%	1.486 W	125K	11.888 μJ
	SFC-max** (FINN)	97.87%	25223 (47.41%)	21657 (20.35%)	92.14%	2.648 W	645K	4.10 μJ
	SFC-fix** (FINN)	97.87%	11600 (21.80%)	12952 (12.17%)	23.93%	1.680 W	125K	13.44 μJ

* CNNs have three convolutional layers followed by a fully connected layer and the output layer. ** SFC and MLP-16 are three-layer MLPs with 256 and 16 neurons in each layer, respectively.

3.5.1. Impact on Accuracy

To evaluate the impact of on-the-fly encoding on classification quality, the OTFGEncoder was integrated into TorchHD and compared with baseline TorchHD and HDC models. Table 3.1 shows that the proposed memoryless encoding maintains comparable accuracy across multiple datasets. In several cases, it matches or slightly improves the baseline while using a much smaller dimension than some prior implementations. The results indicate that the ID-level and BHV memories can be replaced by generator logic with negligible impact on accuracy. This supports the central claim that HDC encoding parameters do not need to be stored explicitly when they can be regenerated deterministically with suitable similarity and orthogonality properties.

3.5.2. Hardware Resource Requirements

We implement the MNIST [64] dataset with various dimension sizes on the Xilinx PYNQ-Z2 FPGA board, a relatively small FPGA-SoC platform. The number of levels in ID-level is set to 1000, and the

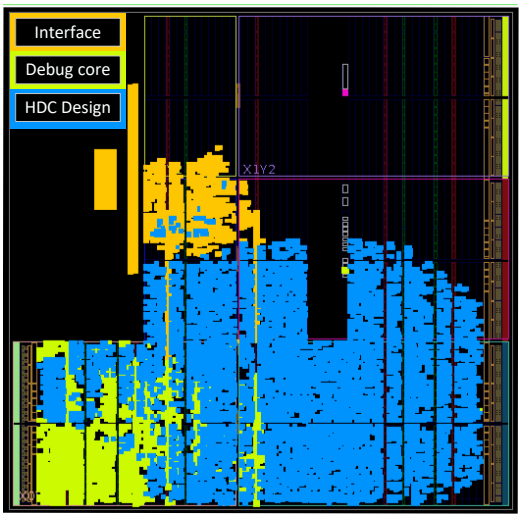


Figure 3.6.: FPGA floorplan of the proposed HDC accelerator on the PYNQ-Z2 platform, showing the placement of the HDC design, interface logic, and debug infrastructure after implementation.

number of features is 784 (28×28). For comparison, we implement other approaches using base-level encoding (as in [16]) and permutation (as in [14]) with the same hardware architecture described in section 3.3. Unlike other state-of-the-art implementations [16, 42], where execution time increases with dimension size, our hardware architecture operates independently of dimension size, and execution time depends only on the feature size f, making the architecture suitable and scalable for large dimension sizes. For example, for the MNIST dataset, where each input has 784 features, each output is generated in 785 clock cycles. The floorplan of the HDC implementation on the PYNQ-Z2 FPGA is shown in Figure 3.6.

In Table 3.2, the scalability of E^3HDC and the effectiveness of our encoding are compared to the base-level and permutation encoding approaches. No pruning is used in any of the implementations. As shown in Table 3.2, the BRAM utilization of our encoding technique is 0.0%, which makes the standard base-level encoding approach inapplicable for larger dimensions. Although the memory consumption of the permutation approach is half that of the standard base-level encoding, it uses many LUTs due to large barrel shifter operations. This significantly affects both frequency and power consumption, making it impossible to increase the dimension size on the PYNQ-Z2 FPGA board.

Our implementation can be deployed with twice the HV dimension of the others on the same FPGA and consumes less power. The base-level approach requires memory and memory access, while the permutation approach requires a large, power-hungry barrel shifter compared to the XOR and PopCount operations used in our method. This result demonstrates the scalability of our design compared to other techniques.

To further illustrate the scalability of the design, we implement higher dimension sizes on the Zynq UltraScale ZCU104 FPGA. The results demonstrate the efficiency and scalability of the proposed architecture, even though this FPGA has higher static power, which leads to increased power consumption.

Moreover, as shown in the last column of Table 3.2, replacing the memories used for storing BHV and ID-level matrices with our proposed structures does not reduce the maximum frequency. In contrast, the complex barrel shifters required for permutation [14] significantly affect the maximum frequency and, consequently, the throughput.

3.5.3. Comparison with Neural Network FPGA Accelerators

Our goal in this work is to propose an efficient accelerator for HDC edge devices. For example, E^3HDC uses 68% fewer LUTs, 7% fewer BRAMs (no BRAM in E^3HDC), and is 2.17× faster than the binary accelerator proposed in QuantHD [77]. To provide a comprehensive comparison, we also evaluate our E^3HDC accelerator against Convolutional Neural Network (CNN) and Multilayer Perceptron (MLP) accelerators mapped to FPGAs. The accelerators are compiled with FINN [78], a publicly available framework for compiling quantized NN accelerators from PyTorch models. We adjust the model architectures to achieve accuracy similar to the HDC models, as shown in Table 3.3. Both CNNs have 3 convolutional layers, followed by a fully connected layer and an output layer. For ISOLET, we use 1D convolution, while for MNIST, we use 2D convolution. Both CNN models are binary, with 1-bit weights and activations. However, for ISOLET, 8-bit inputs are required to achieve the desired accuracy, while for MNIST, inputs are also quantized to 1 bit. In addition to binary CNN, we train binary MLP models, SFC as defined in [78], and MLP-16 for the MNIST and ISOLET datasets as two samples. SFC and MLP-16 are 3-layer MLP with 256 and 16 neurons in each layer, respectively. MLP-16 is a variant of SFC with accuracy reduced to that of our accelerator. Here, SFC-max denotes an accelerator with a configuration, so that the resource utilization is maximized, while SFC-fix is configured to have throughput similar to our accelerator.

We prune the non-effective features in HDC models, which, for MNIST and ISOLET [67], are 64.8% and 30.6% of elements in the dimensions, respectively. To provide a comprehensive comparison, we consider hardware utilization, power, throughput (number of inferences per second), and energy per inference for each model on the PYNQ-Z2 FPGA running at 100 MHz. While the number of Flip-Flops (FFs) is higher in our proposed E^3HDC due to the memory required for CHV, E^3HDC has lower resource utilization in terms of LUT and BRAM compared to the CNN models. Most importantly, the power consumption for HDC is about 5–7× lower, while the throughput is up to 9–42× higher, resulting in a significant reduction in energy per input inference (48× to 291.6× lower). When compared to MLPs, only the MLP-16-max achieves better power consumption, with a remarkable throughput of around 4.76 million inferences per second. However, this comes at the cost of approximately 4× higher LUT and 2× higher FF utilization. For the SFC-fix and MLP-16-fix, we observe much higher power consumption and increased resource utilization, especially in BRAM for the SFC-fix, which requires 23.93% BRAM. Thus, we conclude that the lower power consumption and high throughput make E^3HDC much more suitable for edge devices than a binary CNN or MLP with similar accuracy.

3.6. Conclusion

We propose a hardware-aware encoding technique for implementing HDC as a low-power design, replacing memory storage and generating parameters on-the-fly to minimize both memory usage and power consumption, thereby making the design more scalable. Furthermore, we eliminate ineffective computation and hardware inference by pruning unnecessary features in the QHVs and CHVs without sacrificing accuracy. This approach offers a significant improvement over current state-of-the-art implementations, with negligible impact on classification accuracy. Additionally, we present an automated end-to-end workflow, from training the HDC model in TorchHD to mapping the desired model onto an FPGA. In future work, we aim to further improve accuracy by selecting optimized configuration patterns for the LFSR strategy.

4. Non-Uniform Error Correction for Reliable HDC

This chapter presents a reliability-aware error-correction method for HDC accelerators. It focuses on faults in CHV memory, which directly affect similarity computation during inference. Instead of uniformly protecting the entire memory, the proposed method analyzes the fault sensitivity of different memory regions and applies stronger protection only where it is most needed. This non-uniform strategy improves fault resilience while reducing the memory and hardware overhead of conventional ECC protection [4].

4.1. Motivation and Problem Definition

The previous chapter established that Hyperdimensional Computing (HDC) can be efficiently mapped to edge-oriented FPGA platforms by reducing encoder memory and using lightweight hardware structures. This confirms the suitability of HDC for resource-constrained inference. However, efficient implementation alone is not sufficient for practical deployment. Once an accelerator is small, fast, and energy efficient, the next question is whether it can remain dependable under realistic hardware faults.

This reliability requirement is especially important because edge devices are increasingly used in environments where incorrect decisions can have direct consequences, including wearable healthcare, biomedical monitoring, autonomous sensing, and human-activity recognition [79]. In these applications, the objective is not only to perform inference with low power and low latency, but also to maintain correct behavior when the underlying hardware is affected by transient or permanent faults.

Although HDC is naturally tolerant to random perturbations, practical hardware implementations remain vulnerable to memory faults. The previous chapter reduced this vulnerability on the encoder side by eliminating stored encoding matrices and generating the required hypervectors on-the-fly. However, the learned model still resides in the CHV memory, which is accessed directly during associative search. Faults in this memory are particularly critical because every inference compares the QHV with the stored class representations. Therefore, bit flips in CHV memory can alter similarity scores, disrupt the ranking among classes, and potentially change the final prediction [51]. This makes CHV memory the main remaining memory-related reliability bottleneck.

This challenge becomes more severe with technology scaling and voltage reduction. These techniques are widely used to reduce energy consumption, but they also decrease noise margins and increase the probability of soft errors, timing violations, and memory bit flips [80]. Consequently, the same design pressures that motivate lightweight edge acceleration also increase the need for efficient protection mechanisms.

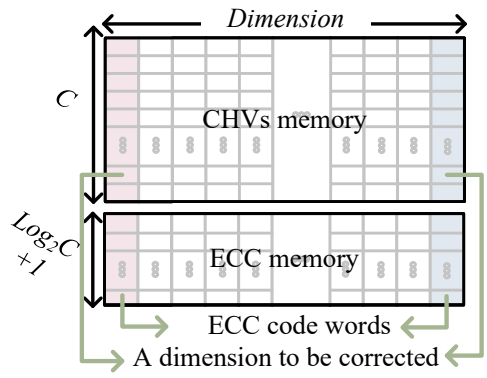


Figure 4.1.: Column-wise CHV and ECC memory layout for reliable HDC classification. Each memory column represents one hypervector dimension across all classes, with the corresponding ECC codeword stored separately for runtime correction.

To integrate ECC into the CHV accelerator, the classification module must access the CHV memory column-wise, as illustrated in Figure 4.1. This memory-access pattern is consistent with the proposed architecture described in chapter 3.

A direct solution is to protect the entire CHV memory using conventional Error-Correcting Code (ECC). However, uniform protection treats all memory locations as equally important, introducing unnecessary storage and decoding overhead. This approach is poorly aligned with the design goal of edge HDC, where reliability must be improved without compromising the low-memory and low-power advantages provided by the accelerator.

The key observation in this chapter is that the CHV memory has non-uniform fault sensitivity. Faults in some regions strongly affect the similarity computation and can change the predicted class, while faults in other regions have little influence on the final decision. This means that uniform protection wastes resources by treating all memory regions as equally critical. A more efficient approach is to allocate error-protection resources based on the actual impact of each region on classification accuracy.

Based on this behavior, the proposed method groups CHV columns by their introduced sensitivity metric (Hamming weight) and uses this grouping as a lightweight indicator of fault sensitivity. Sensitive column groups are protected with stronger ECC, while less sensitive groups use lower-cost protection or fault management. This avoids the cost of uniform full-memory protection while preserving the classification accuracy under memory faults.

This selective strategy offers a practical balance between reliability and hardware efficiency. It enhances the resilience of HDC inference while reducing the storage and hardware overhead associated with conventional uniform ECC. Thus, the chapter extends the previously developed efficient hardware implementation with reliability awareness, while preserving the lightweight design principles required for edge deployment.

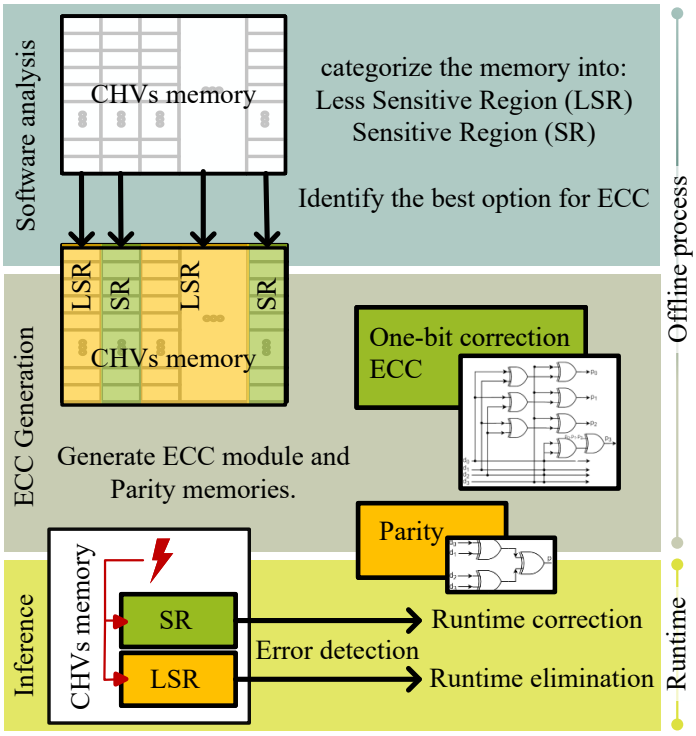


Figure 4.2.: Overview of the proposed non-uniform ECC framework. Offline sensitivity analysis partitions CHV memory into sensitive and less-sensitive regions, and the generated runtime hardware applies region-dependent protection during classification.

4.2. Proposed Strategy

We propose an efficient non-uniform ECC framework optimized for binary/bipolar HDC models, as shown in Figure 4.2. The core idea is to classify memory regions into Sensitive Regions (SR) and Less Sensitive Regions (LSR) based on their susceptibility to faults. The approach selectively applies error correction mechanisms to memory regions of CHVs based on their sensitivity to faults. We significantly increase fault resilience by employing a non-uniform error correction strategy while minimizing hardware and storage overhead. The framework comprises three primary elements: a detailed offline analysis of fault sensitivity, ECC encoding process, and selective runtime integration of ECC within the HDC environment.

4.2.1. Offline Fault Sensitivity Analysis

The basic organization of the CHV memory can be modeled as a $C \times D$ memory array, where C represents the number of classes, and D denotes the size of each HV in CHV memory or the dimensionality of the hyperdimensional space, where $D \gg C$. To validate our method, fault sensitivity analysis is performed through extensive evaluation by Monte Carlo simulations for fault injection into the CHV memory. Faults are randomly sampled from a uniform distribution prior to inference. We conducted 20 Monte Carlo

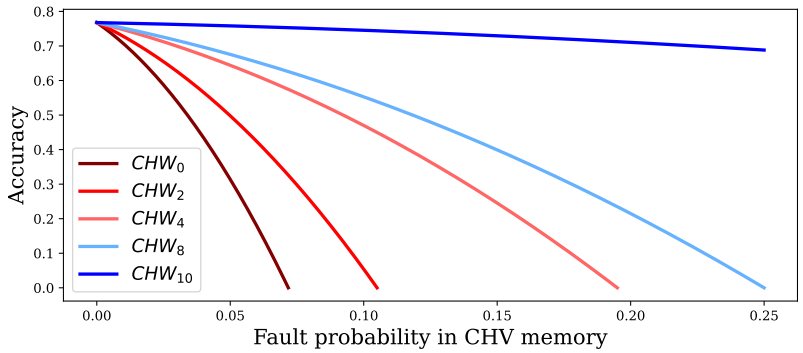


Figure 4.3.: Fault sensitivity across different regions, distinguished by Column Hamming weight (CHw). The results show that fault sensitivity decreases as the Hamming weight increases. In this example, the blue lines (CHw_{10} and CHw_8) represent LSR, while the red lines are located in SR.

simulations for different fault probabilities and reported the mean accuracy to evaluate the impact of these faults.

The impact of each column on the final decision is $1/D$. Therefore, due to the high dimensionality of D, injecting faults into a single column (i.e., all elements of a one-dimensional index) has almost no effect on accuracy. Therefore, the memory regions were expanded from individual memory columns to groups of memory columns defined by the columns' Hamming weight as follows:

$$CHw_k = \{j \mid \sum_{i=1}^{C} CHV(i,j) = k, \quad \text{for } j \in \{1,2,\ldots,D\}\}.$$

Here, CHw_k represents the set of columns with Hamming weight sums equal to k.

We evaluated the fault sensitivity of CHw_K across various dimension sizes, encoding schemes, and datasets. As an illustrative example, Figure 4.3 presents the accuracy degradation ratio for various CHw categories using the MNIST benchmark dataset based on the probability of faults ($Fault_probability = \#Faults/\#bits_in_CHV$). The results confirm that the fault sensitivity in CHV memory correlates strongly with the Hamming weights of the columns in the CHV memory array: Higher Hamming weights lead to a lower impact on accuracy degradation, indicating that these regions are less sensitive to errors.

Based on this observation, we hypothesize that Hamming weight is an effective metric for evaluating fault sensitivity in HDC and its impact on inference accuracy. Based on an experimental analysis on various datasets and across various encoding schemes and dimension sizes, we show that using Hamming weight as an alternative to traditional Monte Carlo simulations is a computationally efficient approach that significantly reduces the time required for fault analysis. This finding highlights the potential of Hamming weight as an effective metric for fault sensitivity analysis without performing time-consuming fault simulations when changing parameters such as encoding scheme, dimension size, or dataset. Based on sensitivity classification, the framework assigns regions to two levels of error correction/detection: Sensitive Regions (SR) and Less Sensitive Regions (LSR).

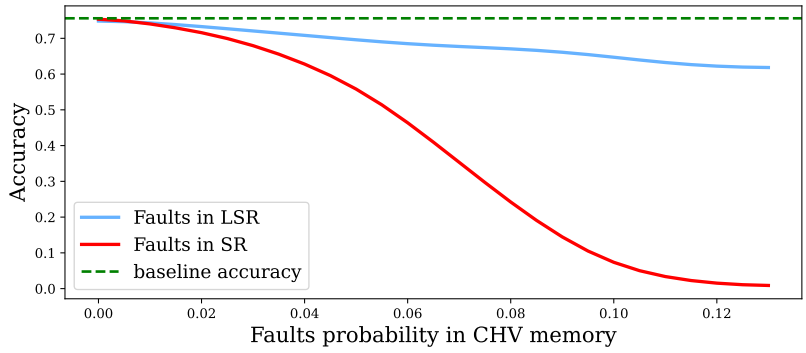


Figure 4.4.: Impact of CHV memory faults on classification accuracy when faults are injected into sensitive (SR) and less-sensitive regions (LSR). The comparison motivates selective protection instead of uniform correction across the full memory.

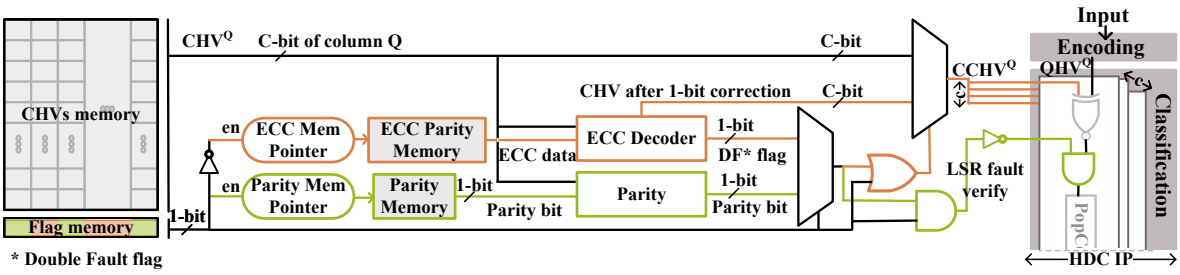


Figure 4.5.: Architecture and FPGA implementation of the proposed non-uniform ECC scheme. Sensitive regions use correction, while less-sensitive regions use lightweight detection and elimination to reduce overhead without adding latency.

Sensitive Regions (SR) are critical to inference accuracy, and we protect this memory region with SEC-DED ECC. Less Sensitive Regions (LSR) exhibit reduced sensitivity to faults, and for this region we rely on HDC robustness and a lightweight parity bit as fault detection mechanisms are employed. In cases of odd fault detection, the entire faulty column is excluded from calculations. Figure 4.4 shows the effect of injecting faults into the SR and LSR columns of CHV memory. As shown, the effect of faults on accuracy for LSR is much lower than for the SR region.

4.2.2. ECC Encoding Process

Each CHV contains a large number of dimensions, so applying a single ECC to an entire CHV is not an efficient approach to enhancing the HDC's robustness. In our proposed strategy, ECC is applied column-wise using the standard SEC-DED Hamming code, where the input data ($I_{1\times C}$) to the ECC module would be the column of $C \times D$ the array, i.e., the D^{th} element of each class in the memory. A C-bit column data ($[I]_{1\times C}$) can be encoded into an n-bit codeword ($[W]_{1\times n}$) by appending $r = n - C$ redundant bits using $[W]_{1\times n} = [I]_{1\times C} \times [G]_{C\times n}$, where ($[G]_{C\times n}$) denotes the generator matrix of the ECC. Similarly, for LSR, a single parity bit can be obtained by XOR operation among column bits. There is no need to have dedicated hardware logic for the encoding part, as it can be done offline via software-based encoding, and then the encoded CHVs can be directly mapped to the HDC accelerators.

4.2.3. Runtime Error Correction in HDC

The ECC module (shown in Figure 4.5) is used for each memory column read module to ensure error detection and correction. To efficiently distinguish between LSR and SR within the CHV memory, a flag is assigned to each column. These flags are stored in a flag memory that is synchronized with the CHV memory. By integrating this flag-based mechanism and dynamically selecting the activation of the correction modules, resource usage is optimized without compromising accuracy. The flag indicates whether to apply ECC decoding for that column (SR) or just a single parity bit for an odd number of fault detections (LSR). The activation path in Figure 4.5 is indicated by an orange for SR and a green for LSR.

For LSR, a lightweight and simple parity check module is used to detect an odd number of faults in a memory column. If an odd number of faults is detected, the column is marked as faulty and excluded from subsequent calculations. The procedure for implementing this process is as follows: The parity check module outputs the value ‘1‘ if an odd number of faults are detected in the column. The parity output is selected by a flag that is ’1’ for the LSR column. The output of the MUX-1 in Figure 4.5 is checked with the flag indicator by an AND operation. Here, a value of ‘1‘ for LSR_Fault_Verify confirms that the column belongs to LSR and contains a fault. Then the result of LSR_Fault_Verify is negated using a NOT gate to exclude the contribution of the XOR results in the HDC similarity module from the calculations through an AND gate. This operation ensures that if the column is identified as faulty, all PopCount modules for this column are deactivated. Otherwise, this has no effect on the XOR result.

In the SR, the ECC correction path is only activated if the flag associated with the column is ‘0‘, i.e., the column belongs to SR and the ECC provides the correct correction. As mentioned above, the ECC module provides two outputs, the corrected output when a single fault is detected or the original memory values are not faulty, and the bypass signal when two faults are detected. If the bypass signal indicates multiple faults or the flag does not belong to SR, the output of the ECC module is bypassed by the MUX-2 in Figure 4.5, and the original memory values are used directly. This mechanism prevents potential inaccuracies that could result from incorrect corrections if multiple faults are present.

To avoid additional latency caused by the ECC module, no additional controller is used. Instead, the pointers to the parity memory are carefully designed so that they always point to the parity memory before the corresponding column is accessed for evaluation. This functionality is made possible by having two different memory pointers and incrementing them as soon as the flag memory accesses the current region’s data. The pointers are then updated to access the next column within the region, ensuring seamless availability of data for processing.

4.3. Experimental Results

The proposed non-uniform error correction strategy is being evaluated in a wide range of applications to demonstrate its versatility. These applications include image processing (MNIST), speech analysis

(ISOLET), and human activity recognition (UCI-HAR). Table 4.1 shows the required ECC codeword for different datasets used in this chapter. To assess the fault tolerance, we simulate random bit flips in the CHV. These faults are randomly distributed across different classes within the CHV memory. Given the stochastic nature of these faults, we use a Monte Carlo simulation approach to ensure robust and statistically significant results. Each result presented in this chapter represents the average of 50 independent runs of the model, each using a different random seed to provide reliable and repeatable insights.

Table 4.1.: ECC codewords for different benchmark datasets.

Benchmark Datasets	MNIST	ISOLET	UCIHAR
ECC dimension $(codeword, databits)$	(15,10)	(32,26)	(13,8)

4.3.1. Performance Evaluation

4.3.1.1. Inference Accuracy Analyses

The proposed Hamming weight categorization streamlines the identification of LSR and SR. Instead of repeatedly performing time-consuming Monte Carlo fault injection for each dataset and implementation (e.g., encoding scheme or number of dimensions), CHw can directly and efficiently group based on their sensitivities.

Figure 4.6 presents the accuracy under the proposed non-uniform error correction strategy. The results demonstrate a significant improvement in inference accuracy compared to the baseline fault-injected system without correction. Notably, the proposed non-uniform ECC outperforms uniform ECC by effectively eliminating multi-bit faulty columns in LSR, which further enhances accuracy. Additionally, we illustrate the impact of protecting only the SR region, highlighting the difference between considering parity for LSR and ignoring potential faults in that region. As expected, the achieved accuracy is consistently higher when employing non-uniform ECC with parity protection. Although eliminating columns with parity improves accuracy, it is insufficient to rely on this approach alone. since removing multiple columns significantly affects the overall accuracy of the conclusions.

With the proposed non-uniform error correction with the SEC-DED for SR and single bit parity for LSR, the fault tolerance can be possible up to the fault probability of 1.4×10^{-1}, 6×10^{-2}, and 10^{-1}, for MNIST, ISOLET, and UCIHAR, respectively, while maintaining accuracy within the 5% of the baseline. Consequently, we also compare the fault resilience of our model against other state-of-the-art approaches, as detailed in Table 4.2. The result shows that the fault resilience achieved by the proposed strategy is superior to existing works at least 10×, making the HDC system more robust.

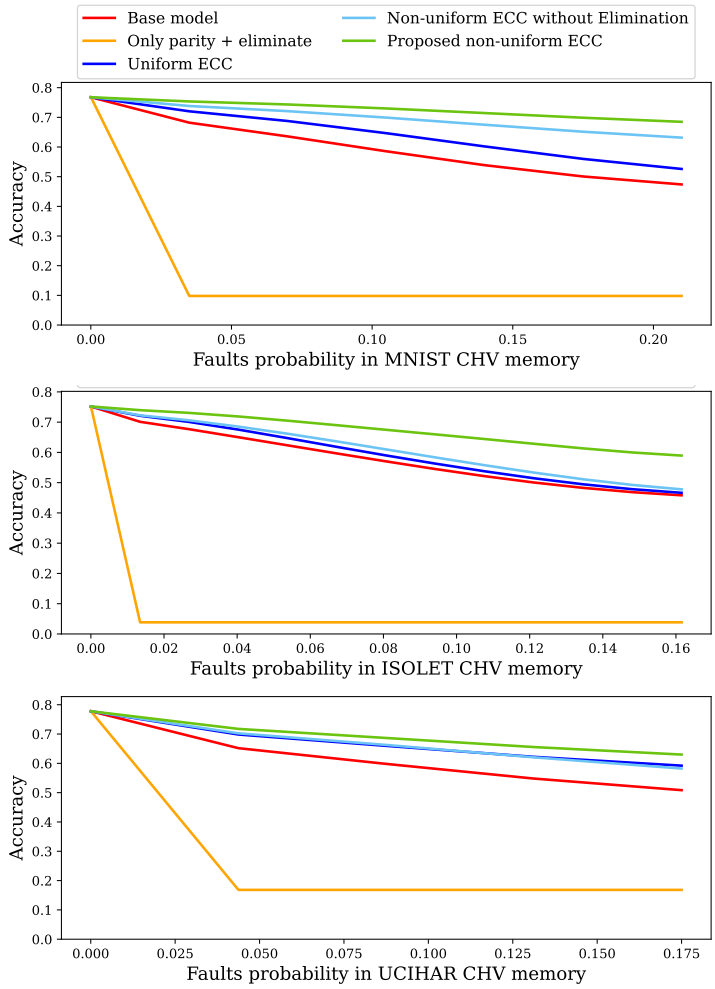


Figure 4.6.: Classification accuracy under increasing CHV memory fault probability for baseline, uniform ECC, and the proposed non-uniform correction strategy across benchmark datasets.

Table 4.2.: Fault probability achieved for different datasets.

Method	MNIST	ISOLET	UCIHAR
[53]	10^{-4}	10^{-4}	10^{-5}
[55]	10^{-2}	-	10^{-2}
[56]	10^{-3}	-	-
Proposed ECC	$\mathbf{1.4 \times 10^{-1}}$	$\mathbf{6 \times 10^{-2}}$	$\mathbf{10^{-1}}$

4.3.1.2. Hardware Implementation

The hardware overhead for different datasets is summarized in Table 4.3, enabling a comparative analysis of resource utilization between the binary HDC implementation, the uniform ECC implementation, and our proposed non-uniform ECC method. For the MNIST HDC design, classification is performed on four columns of the CHV at a time, requiring four parallel SEC-DED ECC decoding logic units to support concurrent memory error detection and correction. Similarly, for the ISOLET dataset, six parallel

Table 4.3.: resource utilization of the HDC IP and ECC module overhead on PYNQ-Z2.

Module	LUT	Registers	BRAM	Power
MNIST HDC IP	15441	20626	56	297 mW
Uniform ECC	2700	700	0	4 mW
non-uniform ECC	**394**(2.55%)	**139**(0.67%)	**0**	**4 mW**
ISOLET HDC IP	17738	48285	56	467 mW
Uniform ECC	3286	1433	0	12 mW
non-uniform ECC	**481**(2.7%)	**1042**(2.15%)	**0**	**12 mW**

Table 4.4.: A comparison between hardware overhead utilization and latency with other state-of-the-art methods.

Method	LUT	Registers	BRAM	DSP	Latency	Dataset
[53]	716	412	48	2	5ms	ISOLET
[55]	4121	4435	61	2	2ms	ISOLET
[56]	23539	35180	95	325	6ms	MNIST
Proposed ECC	1025	3102	5	0	**0*ms***	ISOLET

SEC-DED ECC decoding units are utilized to meet the error correction requirements. Compared to the uniform ECC implementation, the proposed method requires less storage overhead, as it can reduce the ECC parity bits requirement by up to 82%. The decoding hardware overhead is similar in both cases. Notably, our proposed method introduces no additional latency on top of the baseline model, as the decoding complexity of SEC-DED ECC is very low and operates in parallel with memory reads during the classification process. The final implementation floorplan on the PYNQ-z2 board is shown in Figure 4.7, which is the integration of HDC IP, proposed non-uniform ECC, and the interfacing.

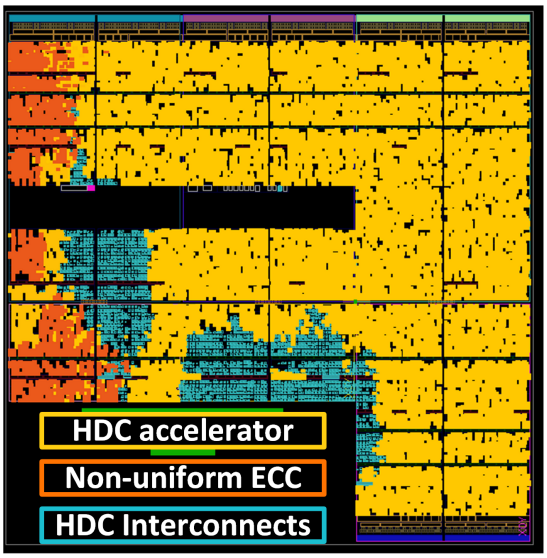


Figure 4.7.: The floorplan of the PYNQ-Z2 board after integrating the HDC accelerator with the proposed fault correction framework.

We also compare our proposed method with other state-of-the-art approaches in terms of hardware overhead and additional inference latency, as presented in Table 4.4. The reported results for [53] are sourced from [55]. The results demonstrate that, unlike related works, our method introduces no additional latency or resource-intensive components, such as BRAM or DSP, while maintaining a modest and reasonable overhead in terms of LUTs and registers.

4.4. Conclusion

Addressing memory faults in HDC systems is essential for ensuring the reliability and practicality of HDC systems in resource-constrained edge environments. In this chapter, we present a framework for non-uniform error correction that increases the robustness of the HDC accelerator by considering the Hamming weight of the memory region. We selectively apply tailored protection strategies based on the fault sensitivity of associative memory regions. Our approach achieves high fault tolerance with minimal hardware and memory overhead by leveraging statistical sensitivity analysis. Our framework achieves enhanced memory error rate resilience compared to state-of-the-art techniques without adding latency or relying on resource-intensive components like BRAM or DSP while incurring only a modest LUT and register overhead. This scalable and efficient solution paves the way for robust, energy-efficient HDC systems, enabling the deployment of cognitive models in edge devices with limited resources.

5. Memory-Free Error Correction for Edge HDC

This chapter advances fault correction by eliminating the memory overhead of conventional Single-Error Correction and Double-Error Detection (SEC-DED) ECC schemes. Although ECC improves the reliability of CHV memory, storing extra check bits is costly for small edge devices. The proposed memory-free error-correction method generates the necessary correction information at runtime instead of storing it in memory. This approach enables fault resilience under high fault rates while maintaining the low-memory and low-cost characteristics of HDC hardware [5].

5.1. Motivation and Problem Definition

The previous chapter showed that HDC edge accelerators can be made more reliable by applying non-uniform error correction to the CHV associative memory. By exploiting the fact that different memory regions have different fault sensitivities, selective protection reduces check bit memory overhead while maintaining strong resilience against memory faults, which is an important step toward practical, fault-tolerant HDC acceleration.

Although the previous chapter reduces the cost of ECC by applying protection selectively, it does not remove the need for ECC check bit storage. The protected CHV columns still require redundant check bits, and these check bits must be stored and accessed during inference. Therefore, the overhead is reduced, but a separate protection memory remains.

This is a critical limitation for edge HDC accelerators. The CHV memory already grows with the hypervector dimension, and adding check bit storage increases area, power, and memory-access cost. For small FPGA-based or ultra-low-power edge devices, this remaining memory overhead can still be too expensive. Moreover, stored check bits introduce their own reliability risk. Since they are kept in memory, they can also suffer from bit flips. A fault in the check-bit memory may weaken the correction mechanism or cause incorrect correction behavior. Therefore, the next step is not only to reduce ECC overhead, but to remove the dedicated check-bit memory altogether.

This motivates a different direction: instead of making ECC storage smaller, the goal is to avoid storing dedicated ECC check bits in the first place. In HDC, the CHV memory is protected column by column, where each column contains the class bits for one hypervector dimension. Because the hypervector dimension is usually much larger than the number of classes, the same check-bit patterns appear repeatedly across many columns. Storing these repeated check bits as a separate memory is therefore unnecessary.

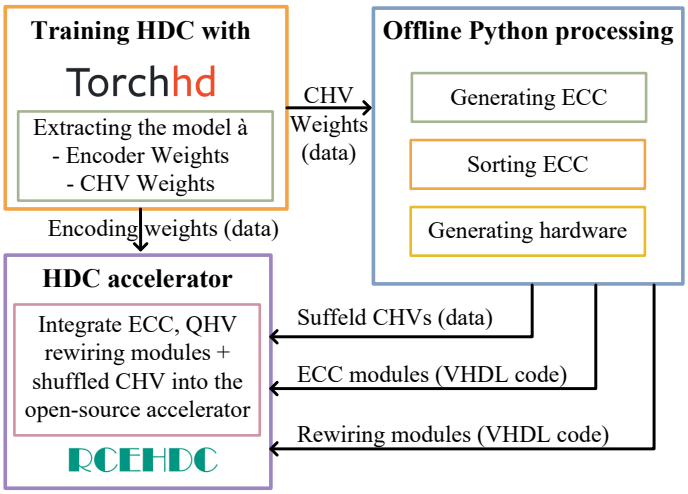


Figure 5.1.: Overview of the proposed memory-free ECC framework. Trained CHVs are processed offline to generate reordered memories, rewiring logic, and lightweight check-bit generation hardware for runtime correction.

Based on this observation, this chapter introduces a memory-free error-correction method for HDC. The proposed method reorders the CHV columns according to their check-bit patterns and stores only a compact set of indices indicating where the pattern changes. During inference, the required check bits are generated on-the-fly from these indices rather than read from dedicated memory. This preserves runtime correction capability while eliminating the check-bit memory overhead.

The proposed method therefore removes the need for stored ECC check-bit memory while preserving runtime correction capability. This reduces memory overhead, eliminates faults in the check-bit memory itself, and maintains the low-latency characteristics required for edge deployment. By replacing redundant storage with deterministic generation, the method makes correction more scalable with hypervector dimension.

5.2. Proposed Memory-Free Fault Correction

Our main goal in this work is to enhance the resilience of inference phase to runtime memory faults. To achieve this, we eliminate the conventional ECC check-bit memory and replace it with a lightweight, logic-based, on-the-fly check bit generator. This replacement eliminates the need for additional memory while maintaining robustness against memory faults.

Our key insight is that CHV memory is typically long (e.g. $D = 10{,}000$) and narrow ($C = 10$), leading to highly repetitive ECC check-bit patterns. For example, a 5-bit ECC check-bit results in only 32 unique values, each of which is repeated thousands of times across the dimensions. Instead of storing these explicitly, we sort the dimensions according to their ECC check-bit values and store only the transition indices, which enables on-the-fly generation of check bits using a simple counter.

The end-to-end view of the memory-free ECC is illustrated in Figure 5.1, which is explained in detail in this section. The rest of this section is organized as follows: First, we present our memory-free correction

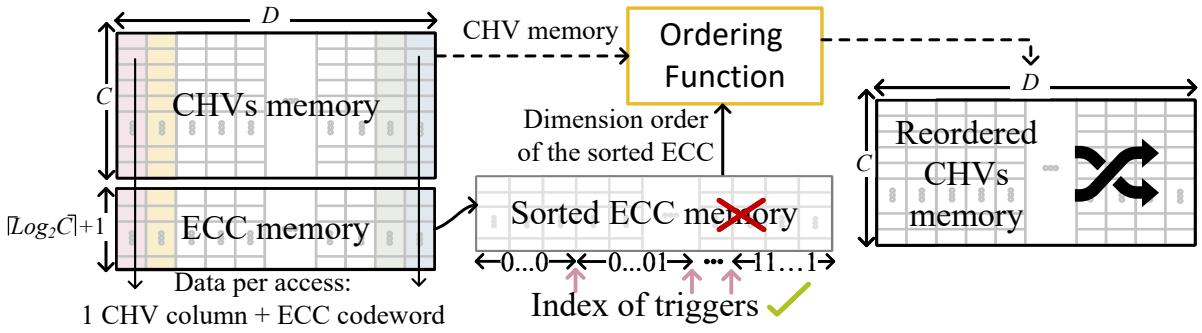


Figure 5.2.: The memory layout of CHVs ($D \times C$) and the corresponding ECC memory ($D \times \log_2(C) + 1$) in the case of convolutional SEC-DED. In each access, one column along with the associated ECC check-bit is read ($C + \log_2(C) + 1$). In the software processing stage, the traditional ECC memory storing full check bits is replaced by storing only the indices of check bits increments (index of triggers). The dimension order of the sorted ECC codewords is then applied to reorder the CHVs.

approach. Then, we briefly describe the required software processes. Finally, we demonstrate how the method is seamlessly integrated into the HDC pipeline. In the rest of this chapter, D and C denote the dimension size and number of classes, respectively.

5.2.1. On-the-Fly Memory Generation

Efficient fault correction is better applied at the dimension (or column) level during classification, rather than protecting each full CHV, which typically contains thousands of bits (Figure 5.2). Full CHV protection introduces high overhead and compromises HDC scalability. Moreover, many HDC accelerators, such as [2, 81], already access CHV memory per dimension for computation.

This architectural feature makes dimension-wise correction both efficient and naturally aligned with typical memory access patterns in hardware HDC implementations. It is also important to note that each dimension in the CHV is independent of the others and contributes equally to the calculation of similarity score. As long as each dimension is compared to the corresponding dimension in the QHV, the order in which the dimensions are accessed or processed does not affect the final result or the accuracy of the HDC model's inference. In the following, we describe the ECC by showcasing the SEC-DED. However, the approach is not limited to SEC-DED and it can be extended as explained in more detail later in subsection 5.2.2.

During classification and per memory access, the system retrieves C bits from the CHVs and $\log_2(C) + 1$ bits from the corresponding check-bit.

Each column in the CHV memory can represent the following Number of Possible Values (NPV):

$$NPV_{\text{dimension}} = 2^C \tag{5.1}$$

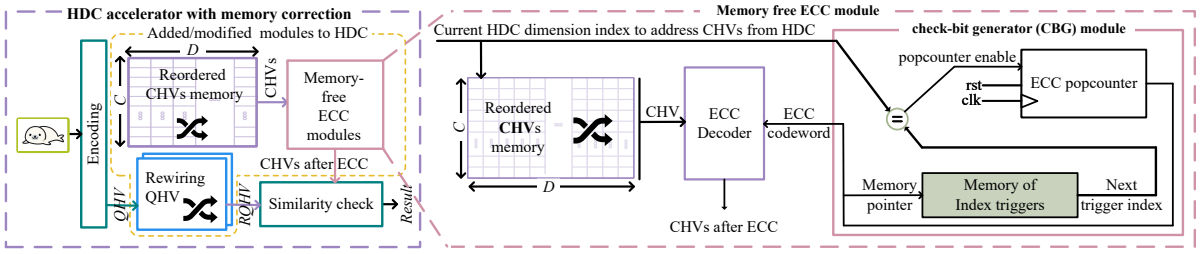


Figure 5.3.: Block diagram of the HDC accelerator with memory-free ECC correction. The encoding output is processed by the QHV rewiring and the reordered CHVs memory, followed by the memory-free ECC module. The ECC module includes the check-bit generator (CBG), which uses a PopCount and a memory with index triggers to generate ECC check-bit on-the-fly. The ECC decoder then corrects the reordered CHVs before the similarity check.

This reflects the number of possible bit combinations, which varies depending on the classification task (i.e., the number of classes: C). In contrast, each column in the check-bit memory has significantly fewer possible values, given by:

$$NPV_{\text{ECC check-bit}} = 2^{\lceil \log_2 C \rceil + 1} \tag{5.2}$$

This shows that $NPV_{\text{ECC check-bit}} \ll NPV_{\text{dimension}}$, which means that the ECC check-bit memory contains substantial redundancy. Only a limited number of unique check bits (as defined in Equation 5.2) are actually required, but these values are replicated thousands of times in the check-bit memory.

This redundancy motivates our proposed approach. By storing the ECC check-bit in ascending order, we enable on-the-fly generation using lightweight state tracker (PopCount) logic, eliminating the need for dedicated check-bit memory. With this approach, only the indices where the check-bit values are incremented need to be stored. Consequently, the memory required in bits to store these indices is

$$\text{Index_Memory} = 2^{\text{ECC_bits}} \times \log_2 D \tag{5.3}$$

which is significantly smaller than the original ECC memory:

$$\text{ECC_Memory} = (\lceil \log_2 C \rceil + 1) \times D \tag{5.4}$$

Besides, as shown in the Equations (5.4), the size of the ECC memory is linearly proportional to the size of CHV; i.e., O (D). Whereas, according to Equation (5.3), the size of the index memory is O ($log_2 D$). This quantitative analysis shows that, besides memory saving, on-the-fly generation of the ECC bits offers a more scalable solution for larger CHV dimensions.

To apply our on-the-fly ECC memory generator, we develop an automated offline process that generates all necessary modules for integration into the HDC accelerator. This process is described in the following subsection.

5.2.2. Offline Processing

The goal of offline processing is to convert the trained CHVs into a reordered format that is compatible with memory-free ECC correction. This requires some modifications to the standard HDC accelerator and its components. The offline process takes the trained CHVs as input and generates the reordered CHV memory, index trigger memory, sorted ECC values, and a rewiring map for the QHV. In this section, we describe in detail how this integration bridge works.

After training, the obtained CHVs weights are passed to our offline processing unit. In the first step, this unit generates and sorts the ECC check-bit values associated with each dimension in the CHV memory. After sorting, the index corresponding to the ascending order of the ECC check-bit values is recorded as an *index trigger memory*, which is stored in the hardware for later use. The new order of the dimensions is also saved and used as an index for the subsequent reordering steps. To maintain functional correctness during classification, the same order must also be applied to the CHVs memory and the QHV. To achieve this, a new CHV memory group is created in which each dimension is reordered based on ascending ECC values. The lightweight rewiring module is introduced to reorder the QHV. By aligning the dimension order of both CHVs and QHV, each dimension is matched correctly during similarity calculations. This ensures that the classification accuracy remains intact.

Moreover, the proposed method is scalable and independent of the number of classes. As comparisons in HDC are inherently parallel and class-independent, different classes within CHVs operate independently, the CHV memory can be divided into multiple groups of classes to further increase fault tolerance. For each group, ECC generation, CHV reordering, and QHV rewiring can be performed independently. This grouping enables localized multi-bit error correction with minimal overhead, as the ECC module operates without storing full ECC check-bit values.

This architectural transformation generates ECC check-bit on-the-fly during classification, eliminating the need for a separate ECC check-bit memory and preserving the robustness of the system. The implementation details and integration into the HDC accelerator are discussed in the following.

5.2.3. ECC Check-bit Generator and Decoding Hardware

As described in subsection 5.2.2, the initialized CHVs are reordered based on their ECC check-bit values, and the rewiring modules align the QHV accordingly. In the revised HDC architecture (Figure 5.3), encoding parameters are loaded from the trained model, while classification components—such as reordered CHV memory and ECC modules—are generated through the offline process as detailed in subsection 5.2.2. Instead of storing check-bit, they are generated on-the-fly using lightweight logic and used with the check-bit for fault correction before similarity checking.

Figure 5.3 shows an overview of the HDC hardware architecture, integrated with the ECC-enabled CHV memory. As shown in the figure, the QHV is routed through the rewiring module to create a Reordered Query Hypervector (RQHV). The RQHV is then forwarded to the similarity check module of the HDC. At the same time, the reorder CHVs is processed by the ECC correction module and also

forwarded to the similarity check. Note that our proposed on-the-fly ECC check-bit generator does not impact the ECC decoder module. This coordinated processing ensures that both the query and reference hypervectors are aligned and corrected consistently, maintaining classification accuracy while introducing fault tolerance.

This reordering has no impact on system performance. As the CHVs can be initialized directly in the reordered format and the rewiring required to reorganize the QHV introduces only a minimal logic overhead, as it is performed automatically by the synthesis tools during the hardware mapping process. The only hardware overhead incurred by this architecture is the inclusion of the ECC and Check-Bit Generator (CBG) modules, both of which are lightweight, as shown in Figure 5.3.

The CBG module contains a PopCount that is only activated when the system switches to a new ECC check-bit value. This event-driven activation minimizes the power consumption and avoids latency during fault-free operation compared to memory access, as the next check-bit is always available. The output of the PopCount serves two primary purposes: firstly, it generates the current ECC check-bit; Secondly, it selects the corresponding entry from the trigger index memory.

The trigger index memory continuously outputs the next trigger value. If this value matches the current dimension index provided by the HDC accelerator, the PopCount is triggered. At this moment, a new ECC check-bit is generated, and the trigger index memory advances to the next transition point. This design ensures that the system efficiently accounts for the ECC transitions while keeping the overhead low and maintaining reliability.

5.3. Experimental Results

The memory-free fault correction strategy is evaluated in a wide range of applications, demonstrating its versatility. These applications include image processing (MNIST), speech analysis (ISOLET), and human activity recognition (UCI-HAR) with $C = 10,\ 26,\ 8$ respectively.

To evaluate the fault tolerance, we simulate random bit flips in the CHV. Using the Monte Carlo approach, these faults are randomly distributed across different classes within the CHV memory. The result in this chapter represents the average of 20 independent runs of the model, each with a different random seed, to ensure reliability and repeatability.

5.3.1. Performance Evaluation with Faults

5.3.1.1. Inference Accuracy Analyses

Figure 5.4 presents the accuracy under the proposed memory-free error correction strategy. The results demonstrate a significant improvement in inference accuracy compared to the baseline fault-injected system without correction. Notably, the proposed memory-free ECC outperforms the standard ECC that utilizes memory to store ECC check-bit, as check bits might also get faulty. The wider range of

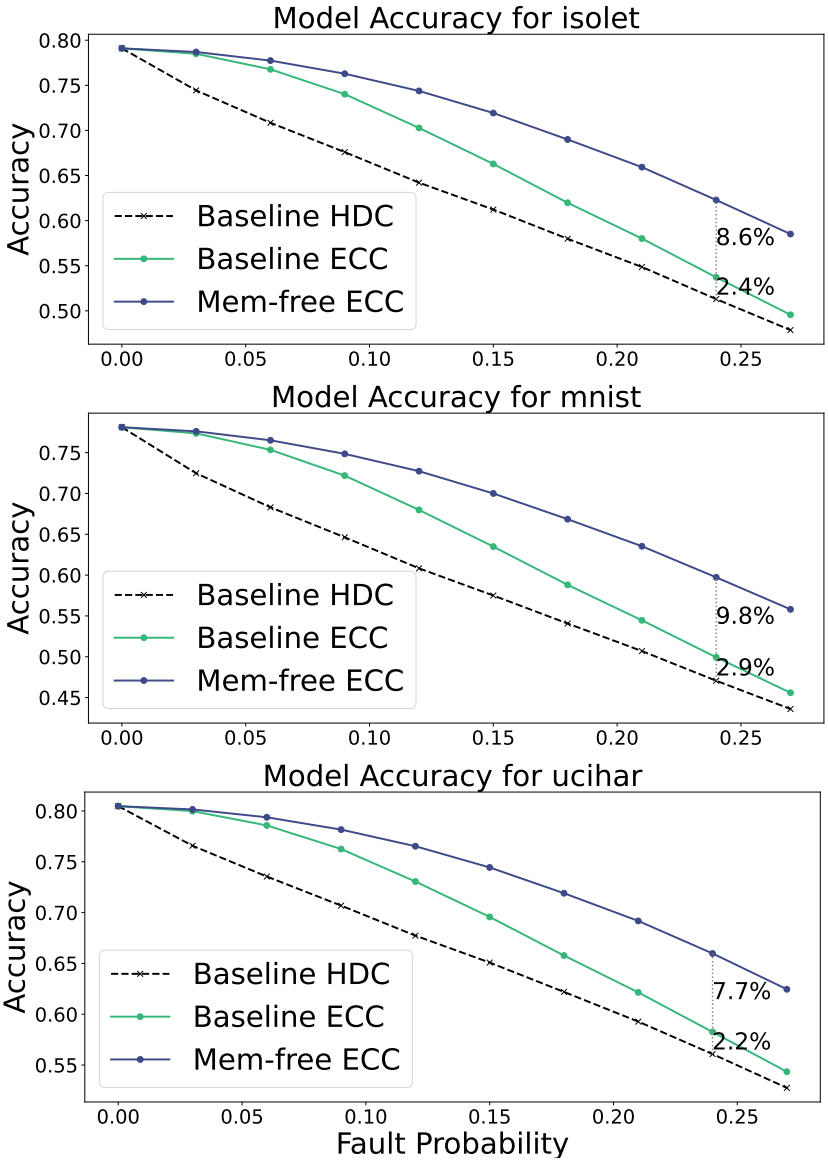


Figure 5.4.: Effect of error probability on model accuracy with baseline and memory-free ECC strategies applied to three different datasets, using $D = 5k$ and *random projection encoding*, showing the accuracy outperformance of mem-free ECC.

comparison based on different HDC configurations is shown in Table 5.1, showing the effectiveness of the method by 24% of faults, as also marked in Figure 5.4.

Consequently, we also compare the fault resilience of our model (below 5% accuracy drop) against other state-of-the-art approaches, as detailed in Table 5.2. The result shows that the fault resilience achieved by the proposed strategy is superior to existing works at least 12×, making the HDC system more robust to memory faults.

The proposed method is not limited to SEC-DED. By dividing the CHV into several class groups, we can increase the number of correctable bits for each dimension, as shown in Figure 5.5, as the classes in the similarity function are independent, and each group only needs its own hardware matcher.

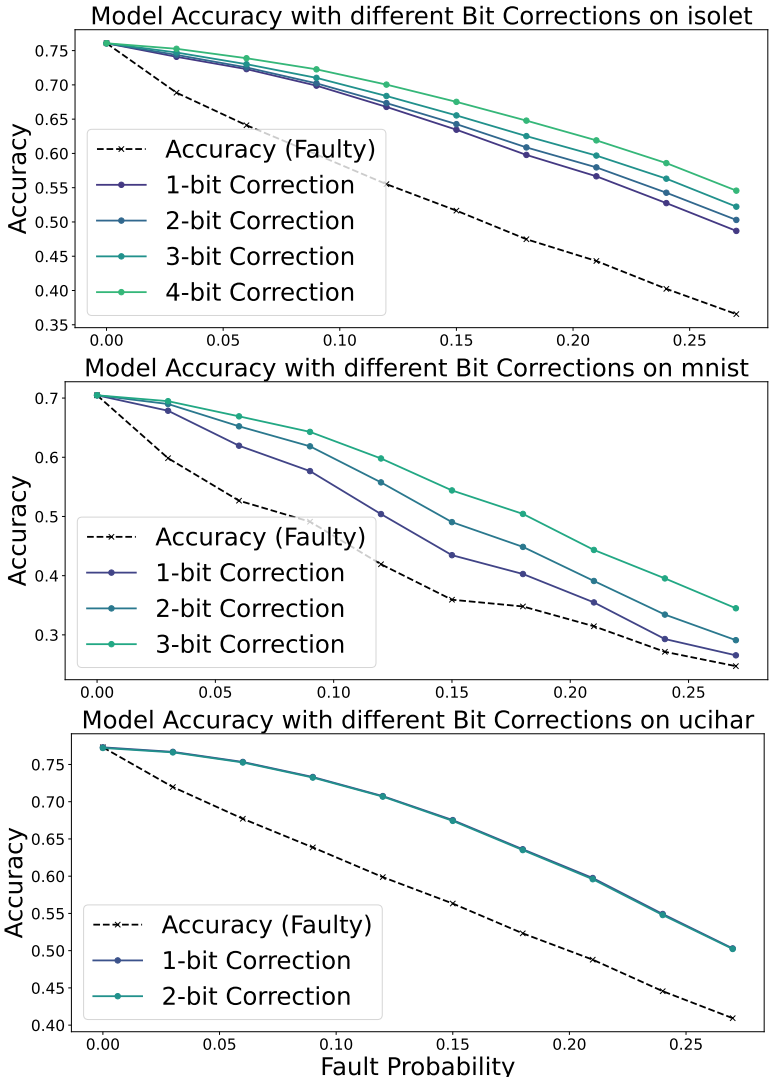


Figure 5.5.: Effect of multi-bit error correction on model accuracy for three different datasets, using $D = 1k$ and *base-level encoder*.

5.3.1.2. Hardware Implementation

The hardware overhead analysis for the MNIST dataset is summarized in Table 5.3 and provides a direct comparison between the binary HDC implementation (D = 1k), the baseline ECC version, and the proposed memory-free ECC architecture. The results show that compared to the HDC design, our memory-free ECC incurs minimal hardware overhead of only 0.23% and 0.02% for LUT and register, respectively. In contrast to the baseline ECC, which requires BRAM for syndrome decoding, our method achieves zero BRAM utilization, resulting in a completely memory-free design. In addition, power consumption remains extremely low (≤1 mW) and the simplicity of the decoding logic ensures that no additional latency is introduced during classification. The ECC decoding runs in parallel with the memory read operations so that the throughput is identical to that of the basic HDC system.

Table 5.1.: Accuracy improvement (in %) of the baseline and memory-free ECC modules compared to plain HDC under a 24% fault injection rate. On average, the memory-free ECC achieves approximately 8% higher accuracy improvement than the baseline ECC.

Encoding	Datasets	D	Baseline ECC(%)	Memory-free ECC(%)
Base-level	ISOLET	1k	3.3	9.2
		10k	2.2	9.1
	MNIST	1k	1	1.1
		10k	3.9	12
	UCIHAR	1k	1.9	8.5
		10k	2.5	9.7
Random projection	ISOLET	1k	2.6	8.7
		10k	2.2	8.1
	MNIST	1k	2.8	9.9
		10k	2.8	9.3
	UCIHAR	1k	2.1	7.6
		10k	2.2	7.7

Table 5.2.: Maximum fault probability tolerated for different datasets before the classification accuracy drops by more than 5%.

Method	MNIST	ISOLET	UCIHAR
[53]	10^{-4}	10^{-4}	10^{-5}
[55]	10^{-2}	-	10^{-2}
[56]	10^{-3}	-	-
Memory-free ECC	$\mathbf{1.2\times10^{-1}}$	$\mathbf{1.2\times10^{-1}}$	$\mathbf{1.5\times10^{-1}}$

Figure 5.6 underlines the scalability and efficiency of the memory-free ECC. When the hypervector dimensionality D increases from 1k to 60k, the memory usage of the ECC base system scales linearly, i.e., O (D) for both MNIST and ISOLET datasets. In contrast, our memory-free approach maintains a logarithmic memory growth(O ($log_2 D$)), thereby negligible memory footprint across all dimensions.

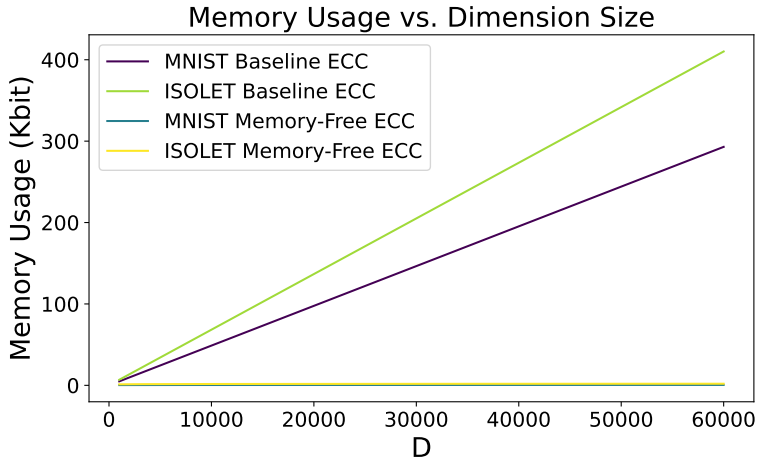


Figure 5.6.: Memory utilization on MNIST and ISOLET datasets, showing a linear and logarithmic memory utilization growth for a baseline and memory-free ECC, respectively.

Table 5.3.: Comparison of scalability of resource utilization between baseline ECC, and proposed memory-free ECC for different D , for C = 10. All the results reported from Vivado on PYNQ-Z2

D	Module	#LUT	#Registers	#BRAM	Power
	MNIST HDC	10840	20760	56	377 mW
1000	Uniform ECC	19	0	1	4 mW
	Mem-free ECC	**26 (0.23%)**	**5 (0.02%)**	**0**	$\geq$ 1 **mW**
10000	Uniform ECC	19	0	2.5	9 mW
	Mem-free ECC	**27**	**5**	**0**	$\mathbf{\geq 1mW}$

Table 5.4.: A comparison between hardware overhead utilization and latency with other state-of-the-art methods.

Method	LUT	Registers	BRAM	DSP	Latency	Dataset
[53]	716	412	48	2	$5ms$	ISOLET
[55]	4121	4435	61	2	$2ms$	ISOLET
[56]	23539	35180	95	325	$6ms$	MNIST
Proposed ECC	**26**	**5**	**0**	**0**	$\mathbf{0ms}$	ISOLET

This makes our approach particularly attractive for embedded or edge devices with strict area and power constraints.

Table 5.4 presents a comparison of the hardware overhead and latency between the proposed memory-free ECC and other recent state-of-the-art error correction techniques for HDC. The previous work [53, 55, 56] mostly utilizes more complex error detection on multi-bit HDC CHVs, making them inappropriate for resource-constraint edge devices.

In contrast, our proposed ECC achieves error correction with a lightweight, memory-free design that uses only 26 LUTs and 5 registers, with no BRAM or DSP utilization, without incurring any extra latency. This makes our solution highly efficient and scalable, especially for resource-constrained platforms such as microcontrollers or low-power FPGAs, while ensuring robust fault tolerance in HDC systems.

5.4. Conclusion

In this chapter, we have presented a novel memory-free fault correction technique for HDC classifiers that is specifically designed to meet the stringent requirements of edge devices. Our method enables highly scalable and efficient protection without incurring the memory overheads associated with conventional ECC techniques. Instead of storing the ECC check-bit, our approach centers around on-the-fly generation of them, without requiring the use of BRAMs and DSPs. This way, our approach eliminates not only the overhead but also the potential for faults in the ECC check-bit memory itself. Importantly, the proposed technique maintains system performance and preserves throughput without impacting the critical path. Our FPGA implementation validates the practicality and robustness of the solution, making it a compelling choice for fault-tolerant, edge-based learning systems.

6. OOD Detection in HDC Accelerators

This chapter addresses runtime uncertainty caused by out-of-distribution inputs in HDC accelerators. Although HDC classifiers always assign each input to one of the known classes, the input may originate from an unknown or unsupported distribution. The proposed method detects such unreliable inputs by reusing similarity information already computed during HDC classification [7].

6.1. Motivation and Problem Definition

The previous chapters focused on making HDC accelerators efficient and reliable against hardware faults. They reduced memory cost, improved fault tolerance, and protected the stored model from memory errors. However, reliable hardware alone is not sufficient. Even if the accelerator operates correctly, the system may still produce an unsafe prediction if the input is abnormal.

This problem occurs when the input does not match the data distribution used during training. Such inputs are called Out-of-Distribution (OOD) samples. In real edge deployments, edge devices often operate in open, noisy, and changing environments, where sensor readings, operating conditions, and surrounding environments can vary over time. Consequently, OOD samples may result from corrupted or noisy sensor data, sensor aging, calibration drift, unseen operating conditions, environmental changes, communication errors, or inputs that do not belong to any trained class. If these inputs are processed as normal Identifier (ID) samples, the classifier still returns one of the known labels, even though the decision may not be reliable.

Therefore, functional safety in edge HDC systems requires more than protection against hardware faults. The accelerator should also recognize when an input is outside the model's reliable operating region. In such cases, the system can reject the prediction, trigger a fallback mechanism, or request further processing, rather than silently returning an unreliable label.

Many existing OOD detection methods are designed for neural networks and are not suitable for HDC accelerators. They may require additional training, auxiliary models, internal activations, input perturbations, multiple inference passes, or complex post-processing. These requirements conflict with the low-latency, low-power, and small-area goals of edge hardware.

To address this limitation, this chapter introduces a lightweight concurrent OOD detection mechanism for HDC accelerators. The proposed method monitors the maximum similarity score that is already produced during classification and compares it with a threshold learned from in-distribution training

samples. Since this score is already available in the classifier, the detector does not require retraining, an additional model, or a second inference pass.

In this way, the chapter extends the reliability scope of the thesis from hardware-induced faults to input-induced uncertainty. It shows that OOD detection can be added to an HDC accelerator with negligible hardware overhead and no additional latency, providing a practical functional-safety layer for edge deployment.

6.2. OOD Detection in HDC Accelerators

The aim of this work is to devise a lightweight, hardware-efficient, concurrent OOD detection mechanism for HDC with high coverage that works well across various encoding schemes, training parameters, and datasets. Our proposed method is seamlessly integrated, with no overhead or modification to the training process and negligible overhead at the hardware interface, making it suitable for resource-constrained edge devices. The detection mechanism relies solely on the maximum similarity of the input QHV to the CHVs during the classification, which can be done by a single comparator.

6.2.1. OOD Detection Methodology

At the level of the similarity check in HDC hardware, the similarity of the QHV with all pre-stored CHVs is calculated, as explained in chapter 2. Since the CHV are constructed based on the training data, the computed similarity between the QHV and each CHV indicates the degree to which the input matches a given class. Our method classifies the input into ID or OOD based on the maximum similarity score returned by the HDC classifier, which will be referred to as the **outlier score** in the rest of this chapter.

Figure 6.1 shows an example of the distribution of ID and OOD outlier scores. For determining if a sample should be flagged as OOD, a threshold τ is required. Any input that generates a score lower than τ is labeled as OOD. We initiate the τ to accept 95% of IDs. In detail, our method works as follows:

- **Setup Phase → Set Initial Threshold (τ):** After the training phase, before extracting the model to hardware, we initialize τ to a value that accepts p% (default is 95%) of sorted outlier scores X in ID samples in the training set, as in Equation 6.1 and Equation 6.2.

$$r = \frac{100 - p}{100} \times (n - 1) \tag{6.1}$$

$$\tau = (1 - g) \times X_j + g \times X_{j+1}, \tag{6.2}$$

  Here, n is the number of elements in X, index j is the integer component of r and g is the fractional part of r.

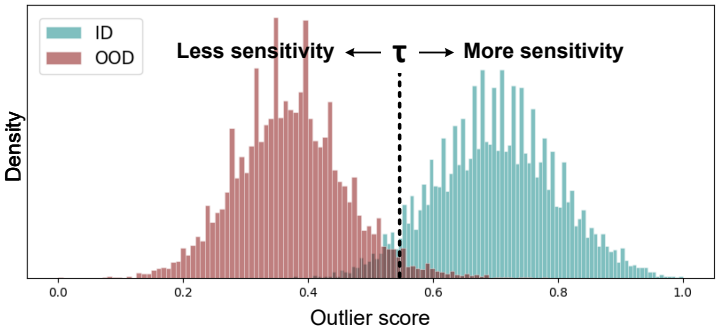


Figure 6.1.: Adjusting τ value based on the ID outlier score

- **Inference Phase → Classification Based on Threshold:** At runtime, while the outlier score x is computed by HDC.

$$f(x,\tau) = \begin{cases} \text{accept}, & \text{if } x \geq \tau \\ \text{reject}, & \text{otherwise} \end{cases} \tag{6.3}$$

The threshold τ can be adjusted during runtime based on the application's sensitivity requirements. Activating this flag triggers the activation of another layer of monitoring system, which is both energy and computationally intensive, so it should only be issued when necessary. Such setting allows for the OOD flag to be confirmed or canceled, with the resulting feedback used to further adjust the threshold. Increasing τ makes the mechanism more sensitive to OOD samples, but at the cost of potentially misclassifying ID samples.

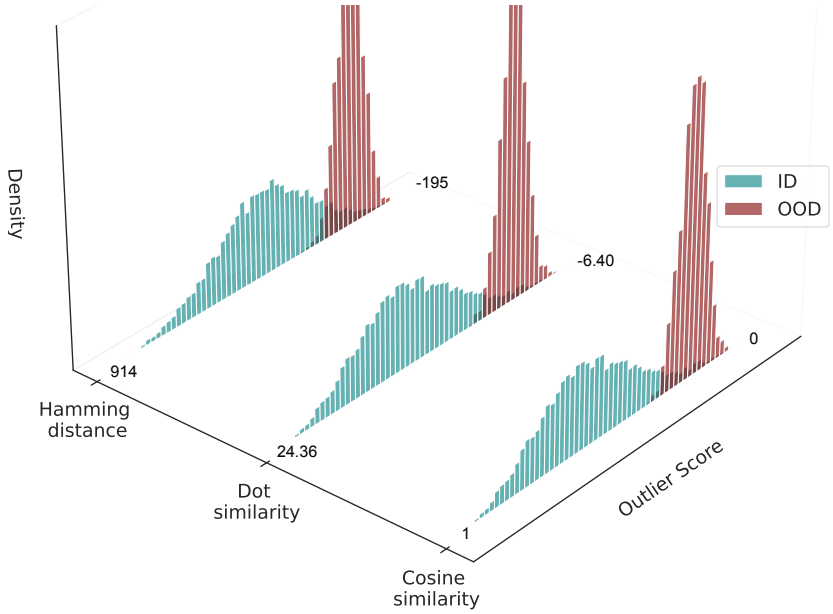
Figure 6.2 provides an overview of the distribution of outlier scores for ID and OOD samples, here chosen as gaussian noise. In Figure 6.2-a, we show the different distribution of MNIST ID and OOD outlier scores with different similarity functions for D = 1000. Figure 6.2-b presents the normalized histograms of the outlier score range of three HDC models trained on different ID datasets. Regardless of similarity function and dataset, the distribution of ID outlier scores is condensed near 0.9, while for OOD, it is widely spread over the whole possible range.

As shown in Figure 6.3, the recognition process is performed with a single comparator immediately in the final level of HDC, just before the classification results are reported. The OOD flag is issued concurrently as the output is generated.

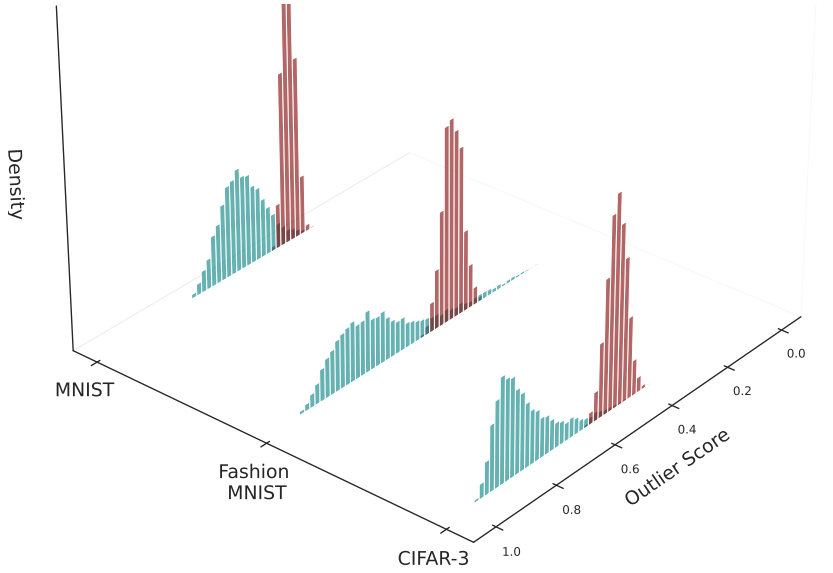
6.2.2. Evaluation Metrics

We employed Receiver Operating Characteristic (ROC) analysis as in [24]. ROC analysis provides insight into the model's ability to differentiate between classes, especially in binary classification scenarios, such as correctly recognizing OOD instances. The ROC curve shows the trade-off between the true-positive rate (TPR) and false-positive rate (FPR) when the threshold τ is adjusted in the ID score range. Our assessment includes the following established criteria:

- *AUC* or the Area Under the ROC Curve captures the overall performance of the model.



(a) The effects of similarity functions on outlier scores on MNIST



(b) ID and OOD outlier scores for three datasets

Figure 6.2.: Outlier-score distributions for in-distribution (ID) and out-of-distribution (OOD) samples. Blue indicates ID samples and red indicates OOD samples. Clearer separation between the two distributions indicates stronger ID/OOD distinguishability for the evaluated datasets or scoring schemes.

- *AUPR* categorized as *AUPR_IN* for in-distribution examples and *AUPR_OUT* for OOD examples, capture the accuracy of the model in classifying positive instances inside and outside the primary class, respectively.
- *FPR-95* is the FPR at a fixed TPR of 95%.

Higher values for AUC, AUPR indicate a better ability to distinguish between positive and negative instances, with a value of 1.0 representing a perfect classifier. In real-world scenarios, we do not want to

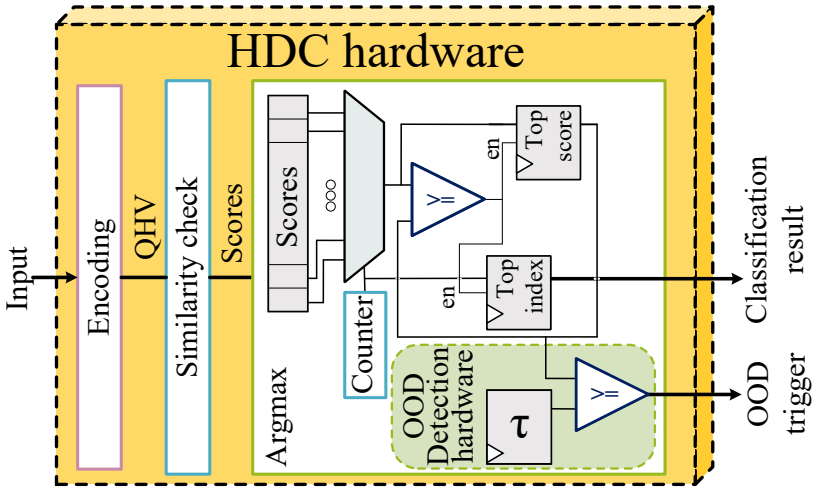


Figure 6.3.: Concurrent hardware implementation of the proposed OOD detector. The detector reuses the maximum similarity score produced by the HDC classifier and compares it with a calibrated threshold, enabling OOD detection without retraining, auxiliary models, or additional inference latency.

enter a fail-safe state more often than necessary, making the FPR-95 a highly significant metric. In addition to the capabilities of the proposed method, we provide the resource utilization of our implementation in relation to the HDC accelerator.

6.2.3. OOD Detection Robustness

We analyze the generalizability and robustness of of our method with various HDC parameters and datasets and systematically investigate the effects of choosing different configurations and training procedures for HDC models. The inherent OOD detection capabilities of HDC models are compared depending on the following key parameters:

- *Encoding techniques:* Affect not only the performance of HDC but also the distributions of HVs
- *Dimension size:* The size of HVs has a significant impact on hardware resources, runtime, and accuracy
- *Precision:* The quantization, affecting the trade-off between computational efficiency and model accuracy
- *Similarity function:* Depends on the application due to their difference in accuracy and computational effort

For all mentioned parameters, traditional one-shot and iterative online training [32] is taken into account, as it has an effect on the model accuracy. Furthermore, the generalizability of OOD detection is evaluated by varying the *datasets*, which the model is trained on, as well as choosing a diverse set of inputs for OOD data. The *types of OOD data* can be ranging from close OOD (e.g. corrupted inputs) to more diverse distributions, to allow a comprehensive analysis of OOD detection.

Table 6.1.: Summary of configuration factors

Configuration Factors	In this evaluation
Encoding	Base-level, Random projection
Precision	Float32, Binary / Bipolar
Similarity Function	Hamming, Dot, Cosine
Dimension Sizes	1000, 5000, 10000
Training Procedure	One-shot, OnlineHD [32]

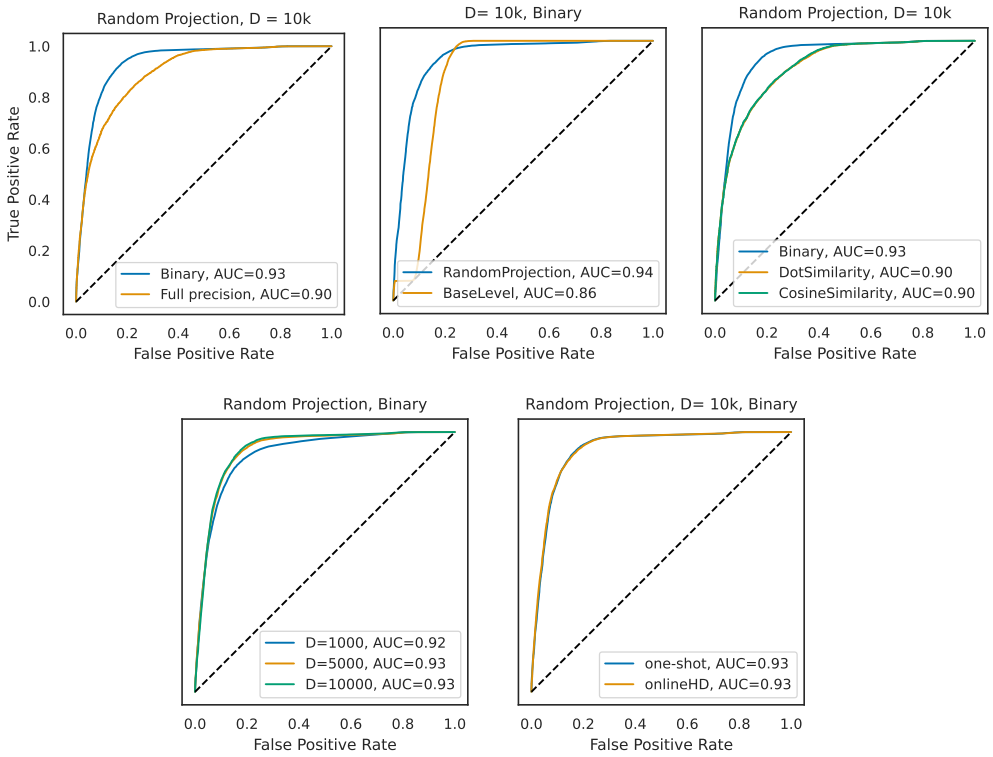
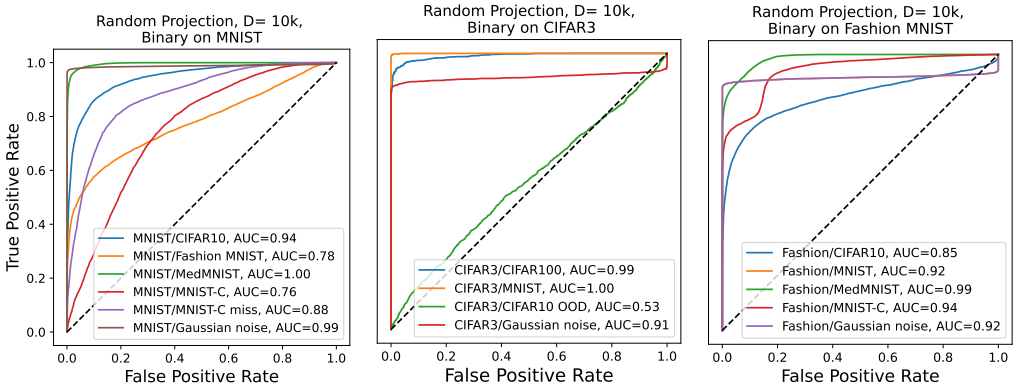


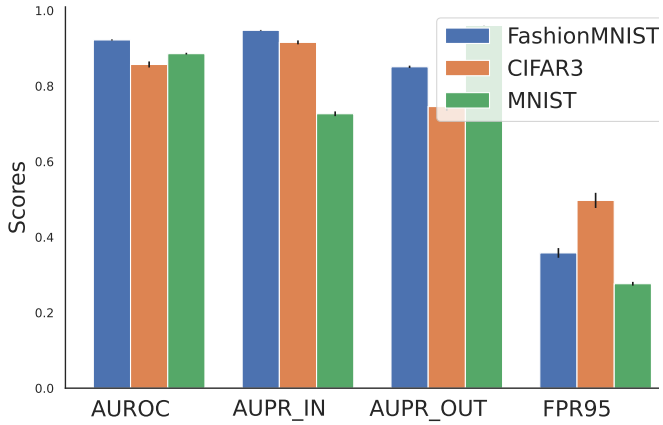
Figure 6.4.: Effect of HDC configuration parameters on OOD detection behavior. The results show how representation and model settings influence the separation between ID and OOD scores and, consequently, the detection performance.

To investigate the effects of variations in HDC parameters, we train several HDC models with different configurations. The parameters considered are summarized in Table 6.1. For encoding methods, we consider base-level and random projection encoding. Furthermore, to examine the impact of *quantization*, we analyze the two most disparate scenarios: full precision and quantization to a single bit (binary/bipolar level), which also affects the chosen *similarity function*. For binary/bipolar level, hamming distance, and for full precision, both dot similarity and cosine similarity are evaluated. We analyze the effects of the following *dimension sizes*: 1000, 5000, and 10000. Also, the HDC models are first trained with a normal one-shot training, and then with the iterative online training proposed in [32].

We train 12 models with different configurations on MNIST, FashionMNIST, and CIFAR-3. Five OOD sets are used for MNIST and FashionMNIST, including MNIST for FashionMNIST and vice versa, resized



(a) OOD detection across datasets



(b) Similarity-score variation

Figure 6.5.: OOD detection results across benchmark datasets and evaluation settings. The results demonstrate the effectiveness of using the native HDC similarity score as a lightweight indicator of input uncertainty.

CIFAR-10, MedMNIST, MNIST-C, and Gaussian noise. In the following, we denote combinations of ID data, which a model is trained on, and OOD data as *ID/OOD*.

The MNIST-C dataset consists of 15 variants of distortions to MNIST dataset, which can be considered a close OOD dataset for MNIST. Therefore, for MNIST/MNIST-C, we consider the OOD detection rate for the entire MNIST-C dataset and check the OOD detection only on the misclassified samples. Also, due to the inconsistent number of samples in the different datasets within MedMNIST, random sampling was performed for MedMNIST. To minimize the impact of sampling on the results, we conducted 20 random sampling sets and report the average of the calculated metrics. Moreover, for MNIST/CIFAR-10, we resize the CIFAR-10 dataset to 28×28 pixels and transform it to single-channel grayscale pictures.

Since HDC does not perform well for the task of CIFAR-10 (less than 30% [33]), we reduce the dataset to the first three classes to reach around 60% accuracy. Here, we consider CIFAR-100 [66], MNIST, the remaining seven classes in CIFAR-10, and Gaussian noise as potential OOD data. We apply 2×2 zero padding to the MNIST dataset to make it suitable as input for the CIFAR-3 model and transfer the images to grayscale RGB in 3-channel format.

For the hardware implementation, we use the framework from chapter 3 and implement it on the AMD Xilinx PYNQ-Z2, in which The input data can be sent from the ARM core of the device to the HDC accelerator running on the FPGA.

6.3. Experiments and Implementation Results

In this section, we first present the results of OOD detection on different datasets, analyze the influence of HDC parameters on detection efficiency, and compare our approach with state-of-the-art methods. Finally, we report the resource utilization and OOD detection rate on the PYNQ-Z2 board using the MNIST dataset as the ID benchmark.

6.3.1. Software Detection Results

We explore the efficacy of our concurrent OOD detection across varying HDC configurations, visualized in Figure 6.4 using the MNIST dataset. These ROC curves illustrates the impact of altering a single HDC model parameter while keeping others constant. Each ROC curve is accompanied by an annotation of these constant parameters at the top.

We investigate OOD detection using models trained on ID datasets, as shown in Figure 6.5. The ROC curves in the three figures depict OOD detection for different ID datasets on a quantize model with random projection encoding, and D = 10000. The barchart in Figure 6.5 shows the average and standard error of OOD metrics across all configurations and datasets. Generally, AUC variance is low for OOD datasets distant from ID data. Detecting OOD samples becomes challenging when they resemble ID data. Specifically, for MNIST/MNIST-C, detection is harder compared to other OOD datasets. A similar trend is observed in CIFAR3, where identifying the excluded 7 classes is notably more difficult.

Table 6.2 summarizes the average results for all OOD datasets with different HDC configurations trained on MNIST. The configurations are categorized by encoding (base-level or random projection), precision (Float32 or Binary), dimensionality (D), similarity function (cosine similarity is omitted due to negligible differences from hamming and dot similarity), and training approaches (standard and online retraining as in [32]).

OOD detection improves with higher dimensionality of HV, although this is not directly related to accuracy. For example, a binary model with base-level encoding shows up to 10% higher accuracy with increased dimensionality but constant FPR95. It is noteworthy that retraining often significantly reduces the FPR95 compared to standard training. Regardless of the training method, FPR95 is more stable and on average 30% lower for base-level encoding than for random projection.

We compare our results with existing methods that target NNs in Table 6.3. We train a binary Multi Layer Perceptron (MLP) with 3 layers of 64 neurons each and an output layer of 10 neurons on MNIST and achieve a test accuracy of 93%. Although the classification accuracy of HDC is 10% lower than that of

Table 6.2.: Results averaged over all OOD datasets for various HDC configurations trained on MNIST. Configurations with the highest and lowest FPR95 per encoding are highlighted in red and green, respectively.

Configuration			Standard Training					With Retraining				
Encoder	Precision	D	Accuracy	AUROC	AUPR_IN	AUPR_OUT	FPR95	Accuracy	AUROC	AUPR_IN	AUPR_OUT	FPR95
Base-level	Float32	1000	0.79	0.85	0.76	0.91	0.26	0.90	0.92	0.88	0.95	0.18
		5000	0.83	0.86	0.77	0.91	0.25	0.93	0.92	0.87	0.95	0.17
		10000	0.83	0.86	0.77	0.91	0.24	0.94	0.92	0.88	0.95	0.18
	Binary	1000	0.71	0.87	0.79	0.92	0.23	0.75	0.91	0.84	0.94	0.20
		5000	0.79	0.86	0.76	0.91	0.23	0.85	0.93	0.89	0.95	0.18
		10000	0.81	0.86	0.77	0.92	0.23	0.88	0.92	0.87	0.95	0.18
Random projection	Float32	1000	0.83	0.88	0.87	0.89	0.42	0.91	0.88	0.86	0.88	0.50
		5000	0.84	0.89	0.87	0.90	0.38	0.94	0.95	0.92	0.96	0.16
		10000	0.84	0.89	0.87	0.90	0.38	0.95	0.96	0.94	0.96	0.13
	Binary	1000	0.79	0.92	0.90	0.92	0.33	0.79	0.91	0.89	0.91	0.35
		5000	0.82	0.93	0.91	0.94	0.21	0.81	0.93	0.91	0.94	0.21
		10000	0.82	0.94	0.91	0.94	0.20	0.82	0.93	0.91	0.94	0.21

Table 6.3.: Comparison of OOD detection in binary HDC with related methods on a binary multi-layer perceptron.

	OOD Detection in base-level		OOD Detection in random projection		MaxLogit [25]		MaxSoftmax [24]		Entropy [26]		EnergyBased [27]		ODIN [22]	
Dataset	AUROC	FPR95	AUROC	FPR95	AUROC	FPR95	AUROC	FPR95	AUROC	FPR95	AUROC	FPR95	AUROC	FPR95
CIFAR10	1.00	0.00	0.94	0.30	0.86	0.55	0.86	0.56	0.85	0.55	0.61	0.90	0.86	0.65
FashionMNIST	0.97	0.15	0.99	0.00	0.85	0.56	0.86	0.63	0.85	0.56	0.60	0.91	0.85	0.57
MNIST_C	0.75	0.56	0.76	0.69	0.73	0.87	0.73	0.90	0.72	0.87	0.55	0.92	0.73	0.87
Noise	1.00	0.00	0.99	0.00	0.89	0.41	0.89	0.43	0.88	0.38	0.62	0.88	0.96	0.38
Average	0.93	0.18	0.92	0.25	0.83	0.60	0.83	0.63	0.83	0.59	0.83	0.90	0.85	0.62

MLP, it shows greater potential for correctly recognizing OOD samples. In particular, for FashionMNIST, HDC reports 0% false positives with 95% true positive rate, compared to an average of 68% for MLP.

6.3.2. Hardware Implementation

Our OOD detection method is simple enough to be integrated into an MNIST HDC model with $D = 2000$ on PYNQ-Z2 as shown in Figure 6.3. Table 6.4 shows the hardware consumption of our method compared to the original HDC implementation explained in chapter 3 with base-level encoding, $D = 2000$, binary, and standard training, achieving 72.66% accuracy on hardware. The data, captured from Vivado 2022.2 for the HDC IP and OOD detection modules, demonstrate the negligible hardware consumption (2 LUT, 1 Register) and the negligible power consumption of our concurrent OOD detection. There is no additional latency, as the OOD flag will be issued with the classification output.

Table 6.4.: resource utilization of the OOD and HDC IP on PYNQ-Z2

Module	LUT	Registers	BRAM	Power
HDC	25651	61501	84	294 mW
OOD Detection	2(0.003%)	1(0.0005%)	0	<1 mW

Table 6.5.: OOD detection rate on Hardware HDC accelerator for MNIST

Detection rate	CIFAR10	MNIST-C miss	Fashion MNIST	Med MNIST	noise
Software	99.71%	66.84%	100%	53.38%	100%
Hardware	99.81%	69.65%	98.73%	55.98%	100%

We verify our hardware by comparing it against the results in software and report our results in Table 6.5. Here, we set τ to accept 95% of the ID (MNIST) dataset samples. The report shows results closely matching the software results, as expected. Due to some mismatch between the computations in hardware and software, we find negligible difference in the detection rate, however it is in an acceptable range.

6.3.3. Discussion

Our analysis demonstrates the robustness and generalizability of the proposed OOD detection technique in HDC across various encoding methods, quantization types, and dimension sizes. For this we inspected the accuracy of our method with three well-known image classification datasets. Key findings with respect to FPR95 include:

- Base-level encoding shows more stable OOD detection, while random projection shows higher variance.
- Retraining [32] has the greatest impact on detection rates.
- Larger dimensions generally improve detection efficiency.
- Binarization enhances OOD detection in one-shot training with random projection.
- HDC is naturally more capable of detecting OOD inputs than a Binary neural network (BNN) trained for the same task.
- While model accuracy is not strongly correlated with OOD detection, higher accuracy slightly improves it.

6.4. Conclusion

There is a crucial requirement to reduce the uncertainty of HDC models used in safety-critical applications by eliminating overconfident predictions when they are faced with OOD. In this work, we proposed a lightweight, hardware-friendly, concurrent OOD detection scheme with high coverage, suitable for critical edge devices. It is robust across various HDC encoding, quantization, dimension size and similarity functions. It uses the minimum distance of the encoded input data with the CHVs as a key score for OOD recognition in HDC, which requires only one comparator with a fixed value for the recognition

process. Since the calculation of this score is one of the operations during classification, no additional hardware, time, or retraining is required. Compared to existing methods, our method shows a significant improvement in detection. The implementation is also simple and easy to handle, as only 2 LUTs and 1 register are required. This method leads to effective concurrent identification of OOD inputs with negligible hardware overhead, which makes it suitable for all edge devices. In summary, our OOD detection shows promising results, which enhance functional safety in HDC classifiers.

7. Runtime Uncertainty Detection in HDC

This chapter presents a lightweight runtime monitoring framework for detecting unreliable HDC predictions. It addresses uncertainty caused by both input disturbances and hardware-induced faults. Instead of adding complex checking logic, the proposed method uses similarity-based indicators available during normal HDC inference to identify unsafe or untrusted outputs. The resulting framework provides runtime reliability monitoring with negligible hardware overhead and no additional latency [8].

7.1. Motivation and Problem Definition

The previous chapter showed that HDC accelerators can detect OOD inputs using an extremely lightweight mechanism based on the maximum similarity score. This demonstrated an important principle: the similarity scores already produced during HDC inference contain information not only about the predicted class, but also about the trustworthiness of the prediction. When an input does not align well with any stored CHV, the output label should not be accepted blindly.

However, real-world uncertainty extends beyond OOD inputs. In practical edge deployments, unreliable decisions can result from noisy sensor data, corrupted inputs, distribution shifts, memory bit flips, faulty encoder parameters, or defects in the datapath used to generate and compare hypervectors. These disturbances may originate outside the accelerator, within the memory subsystem, or inside the accelerator's logic. From the perspective of the final classifier, however, they have a common effect: they disrupt the alignment between the QHV and the stored CHVs, leading to abnormal similarity score behavior.

This motivates a broader runtime reliability framework. Rather than building a separate detector for each possible source of uncertainty, this chapter investigates whether the similarity information already produced by HDC inference can serve as a unified indicator of unreliable operation. This is especially important for edge accelerators, where full redundancy, extensive correction, and multiple monitoring layers are often too expensive.

The objective is not to react to every bit flip or minor perturbation. HDC can naturally tolerate many small disturbances, and detecting them would only waste hardware resources if the final prediction remains unchanged. The relevant question is whether the disturbance is large enough to affect the decision. The proposed method therefore monitors similarity-score degradation and flags only reliability-relevant anomalies. After detection, the system may reject the output, repeat the inference, enable a stronger protection mode, or switch to a fail-safe response.

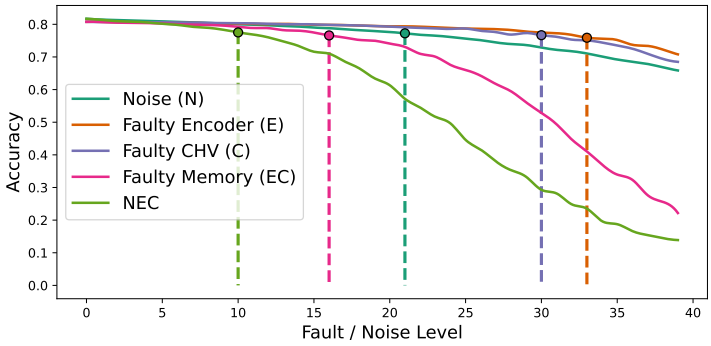


Figure 7.1.: Accuracy degradation under various fault types and uncertainty levels for the MNIST dataset, using a random projection encoder with a dimensionality of 10K. The scenarios include CHV faults (C), Encoder faults (E), combined Encoder and CHV faults (EC), and combined Noise, Encoder, and CHV faults (NEC). The vertical lines indicate the points where the accuracy decreases by 5% for each metric.

This chapter therefore extends similarity-based monitoring from OOD detection to general runtime uncertainty detection. The proposed framework treats unreliable inference as a system-level condition that may arise from either input anomalies or hardware faults. It monitors the classification similarity score as a reliability indicator and flags abnormal behavior when the score deviates from the expected in-distribution operating range. In this way, the same lightweight principle used for OOD detection is generalized to cover noisy inputs, encoder faults, CHV memory faults, combined memory faults, and other hardware-induced anomalies.

7.2. Reliability Monitoring Framework for Hyperdimensional Computing

The proposed method builds on the observation that certain runtime features of the similarity computation, such as the maximum value of class similarity scores, reflect the internal reliability of HDC inference. Deviations from their expected distributions can indicate unreliable operation caused by OOD data, data corruption or hardware faults. In this work, we first systematically evaluate various similarity-based metrics to identify which correlates best with such deviations for uncertainty detection, and then conduct a thorough evaluation of the proposed method using the best metric.

7.2.1. Fault Detection Methodology

7.2.1.1. Working Hypothesis

During HDC training, each CHV aggregates the high-dimensional representations of all samples in its class, forming a prototype that captures the dominant statistical patterns. Under normal conditions, new inputs yield QHVs that align closely with their true CHV, producing a distinct, high-similarity cluster in the similarity score distribution. In contrast, memory faults, hardware degradation, input corruption, or distribution shifts disturb this balance. Figure 7.1 shows the sensitivity of a model (trained on MNIST) to different fault scenarios, including encoder memory fault (E), CHV memory fault (C), the combination

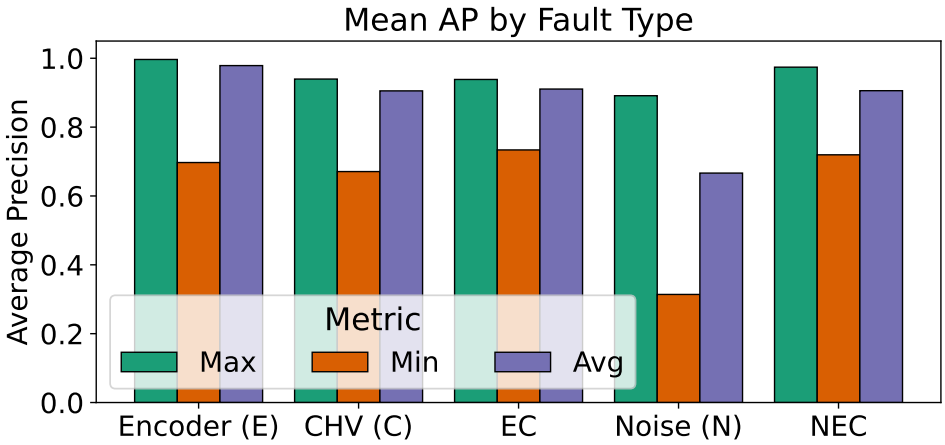


Figure 7.2.: Mean fault detection precision averaged over all datasets for different metrics at a 5% accuracy drop. Results are reported under five uncertainty scenarios: Noise (N), Encoder faults (E), CHV faults (C), combined Encoder and CHV faults (EC), and combined Noise, Encoder, and CHV faults (NEC).

of these two (memory fault or EC), noise in the input data (N), and the combination of all scenarios together.

The resulting QHVs deviate from the subspace defined by valid training representations and become less aligned with any stored CHVs. Since random or mismatched hypervectors are almost orthogonal in high-dimensional space, these degraded conditions should lead to noticeably lower similarity values. Following this assumption, we analyze how precisely an abnormal state can be detected by using either the maximum similarity, minimum similarity, or average similarity score. Figure 7.2 shows the average precision for these detection metrics over multiple datasets for a 5% accuracy drop. Evidently, the detection of outliers functions best when considering the maximum similarity score.

7.2.1.2. Origins of the Score Shift

Any degradation mechanism that weakens the alignment between QHV and CHV directly lowers the similarity score achieved by *Argmax* in HDC. In practice, several types of faults can cause this degradation and must be detected to maintain reliable operation:

- *Noisy or OOD inputs:* Inputs that the network has not been trained for, or those heavily corrupted by noise, generate QHVs that do not correctly align with any stored CHV. While a limited level of noise can be tolerated by HDC, excessive corruption that risks misclassification should trigger detection rather than a forced decision. This ensures that uncertain or abnormal inputs do not lead to unreliable outputs.

- *Transient faults in accelerator memory:* Errors in the memory that stores encoder parameters or CHVs can significantly distort representations. Bit flips or retention failures alter the learned prototypes and encoded queries, causing large similarity deviations. Detecting such faults is

essential, although small changes can often be tolerated, but if they lead to misclassification, they should be detected to prevent unreliable outputs.

- *Permanent hardware faults:* transient or permanent faults in the encoder or classifier hardware, such as stuck-at faults or timing violations, can corrupt QHV generation or similarity computation. These faults may appear randomly during operation and must be identified through built-in fault detection mechanisms.

Together, these effects alter the top similarity values, enabling detection of reliability degradation via simple thresholding.

7.2.2. Threshold-Based Reliability Detection

As discussed earlier, a high correlation between the QHV and the corresponding CHV indicates that the model is functioning correctly. Based on our evaluations, a properly functioning HDC model produces maximum similarity scores within a well-defined range. To identify unreliable or faulty conditions, a detection threshold (τ) is established from the distribution of similarity scores obtained from the training dataset (Figure 7.3). In practical deployments, dataset shift may introduce deviations between the training and runtime data. However, under the standard model, the test data is drawn from the same underlying distribution as the training set, indicating that the training data provides a sufficient basis for estimating the similarity score distribution and deriving the threshold τ. Any input producing a score below τ is flagged as unreliable.

Similar to chapter 6, the threshold τ can be adjusted based on the sensitivity requirements of the target application, as it directly affects the detection rate of abnormal or unreliable conditions that will be discussed later. For a default preliminary example, τ is set such that 95% of the ID are accepted. However, τ can be adjusted based on the evaluation metrics and scores under simulated faults. Given the sorted similarity scores X obtained from ID samples, the threshold is computed as:

$$r = \frac{100 - p}{100} \times (n - 1), \tag{7.1}$$

$$\tau = (1 - g) \times X_j + g \times X_{j+1}, \tag{7.2}$$

where p is the acceptance rate (default 95%), n is the number of samples, $j = \lfloor r \rfloor$, and $g = r - j$.

At inference time, the decision function is:

$$f(x, \tau) = \begin{cases} \text{accept}, & x \geq \tau \\ \text{flag unreliable}, & x < \tau \end{cases} \tag{7.3}$$

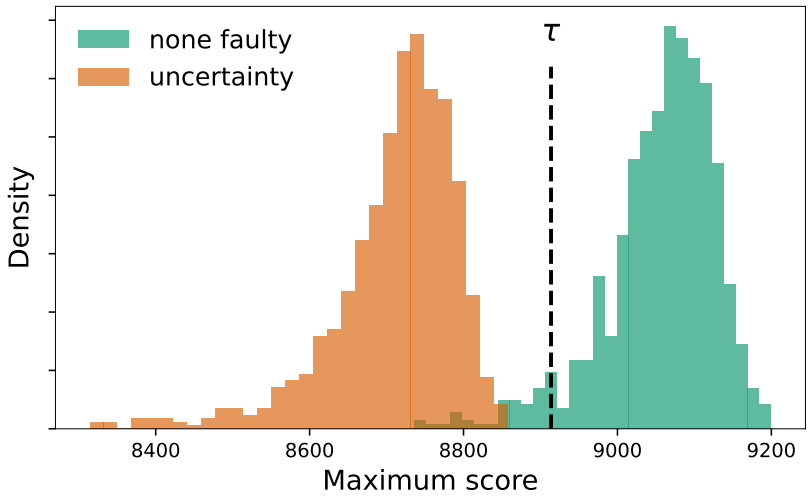


Figure 7.3.: Adjusting τ based on the in-distribution similarity score distribution.

The threshold can be dynamically adjusted to trade off between sensitivity and false alarms. Increasing τ improves safety and conservatism but may lead to more false positives. Once the reliability flag is raised, the system may trigger auxiliary checks or fallback procedures, such as redundant inference, retraining, or human intervention.

Figure 7.4 shows the behavior of similarity score distributions for MNIST dataset (as baseline) under various fault and uncertainty conditions. These conditions include faulty encoder parameter memory, faulty CHV memory, simultaneous faults in both the encoder parameters and CHV memory, hardware stuck-at faults, noisy input data, and OOD inputs. Stuck-at faults can be modeled as a persistent type of memory fault because they fix signal values at constant logic levels. In the encoder datapath, these faults propagate to the generated QHV as consistent bit-level distortions, equivalent to permanent bit-flips in specific dimensions. Due to their deterministic nature, they produce consistent deviations in similarity scores and are therefore implicitly captured in high fault-rate scenarios. This justifies modeling stuck-at faults as an extreme case of memory faults. Consequently, the resulting similarity score distributions under stuck-at faults exhibit even more distinct separation from normal operation than the cases shown here. For brevity, these extreme-case distributions are not included in Figure 7.4.

7.2.3. Hardware Implementation

The proposed reliability detection mechanism can be seamlessly integrated into hardware with minimal overhead. It requires only a single comparator at the classifier output to evaluate the maximum similarity score against the predefined threshold (τ). This design allows parallel reliability monitoring alongside normal inference, without affecting the system's throughput, latency, or energy efficiency. Figure 7.4 shows the hardware-level integration of the detection unit along with the HDC Intellectual Property (IP).

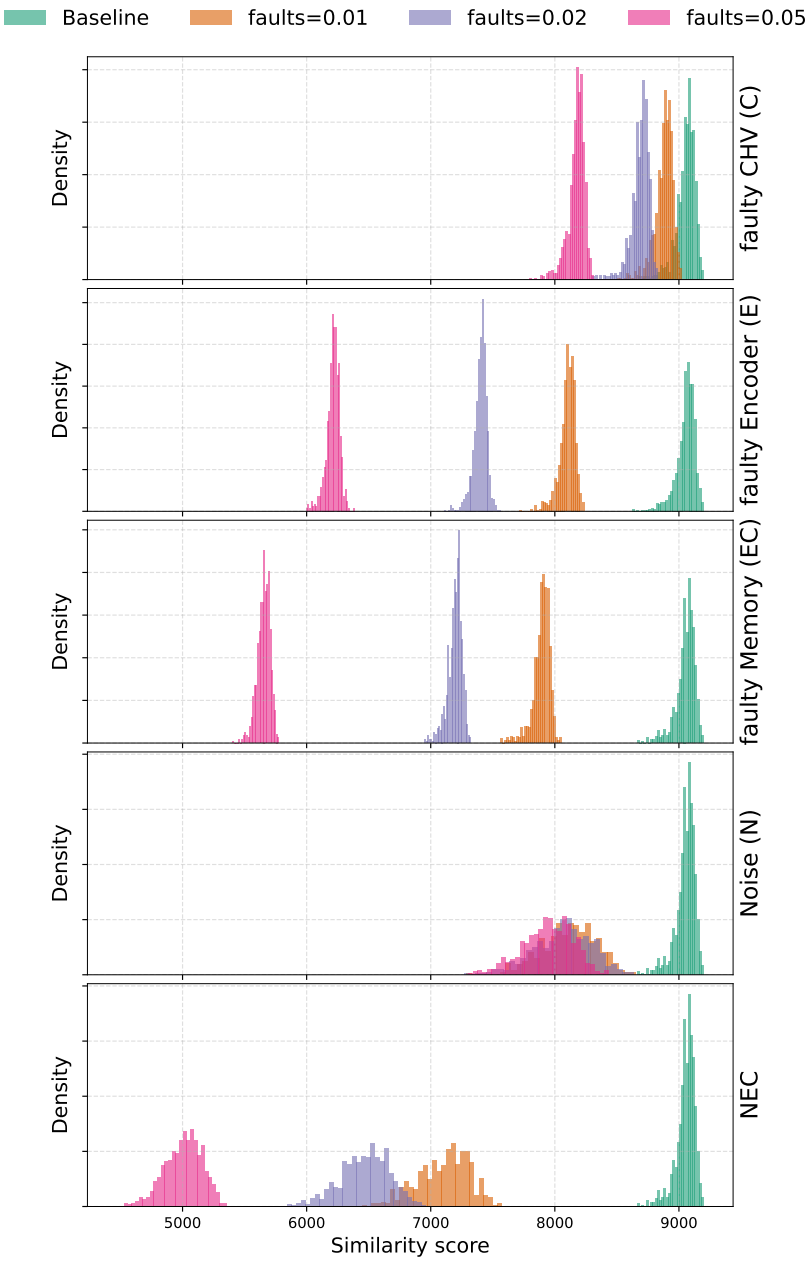


Figure 7.4.: Representative histograms of similarity score distributions for baseline (fault-free) data and faulty models. Results are shown under five uncertainty scenarios: Noise (N), Encoder faults (E), CHV faults (C), combined Encoder and CHV faults (EC), and combined Noise, Encoder, and CHV faults (NEC).

7.2.4. Evaluation Metrics

The effectiveness of the proposed reliability detection mechanism is assessed using Precision-Recall (PR) analysis, which provides a more informative characterization of classifier performance under class imbalance [82]. Unlike ROC curves, which can remain overly optimistic when the negative class dominates, the PR curve directly reflects the trade-off between *precision* (the proportion of correctly

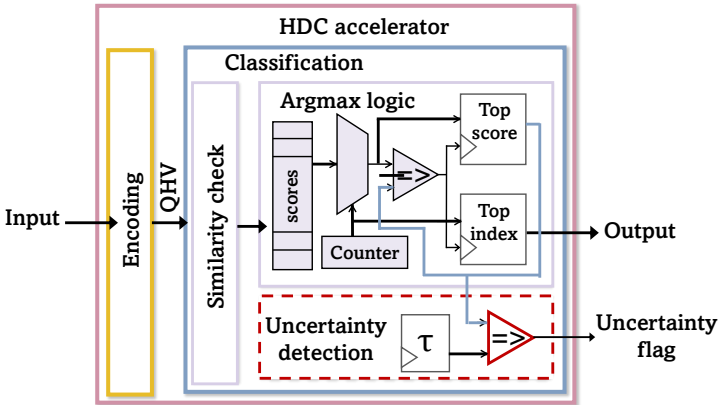


Figure 7.5.: Hardware overview of the proposed reliability detection mechanism in HDC. The adaptive threshold τ determines whether the output is considered reliable.

detected unreliable samples among all samples flagged as unreliable) and *recall* (the proportion of all unreliable samples successfully detected) as the detection threshold (τ) varies.

Two key metrics are reported:

- **AP:** The *Average Precision*, corresponding to the area under the PR curve. It summarizes the model's overall ability to rank unreliable samples ahead of reliable ones.
- **F1 Score:** The harmonic mean of precision and recall, defined as

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{7.4}$$

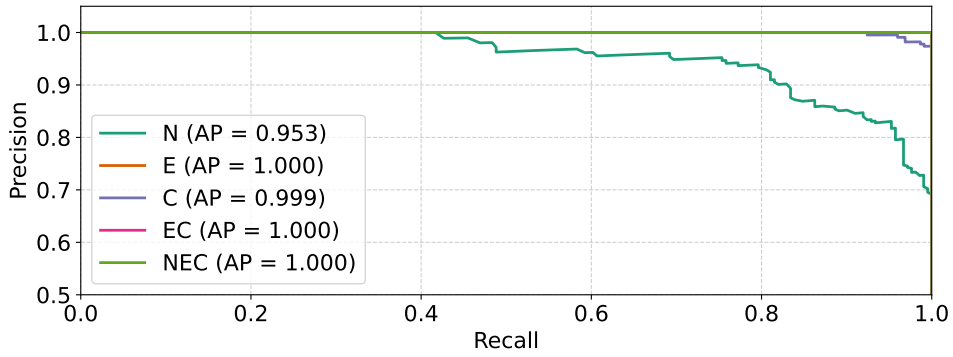
  It quantifies the balance between correctly identifying unreliable samples and avoiding false alarms at a specific threshold.

Higher AP and F1 values indicate stronger discrimination between normal and abnormal inferences, while a substantial margin above the baseline precision demonstrates the method's robustness under class imbalance. In practice, F1 scores above 0.8 already indicate strong separation between reliable and unreliable inferences, while values approaching 0.9 or higher correspond to near-optimal discrimination. An example for PR curve is presented in Figure 7.6. Here, the detection mechanisms achieves high AP of at least 95% for base-Level encoding and 94% for random projection.

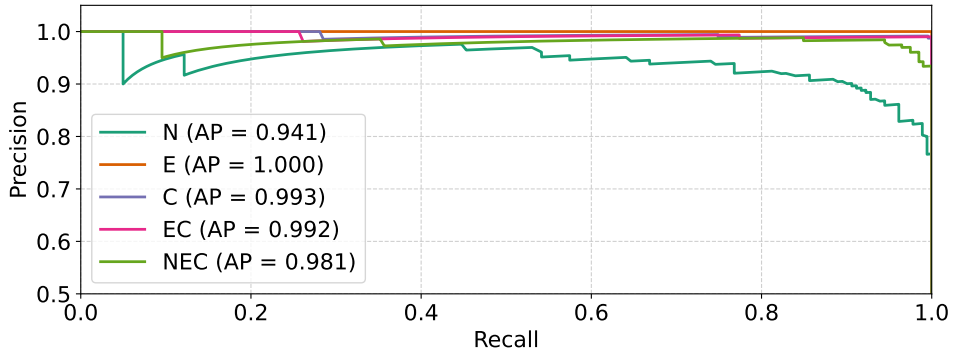
For OOD detection with balanced datasets we also report the false-positive rate when the true-positive rate is fixed at 95% (FPR-95), indicating how often normal inputs are incorrectly flagged as unreliable.

7.2.5. Reliability Under Parameter Variations

To evaluate the generalizability and robustness of the proposed reliability detection mechanism, we analyze its behavior across a wide range of HDC design parameters and datasets. Specifically, we vary the encoding technique and the hypervector dimension size, as both significantly influence the representation quality and system efficiency. The choice of encoding technique directly affects the structure and



(a) Base-level



(b) Random projection

Figure 7.6.: Precision–recall (PR) curves for the Isolet dataset at a 5% accuracy drop, using a dimensionality of 10,000. Results are shown for CHV faults (C), Encoder faults (E), combined Encoder and CHV faults (EC), and combined Noise, Encoder, and CHV faults (NEC), where the curves largely overlap.

separability of the hypervectors, thereby influencing both classification accuracy and similarity score distribution. Similarly, the hypervector dimension size determines the level of representational richness and impacts resource usage, latency, and accuracy.

In addition, we compare *one-shot* and *online* training methods [32], as they yield different inter-class separability and similarity score concentration characteristics. Various noise models, synthetic corruptions, and cross-dataset inputs are also employed to evaluate robustness under realistic reliability stress conditions.

These parameters have the most significant influence on the detection rate, as discussed in chapter 6. However, other factors, such as the numerical precision (i.e., the number of bits) and the choice of similarity function, also analyzed in chapter 6, were found to have a negligible impact on the detection performance. Therefore, these aspects are excluded from the scope of this chapter, and we focus our evaluation on the binary version of HDC employing the Hamming distance function within the *Argmax* operation.

As shown in Table 7.1 multiple HDC configurations are trained using combinations of encoding schemes, dimension sizes, precision levels, and similarity functions. Base-level and random projection

Table 7.1.: Summary of Configuration Factors

Configuration Factor	Values Evaluated
Encoding	Base-level, Random projection
Dimension Sizes	1000, 10000
Training Procedure	One-shot, OnlineHD [32]
Dataset	MNIST, FashionMNIST, Isolet, UCI-HAR

encodings are both evaluated, with dimension sizes of 1000, and 10000, using Hamming, dot, and cosine similarities. Each configuration is trained with one-shot and online learning strategies.

The evaluation covers four benchmark datasets spanning applications including image classification, voice recognition, and wearable sensor analysis. MNIST and Fashion-MNIST are standard benchmarks for grayscale image classification. Isolet represents a speech recognition task based on spoken letter audio features. Finally, UCI-HAR includes wearable sensor readings from accelerometers and gyroscopes for human activity recognition. These datasets collectively cover a diverse range of input domains, enabling evaluation of the reliability detection mechanism across different data distributions and encoding strategies.

Each dataset is tested under various reliability stress conditions, including additive Gaussian noise, OOD inference, and random bit perturbations applied to both the encoder parameter hypervectors and the stored CHV memory.

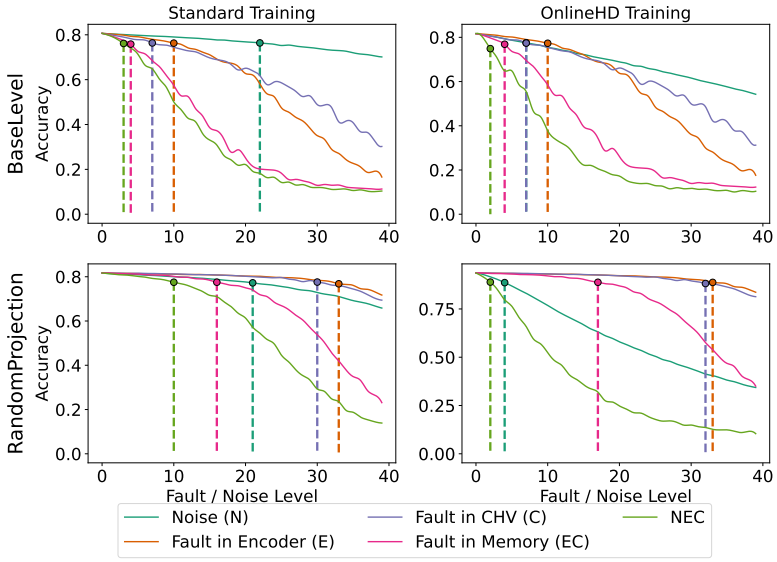
7.3. Experiments and Implementation Results

In this section, we first present the results of uncertainty detection on different datasets, analyze the influence of HDC parameters on detection efficiency, and compare our approach with state-of-the-art methods. Finally, we report the resource utilization on the PYNQ-Z2 board using the MNIST dataset as the ID benchmark.

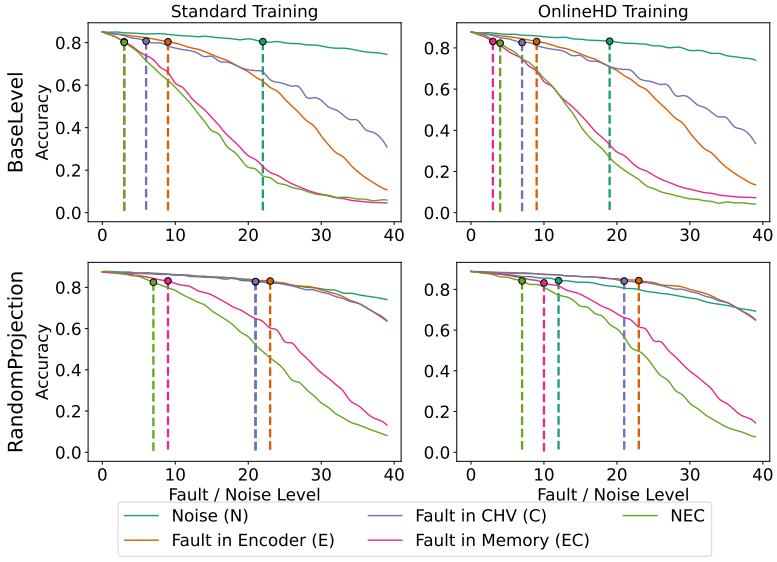
7.3.1. Simulation-Based Detection Results

For a reference of how different sources of uncertainty affect the classifier, we analyze the prediction accuracy under varying noise/fault levels and report them in Figure 7.7. We notice that, for the majority of models, faults in the encoder have the strongest impact (aside from NEC, which includes the encoder). In the remaining experiments we adjust the fault level, so that the accuracy drops by 5%.

Results for all experiments are presented in Table 7.2. Models with dimension sizes of 1000 and 10000 with standard training, as well as 10000 with onlineHD training are explored. When comparing results per dataset, it shows that a higher dimension size results in better AP and F1 scores. This is because in a larger hyperdimensional space, variations in the QHVs will result in higher differences in the similarity score. For most classification tasks, a dimension size of 10000 achieves sufficient accuracy, thus, our



(a) MNIST



(b) ISOLET

Figure 7.7.: Accuracy drop for classifiers trained on MNIST and ISOLET, configurations and types of uncertainty. Dashed lines denote the fault/noise level resulting in a drop of 5%.

method fits real-world applications. Furthermore, we find that models with base-level encoding produce more distinguishable similarity scores than the ones with random projection. Figure 7.8 shows that the distribution for the random projection encoding is much wider than the Base-level, which is reflected in the detection accuracy.

Considering onlineHD models, while the prediction accuracy increases, the CHVs themselves become more correlated and less orthogonal, reducing the relative distance between in-distribution and faulty samples; therefore, the similarity scores across classes and noise conditions converge, making faults and OOD inputs harder to distinguish.

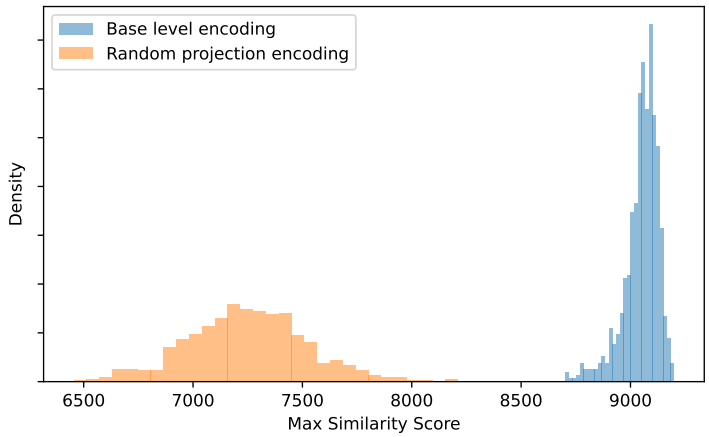


Figure 7.8.: Distributions of maximum similarity scores for HDC models trained on MNIST with base-level and random projection encoding.

Table 7.2.: Results for AP and F1 score at 95% threshold for fault detection over different datasets with various HDC configurations.

Encoder	Dataset	Dimension	Model Accuracy	Noise (N) AP, F1	Faulty Encoder (E) AP, F1	Faulty CHV (C) AP, F1	Faulty Memory (EC) AP, F1	NEC AP, F1
Base-level	MNIST	1k	70.47%	79%, 0.43	100%, 0.98	74%, 0.38	100%, 0.98	99%, 0.94
		10k	80.64%	100%, 0.98	100%, 0.98	100%, 0.97	100%, 0.98	100%, 0.98
		onlineHD	81.67%	84%, 0.51	100%, 0.98	100%, 0.98	100%, 0.98	100%, 0.98
	Isolet	1k	70.69%	74%, 0.38	100%, 0.97	74%, 0.38	100%, 0.97	100%, 0.97
		10k	83.58%	99%, 0.97	100%, 0.98	100%, 0.98	100%, 0.98	100%, 0.98
		onlineHD	87.81%	97%, 0.91	100%, 0.97	100%, 0.97	100%, 0.97	100%, 0.97
	Fashion MNIST	1k	65.10%	95%, 0.87	100%, 0.97	95%, 0.85	100%, 0.97	100%, 0.97
		10k	67.92%	99%, 0.98	100%, 0.98	100%, 0.98	100%, 0.98	100%, 0.98
		onlineHD	67.51%	76%, 0.25	100%, 0.98	78%, 0.32	100%, 0.98	100%, 0.98
	UCIHAR	1k	74.24%	79%, 0.41	100%, 0.98	89%, 0.70	100%, 0.98	100%, 0.98
		10k	82.63%	99%, 0.98	100%, 0.98	92%, 0.66	100%, 0.98	100%, 0.98
		onlineHD	87.61%	95%, 0.91	100%, 0.98	99%, 0.98	100%, 0.98	100%, 0.98
Random Projection	MNIST	1k	78.86%	80%, 0.41	90%, 0.61	86%, 0.55	86%, 0.51	83%, 0.46
		10k	81.69%	91%, 0.69	100%, 0.98	100%, 0.98	100%, 0.98	100%, 0.98
		onlineHD	93.62%	69%, 0.30	88%, 0.63	96%, 0.87	81%, 0.46	84%, 0.53
	Isolet	1k	82.36%	87%, 0.61	84%, 0.49	73%, 0.30	85%, 0.51	81%, 0.44
		10k	86.72%	99%, 0.97	100%, 0.97	100%, 0.97	100%, 0.97	100%, 0.97
		onlineHD	88.83%	96%, 0.89	100%, 0.97	100%, 0.97	100%, 0.97	100%, 0.97
	Fashion MNIST	1k	65.57%	81%, 0.38	94%, 0.94	88%, 0.54	98%, 0.97	88%, 0.55
		10k	66.55%	85%, 0.46	100%, 0.98	100%, 0.98	100%, 0.98	100%, 0.98
		onlineHD	75.18%	63%, 0.18	72%, 0.21	74%, 0.23	66%, 0.15	65%, 0.16
	UCIHAR	1k	81.37%	82%, 0.36	73%, 0.15	74%, 0.23	76%, 0.15	92%, 0.62
		10k	85.71%	97%, 0.89	89%, 0.36	97%, 0.96	100%, 0.99	97%, 0.98
		onlineHD	92.77%	77%, 0.27	93%, 0.69	87%, 0.44	76%, 0.16	95%, 0.91
Average		1k	–	82.0%, 0.48	92.5%, 0.76	81.8%, 0.49	93.1%, 0.76	92.9%, 0.74
		10k	–	96.2%, 0.86	98.6%, 0.90	98.7%, 0.93	99.9%, 0.98	99.6%, 0.98
		onlineHD	–	82.2%, 0.53	94.2%, 0.80	91.7%, 0.72	90.3%, 0.71	92.9%, 0.81

Nonetheless, our proposed method reaches an average precision of over 89% over all models with 10000 dimension size and standard training. For all but one scenario (noise), the F1 score is above 0.95.

Although the classification accuracy of HDC is 10% lower than that of MLP, it shows greater potential for correctly recognizing OOD samples. In particular, for FashionMNIST, HDC reports 0% false positives with 95% true positive rate, compared to an average of 68% for MLP.

Table 7.3.: resource utilization of the *SafeHDC* added extension and HDC IP for dimensionality of $2K$ on PYNQ-Z2.

Module	LUT	Registers	BRAM	Power
HDC	25651	61501	84	294 mW
SafeHDC	2(0.003%)	1(0.0005%)	0	<1 mW

7.3.2. Hardware Implementation

The proposed detection method is simple enough to be integrated into an MNIST HDC model with $D = 2000$ on PYNQ-Z2 as shown in Figure 7.5. Table 7.3 shows the hardware consumption of our method compared to our HDC implementation RCEHDC with base-level encoding, $D = 2000$, binary, and standard training, achieving 72.66% accuracy on hardware. The data captured from Vivado 2022.2 for the HDC IP and detection modules demonstrate the negligible hardware consumption (2 LUTs, 1 register) and negligible power consumption of our concurrent uncertainty detection mechanism. There is no additional latency, as the flag will be issued with the classification output.

7.3.3. Discussion

Our analysis highlights the robustness and generalizability of the proposed reliability and OOD detection mechanism across diverse HDC configurations. As shown earlier, the method consistently maintains high detection accuracy for all fault types and datasets, regardless of the encoding approach, training scheme, or hypervector dimensionality.

Across all configurations, the average detection metrics remain strong, achieving an AP of up to 99.6% and a F1 score of 1.00 for large-dimensional models ($D = 10$k). These results demonstrate that the proposed detection method reliably identifies both hardware-induced faults and uncertain inputs while preserving the inherent efficiency of HDC.

Several observations can be drawn from the experiments:

- **Impact of dimensionality:** Increasing the hypervector dimension consistently improves detection performance. Models with $D = 10$k outperform those with $D = 1$k by an average of 30% in AP, highlighting the benefit of richer high-dimensional representations for reliability estimation.
- **Encoding method:** Base-level encoding provides more stable detection across datasets, while random projection shows higher variance but achieves comparable accuracy at larger dimensions.
- **Effect of retraining:** OnlineHD [32] improves classification accuracy but does not yield the highest fault- or noise-detection rates. The primary factor is the distribution of similarity scores. OnlineHD results in higher variance and increased overlap between in-distribution and faulty samples, reducing separability and making threshold-based detection more challenging. This also leads to a lower effective margin for a fixed threshold τ. Its average AP (90.3%) and F1 (0.72)

remain below those of the larger one-shot models, suggesting that retraining mainly enhances classification rather than reliability separation.

- **Fault-type sensitivity:** The proposed detector achieves near-perfect identification of memory-related faults, with AP and F1 values close to 1.00 across all configurations, demonstrating its effectiveness in capturing both encoding and classification anomalies.
- **Model accuracy vs. detection:** While model accuracy does not directly correlate with fault detection performance, higher-accuracy configurations tend to yield slightly better reliability discrimination.
- **Choosing τ:** Detection performance remains stable across a range of threshold values (e.g., 95%–100% ID acceptance), indicating low sensitivity to precise tuning of τ. While SafeHDC remains stable under nominal conditions, τ can be adapted at runtime using lightweight monitoring, such as uncertainty flags.

Overall, these findings show that the proposed method effectively detects runtime faults and uncertain conditions with negligible hardware cost. The method performs best in configurations with well-separated similarity distributions, such as base-level encoding and higher dimensionality, while performance degrades in cases with increased overlap, such as random projection with OnlineHD. This makes it suitable for edge-deployed HDC systems where functional safety and low overhead are critical. Additionally, as the next step, τ can be adapted at runtime using lightweight similarity monitoring, such as uncertainty flags, without retraining, to maintain performance under changing conditions.

7.4. Conclusion

Ensuring reliability and safety in HDC-based systems requires mechanisms that can effectively detect uncertain and faulty conditions, including those arising from noisy or OOD inputs and hardware faults. In this work, we introduced an ultra-lightweight, concurrent reliability detection mechanism that enhances the functional safety of HDC accelerators. The proposed scheme consistently identifies anomalies within the encoding and classification processes without retraining or added computational latency. It leverages the maximum similarity score between encoded inputs and CHVs as a reliability score, implemented with only one comparator alongside minimal hardware resources (2 LUTs and 1 register). Experimental results validate that the approach achieves high detection coverage (on average 94.62%) and robustness across different encoding and dimensionality schemes, ensuring effective runtime monitoring for edge-deployed HDC models. This work thus provides a practical solution for real-time uncertainty and fault detection, enabling dependable operation of HDC accelerators in safety-critical applications.

8. CHV Fault Detection in HDC

This chapter discusses concurrent error detection for faults in CHV memory. Because CHVs are used directly during associative search, faults in this memory can alter similarity scores and cause incorrect predictions. The proposed CED-HDC method monitors classification behavior during inference and detects faulty outputs before the system relies on them. This offers a lightweight safety mechanism for HDC accelerators with minimal hardware overhead [6].

8.1. Motivation and Problem Definition

The previous chapter proposed a broad lightweight runtime detector for HDC accelerators. The detector monitored the maximum similarity score and showed effective detection for several uncertainty sources, including noisy inputs, OOD samples, memory faults, and different hardware-induced anomalies.

However, CHV memory faults remained the most challenging case. Unlike other sources of uncertainty, these faults do not always cause a clear drop in the maximum similarity score. Instead, the similarity score distribution under CHV faults can overlap with the distribution observed during fault-free inference. This overlap weakens threshold-based detection and makes CHV faults harder to identify using a general runtime monitor.

This issue is important because, in the implemented HDC accelerator, the classifier is the most vulnerable part of the hardware, especially the CHV memory used directly in similarity computation. Therefore, faults in this memory can directly disrupt the similarity comparison and may alter the final prediction.

Therefore, this chapter studies fault detection in CHV memory in more detail. The objective is to understand how faults in the classifier memory affect similarity-score behavior and to design a detector better suited to this fault source. Rather than relying solely on the broad detector from the previous chapter, this chapter introduces a lightweight, dedicated detection mechanism for the CHV memory section.

The proposed detector follows the same hardware-aware philosophy as the rest of the thesis. It avoids expensive redundancy, full recomputation, or heavy correction logic, and instead uses information already produced during HDC classification. In this way, the chapter targets the weakest detection case observed previously while keeping the detector suitable for resource-constrained edge accelerators.

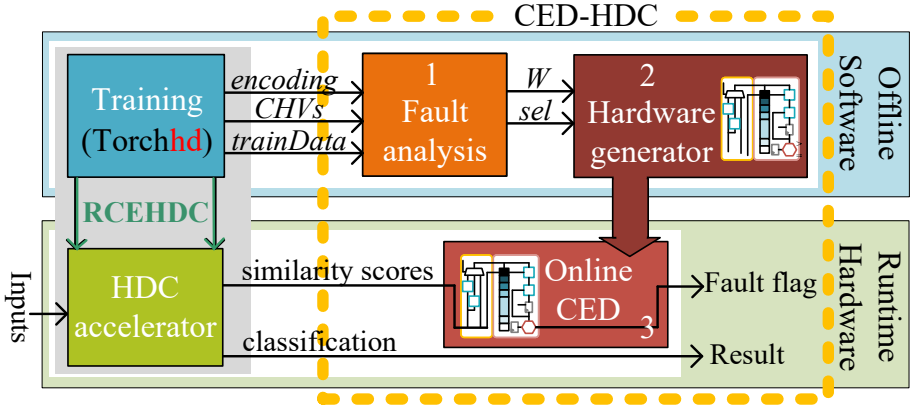


Figure 8.1.: Detection mechanism flow: 1. Fault analysis, 2. Generate CED hardware based on configuration, 3. Employ CED hardware in HDC accelerator. W denotes the monitoring window size for keeping track of the output scores, and *sel* selects the optimal fault detection method. Model training is performed using Torchhd [48], while the HDC-to-FPGA mapping is handled by RCEHDC.

8.2. Proposed CED Method

An overview of the proposed method is depicted in Figure 8.1. This work aims to provide a general CED technique that operates independently of dataset type, encoding scheme, dimensionality, and quantization. Therefore, we first perform a comprehensive offline fault analysis of each model individually. Based on the evaluation results, a monitoring window of size W is defined to adjust the fault coverage to user needs. Finally, the online testing module is generated and implemented alongside the HDC IP on the inference side.

To highlight the importance of conducting fault analysis for each model individually, Figure 8.2 shows the relationship between accuracy and fault probability in HDC across different datasets and encoding schemes. The results confirm that fault detection in HDC only becomes critical when the errors reach a certain level that causes a reduction in accuracy. Furthermore, the level of fault tolerance varies significantly depending on the dataset and encoding scheme, highlighting the need for tailored fault analysis to ensure optimal reliability. Following the fault analysis, the most suitable detection method is identified and subsequently configured for implementation in the hardware in the online testing module.

In the remainder of this section, we explore various HDC configurations, where the dimension size and quantization bit width are denoted by D and Q, respectively. The subsequent subsections provide an in-depth explanation of the fault analysis methodology and the hardware implementation of the CED scheme.

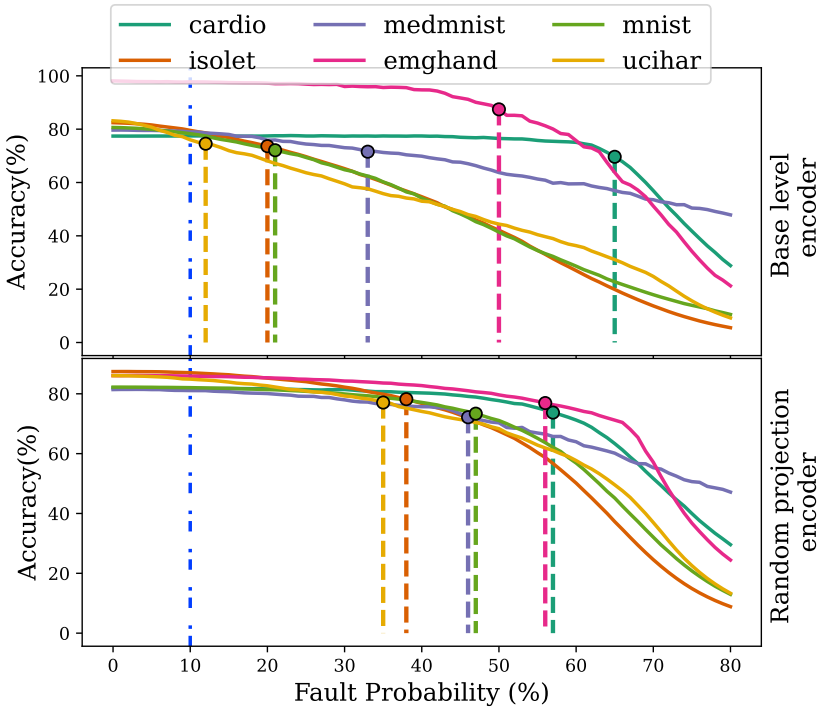


Figure 8.2.: Degradation of accuracy with different data sets and different error probabilities. Vertical lines mark error probabilities that cause a drop in accuracy of 10%. While practical ECC implementations fail to detect beyond 10% (vertical dashed blue line), the HDC models are robust and remain stable at those fault probabilities. Configuration: D=10000, and Q=1.

8.2.1. Offline Fault Analysis

Our proposed CED technique, instead of monitoring the memory data, monitors similarity scores extracted from the similarity check module within HDC and detects hardware faults by analyzing them. The similarity score for a given class c is computed as

$$similarity_score^c = \sum_{i=1}^{D} |QHV_i - CHV_i^c|$$

Any change in CHVs caused by faults affects their position in the hyperdimensional space, leading to corresponding changes in the similarity score. These variations in similarity scores serve as indicators to identify faults and determine their impact on system reliability.

To refine this approach, various metrics based on similarity scores are considered for fault detection, including:

- Maximum similarity score
- Sum of all similarity scores
- Average similarity score
- Distance between the highest and lowest scores (maximum distance between scores)

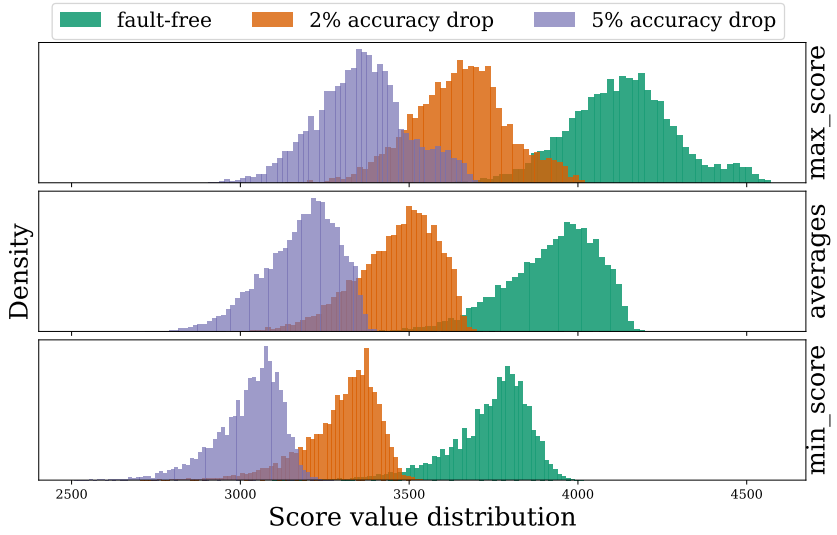


Figure 8.3.: Impact of accuracy drop on the distribution of detection scores. Since the average and summation metrics yield similar results, summation is excluded to avoid redundancy. Configuration: base-level encoder, with D=1000, Q=1 and MNIST dataset.

- Minimum similarity score

The goal is to identify a metric that can effectively distinguish between the base model without any bit-flips (fault-free) and faulty models, particularly when there is noticeable (e.g., user-defined) drop in accuracy. Initially, the range and lowest possible values of each metric are calculated for the fault-free model, serving as a baseline (τ). Metrics are then recalculated after incrementally injecting faults into the CHVs. Fault detection is only possible if the values of the metrics in the faulty model are outside the baseline range established for the fault-free model.

The effectiveness of each metric across different datasets and fault probabilities is analyzed, as shown in Figure 8.4. Fault coverage varies depending on the dataset and metric used. Among the evaluated metrics, maximum score, average, summation, and minimum score achieve the highest fault coverage, though their performance varies depending on the dataset.

Figure 8.3 presents the distribution of the top three metrics scores (maximum, average, and minimum scores) for fault-free model and models with accuracy drops of 2% and 5% on the MNIST dataset. As accuracy drops, the score distributions become more distinguishable, enabling more effective fault detection. However, overlaps in the histograms at a 2% accuracy drop indicate undetectable faults according to this method. The first point of 100% fault detection occurs at a 5% accuracy drop, for the minimum score metric. Consequently, in this model, the lowest detectable error probability without false positives is 5%.

To enhance configurability and address fault detection for lower accuracy drops, we introduce a windowing mechanism. Rather than relying on a single score, the decision is based on the average score of the detection method within a defined window size (W) for the last W inferences. Windowing improves fault detection by smoothing out noise and reducing the impact of outliers. Additionally, it enables

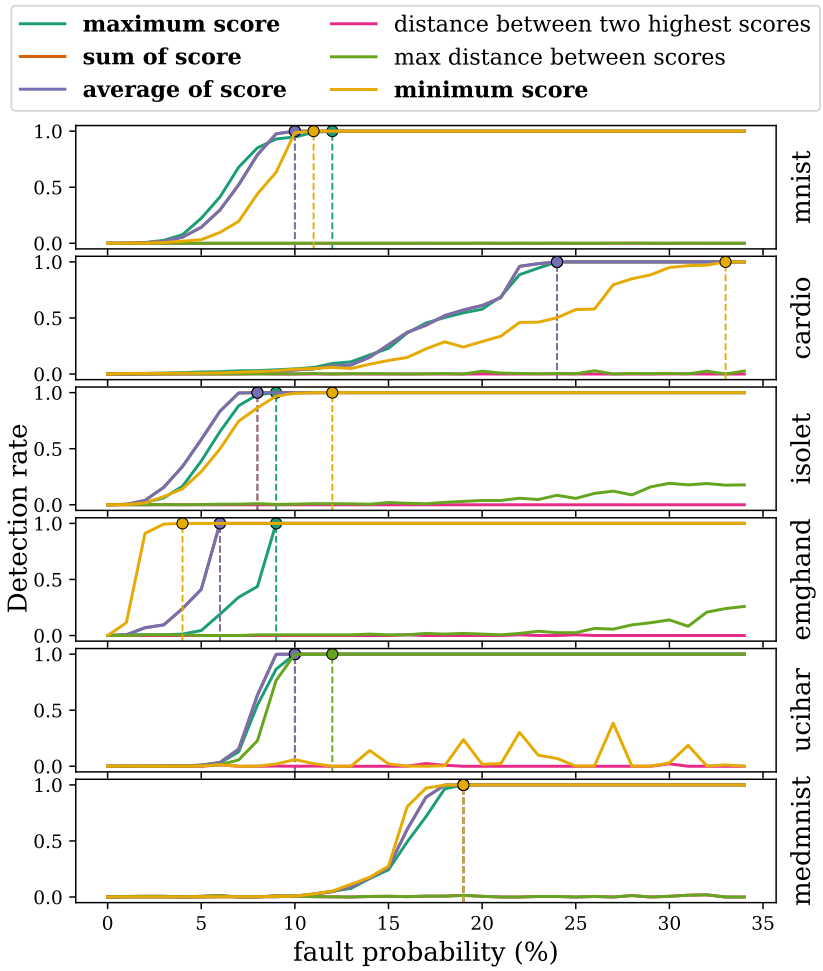


Figure 8.4.: Detection metrics under varying fault injection rates for various datasets. The first full detection point is marked by a vertical line at the respective fault probability. Maximum score, sum of score, average of score, and minimum score show the best results, varying by dataset. Configuration: base-level encoding, D=10000, and Q=1

adaptive tuning of the detection process for varying fault probabilities. The best size for the window is selected based on Algorithm 1.

The detection process in Algorithm 1 begins by storing the similarity scores from the fault-free model as a baseline in a variable named *base_scores*. Faults are then incrementally injected into the CHVs until the model's accuracy drops beyond the specified tolerance. Each detection method is subsequently evaluated. The minimum value of the detection method applied to the *base_scores* is computed and stored as the threshold τ. Starting with a window size $W = 1$, the fault detection coverage is determined using τ. The window size is progressively doubled until full fault coverage (100%) is achieved. The best-performing method is selected, along with its corresponding τ and W, for hardware implementation. The final hardware configuration is designed based on these parameters, with further implementation details provided in the next subsection.

Algorithm 1 Fault Analysis and Detection Method Evaluation

Input: model, inputs, tolerance
Detection methods: `max_score`, `min_score`, `average`
for $data \in inputs$ **do**
    $base_scores \leftarrow$ SimilarityScore($model, data$)
end for
$fault_level \leftarrow 0\%$
$accuracy_drop \leftarrow 0$
while $accuracy_drop \leq tolerance$ **do**
    $fault_level \leftarrow fault_level + 1\%$
    $model_f \leftarrow$ InjectFault($fault_level, model$)
    $accuracy_drop \leftarrow$ TestModel($model_f, inputs$)
end while
for $method \in detection_methods$ **do**
    $\tau \leftarrow \min(method(base_scores))$
    $W \leftarrow 1$
    $detection \leftarrow 0$
    while $detection < 100$ **do**
        $detection \leftarrow method(model_f, W, \tau)$
        $W \leftarrow 2W$
    end while
    Save $method$, τ, and W
end for
Log the best hardware configuration based on the options

8.2.2. CED HDC Hardware Implementation

This work leverages a simple yet efficient hardware architecture as shown in the right-hand side of Figure 8.5 with the adjustable windowing approach. The proposed hardware supports the sum (average) of all similarity scores, the minimum score, and the maximum score, as they show the highest fault coverage in our evaluation. While both average and summation metrics provide the same detection coverage, the summation metric is preferred due to its lower cost.

8.2.2.1. Configuration

The customizable hardware is designed to operate seamlessly across metrics and adapt to any window size through the configurations. Based on the results of the offline fault analysis, one of the aforementioned methods is selected by the *sel* signal of the multiplexer. The window size (W) is used to configure the depth of the internal memory.

8.2.2.2. Metrics calculation

To achieve an efficient hardware implementation, we consider optimizations, such as a sequential adder and a sequential minimum finder working together with the argmax within HDC. All metrics are calculated simultaneously and sequentially (the maximum score is extracted directly from the argmax), the detection

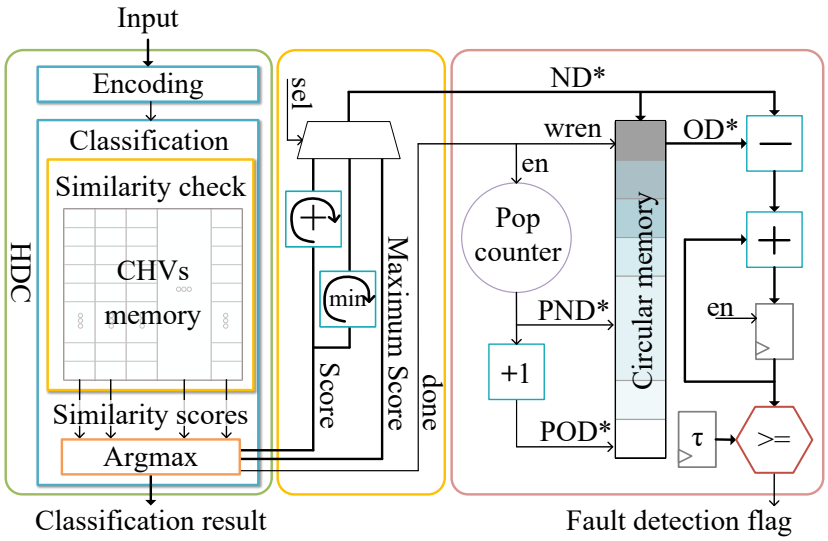


Figure 8.5.: Fault detection architecture. To simplify the architecture, the average score of the window is calculated before storing the new data in memory. The maximum score is obtained using Argmax, while the minimum and sum of scores are calculated sequentially.
ND*: New Data, PND*: Pointer to the Newest Data (writing address), POD*: Pointer to the Oldest Data (reading address).

implementation does not add any latency. The *done* signal indicates the availability of new data for the fault detection module.

8.2.2.3. Fault detection

As soon as a new data arrives, the summation of the data within the window is dynamically updated by removing the value of the oldest metric from the cumulative sum and adding the most recent metric. This process is managed through a simple circular memory structure. The circular memory design eliminates the need for complex memory controllers or intricate pointer management. Following the First-In-First-Out (FIFO) principle, older data is seamlessly overwritten as new data is added. Since the oldest data always follows the most recent, the read pointer can be effortlessly calculated by incrementing the write address. Also, to track the write addresses, a PopCount mechanism is employed, incrementing with each new input.

As a result, the system continuously tracks the average of the detection metric values within the window, which is efficiently calculated by summing all values in the window and shifting out the $\log_2 W$ least significant bits. If the average value falls below a predefined acceptance threshold, a fault flag is triggered. This implementation eliminates additional latency and enables efficient fault detection in real-time.

8.3. Experimental Results

In this section, we first present the experimental setup, including the selected HDC configurations, the datasets used, the fault injection procedure, and the HDC hardware platform and its setup. Next, we analyze how different datasets, HDC configurations, and the window size affect the fault coverage. Finally,

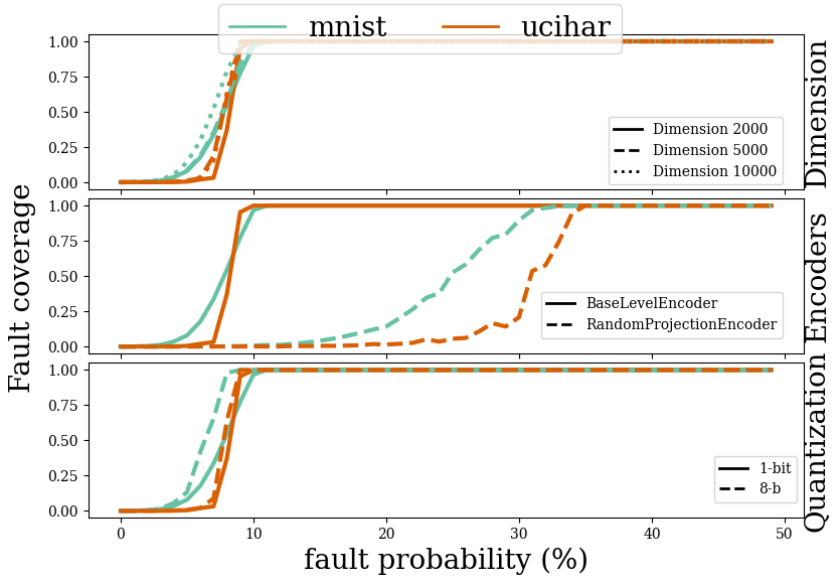


Figure 8.6.: Effects of HDC configurations on fault coverage. The proposed CED method operates effectively across different HDC configurations.

we provide a report on hardware implementation, utilization, and performance using the implementation of the MNIST dataset with our CED detection method.

8.3.1. Fault Detection

In this chapter, faults are injected into the CHV memory with fault probabilities from 0% to 60%. For a model with dimension D, number of classes C, quantization precision Q, and fault probability P_f, the number of injected bit flips is computed as

$$N_{\text{flip}} = D \times C \times Q \times P_f. \tag{8.1}$$

Each result is averaged over 20 independent runs with different random seeds.

To obtain a comprehensive evaluation of the proposed CED method, the effect of different HDC parameters on the efficiency of the detection process is investigated. The evaluation process considers three different dimension sizes (D) of 2000, 5000, and 10000, and two quantization levels (Q), 1-bit and 8-bit. In addition, two types of encoding methods, base-level and random projection, are employed.

Figure 8.6 visualizes the applicability of the proposed method by analyzing the impact of different HDC configurations on fault coverage, visualized for the MNIST and UCI-HAR datasets. The proposed detection remains effective across all configurations. The encoder has the most significant impact on fault coverage, with random projection proving more robust than the base-level encoder.

As shown in Figure 8.7, the window mechanism enables earlier fault detection. The minimum detectable accuracy drop (and corresponding fault probability) for a given window size is determined by selecting the best method among maximum, minimum, or average scores. For a window size of 1, detection is not

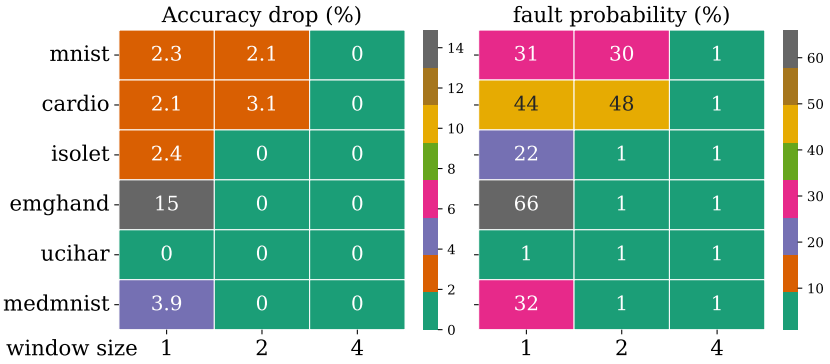


Figure 8.7.: Minimum fault probability / accuracy drop for full detection with given window size. With $W = 4$ even a fault probability of 1% and no accuracy drop is detectable. Results are reported for the best detection metric. Configuration: Random projection encoding, D=2000, Q=1.

Table 8.1.: resource utilization of the HDC IP and CED module on PYNQ-Z2 with respect to different window sizes (W)

Module	LUT	Registers	BRAM	Power
HDC IP	25651	61501	84	294 mW
$W = 1$	21(0.081%)	12(0.019%)	0	<1 mW
$W = 2$	22(0.085%)	14(0.022%)	0	<1 mW
$W = 4$	24(0.093%)	16(0.026%)	0	<1 mW
$W = 8$	25(0.097%)	18(0.029%)	0	<1 mW

always guaranteed, but with a window size of 4, all faults are detected across all benchmarks at only 1% fault probability, even before accuracy drops. As τ remains the lowest possible score in the fault-free model, increasing the window size does not cause false positives. To ensure an adjustable error detection rate, all detection methods should be evaluated in different window sizes and the one that best meets user requirements should be selected.

8.3.2. Hardware Implementation

The proposed CED method is small enough to be seamlessly integrated into a MNIST HDC model with $D = 2000$ on the PYNQ-Z2 platform, as shown in Figure 8.5. Due to our customizable design, the hardware overhead is reported for different window sizes in Table 8.1, providing a comparison of resource utilization between our method and the proposed RCEHDC implementation with base-level encoding, $D = 2000$ and 1-bit quantization. The fault detection hardware works independently of configuration parameters, including quantization bit width, and maintains consistent utilization across models as long as a fixed-point format is used in the classification.

The data for both the HDC IP and the integrated CED detection modules across different window sizes, as well as the negligible power overhead introduced by our fault detection mechanism. Additionally, our method does not introduce any latency, as the fault flag is generated simultaneously with the classification output of the last data within the window.

Table 8.2.: A comparison between resource utilization and latency with other state-of-the-art approaches.

Method	LUT	Registers	BRAM	DSP	Latency
[53]	716	412	48	2	$5ms$
[55]	4121	4435	61	2	$2ms$
Proposed (W = 8)	**25 (0.34‰)**	**18 (0.43‰)**	**0**	**0**	**0 ms**

We also compare our method with other state-of-the-art approaches in terms of hardware overhead and additional inference latency, as shown in Table 8.2. The only previous work that reports hardware metrics is [55]. However, their method is only effective for error probabilities below 10%, which is tolerable for most HDC models according to our observations. The reported results for [53] are also extracted from [55]. The results show that we have less than 1% hardware overhead compared to the prior state-of-the-art approaches. Furthermore, the overhead of all proposed techniques depends on the dataset and dimension size, whereas for our CED it is completely independent of the model and remains the same for any model.

8.3.3. Discussion

Our analysis of HDC fault tolerance and the proposed CED highlights its versatility across datasets, encoding methods, quantization bit widths, and dimension sizes. The study confirms that HDC's inherent fault tolerance enables adaptability to various operational scenarios, making it a reliable approach for handling runtime faults.

The results show that fault tolerance in HDC varies significantly depending on the dataset, while encoding methods play a key role among the other HDC configurations. Random projection encoding stands out due to its superior fault tolerance. While quantization type has minimal impact on detection, dimension size plays a more evident role, improving both fault tolerance and accuracy. However, detection is always possible before any accuracy drop occurs, regardless of dataset or HDC configuration. In conclusion, the proposed CED requires minimal additional hardware, making it an efficient extension for HDC accelerators to enhance reliability in noisy edge environments.

8.4. Conclusion

In conclusion, we introduce the first CED technique for HDC, specifically designed for hardware implementation, addressing the critical need for reliable and efficient fault detection in HDC. Unlike existing CEDs, which rely on excessive hardware resources and fall under the algorithmic fault tolerance of HDC, our approach overcomes these limitations, making it practical and effective even under severe fault scenarios. Our proposed CED scheme is both efficient and lightweight, ensuring compatibility with low-power edge devices. By detecting faults only when they compromise system reliability, the method maintains system resilience without sacrificing performance or efficiency. Additionally, it is

easily adjustable to tolerate a user-defined accuracy drop for balancing resource utilization and fault coverage.

9. Conclusion and Perspectives

9.1. Conclusion

This thesis addresses the efficient and reliable deployment of Hyperdimensional Computing on resource-constrained edge devices. Edge-AI systems require learning models that operate under strict memory, power, and hardware limitations while still providing dependable runtime behavior. Although HDC is a promising candidate for this domain due to its simple operations, high parallelism, and inherent robustness, practical hardware implementations still face significant limitations. In particular, conventional HDC encoders often require large hypervector memories or costly permutation and projection logic, while deployed accelerators remain vulnerable to memory faults, hardware-induced errors, and uncertain inputs.

To address these challenges, this thesis developed a hardware-aware HDC implementation that improves memory efficiency, hardware scalability, runtime reliability, and functional safety. The first part of the contribution focused on reducing the implementation cost of HDC accelerators. A memory-efficient on-the-fly generation encoder, OTFGEncoder, was introduced to replace stored ID-level and BHV memories with deterministic hardware generation. This memory-to-logic transformation removes a major bottleneck in HDC encoding and enables more scalable FPGA implementations. Building on this encoder, an end-to-end software–hardware framework was developed to map Python-based HDC models to configurable FPGA accelerators. The results showed that HDC can be implemented efficiently on small FPGA platforms while maintaining classification accuracy and reducing memory and power requirements.

After removing the encoder memory, the second part of the thesis focused on improving the reliability of HDC accelerators under remaining memory and hardware faults. In this context, CHV memory is a critical component of HDC inference because it directly affects classification results. This thesis first introduced a non-uniform error-correction method based on the observation that different regions of the CHV memory have varying levels of fault sensitivity. By selectively protecting the most sensitive regions, the proposed method improves fault resilience with lower overhead than uniform ECC protection. To further reduce the memory cost of fault correction, a memory-free ECC technique was proposed. This method avoids storing check bits and instead generates the required correction information during operation, making fault correction more suitable for highly constrained edge platforms.

The final part of the thesis addressed runtime safety and uncertainty detection. Lightweight detection methods were introduced to identify unreliable predictions caused by out-of-distribution inputs, hardware

faults, and CHV memory errors. These methods reuse information already available during HDC inference, especially similarity scores between the query hypervector and CHVs, thus avoiding expensive auxiliary models or complex runtime computations. The proposed monitoring mechanisms provide a practical safety layer for HDC accelerators with negligible hardware overhead and no additional latency.

Overall, this thesis demonstrates that HDC can be made more practical for edge-AI deployment by considering its algorithmic properties alongside hardware constraints. The proposed framework improves memory efficiency by eliminating large encoder memories, enhances hardware scalability through on-the-fly hypervector generation and configurable FPGA implementation, increases runtime reliability through selective and memory-free error correction, and supports functional safety through lightweight detection of hardware faults and anomalous inputs. Together, these contributions preserve the lightweight and highly parallel nature of HDC while extending it toward more dependable and safety-aware edge implementations.

9.2. Future Perspectives

Several directions remain open for future research. First, the on-the-fly generation strategy can be further optimized by exploring additional generator structures, seed selection methods, and configuration patterns. Such optimization may improve the quality of generated hypervectors and further reduce the gap between generated and fully random encoding parameters.

Second, the proposed framework can be extended to support a broader range of HDC encoding models. Although the current platform already supports online learning, further optimization is needed to make this mode more suitable for resource-constrained edge devices. In particular, future work can focus on reducing the additional memory and logic required for model updates while preserving the accelerator's low hardware cost.

Third, reliability-aware HDC design can be further expanded beyond memory faults. Future work may consider a broader range of hardware fault models, including timing errors, aging-induced faults, approximate computing effects, and faults in emerging memory technologies. Combining the proposed lightweight detection and correction techniques with adaptive voltage scaling could also enable more energy-efficient and reliable HDC accelerators.

Fourth, the proposed runtime monitoring mechanisms can be integrated into larger safety-aware edge-AI systems. In practical applications, detecting an unreliable prediction is only the first step. Future systems should also define appropriate recovery actions, such as reclassification, fallback execution, sensor resampling, model updates, or communication with a higher-level controller.

Future work can explore integrating the proposed HDC accelerator framework into real-world embedded platforms and open-source edge-AI toolchains. This integration would make the proposed techniques more accessible and support further research on efficient, reliable, and trustworthy HDC-based edge intelligence.

References

[1] M. S. Roodsari, J. Krautter, and M. Tahoori, "Otfgencoder-hdc: Hardware-efficient encoding techniques for hyperdimensional computing", in *2024 IEEE Design, Automation and Test in Europe Conference (DATE)*, IEEE, 2024, pp. 1–2.

[2] M. S. Roodsari, J. Krautter, V. Meyers, and M. Tahoori, "E 3 hdc: Energy efficient encoding for hyper-dimensional computing on edge devices", in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2024, pp. 274–280.

[3] "Rcehdc", Accessed: 2025. [Online]. Available: `https://github.com/ms-pr/RCEHDC`.

[4] M. S. Roodsari, S. Hemaram, and M. Tahoori, "Non-uniform error correction for hyperdimensional computing edge accelerators", in *2025 30th IEEE European Test Symposium (ETS)*, IEEE, 2025.

[5] M. S. Roodsari, M. Mayahinia, and M. Tahoori, "Mf-ecc: Memory-free error correction for hyperdimensional computing edge accelerators", in *2026 31st Asia and South Pacific Design Automation Conference (ASPDAC)*, IEEE, 2026.

[6] M. S. Roodsari, V. Meyers, and M. Tahoori, "Ced-hdc: Lightweight concurrent error detection for reliable hyperdimensional computing", in *2025 43rd IEEE VLSI Test Symposium (VTS)*, IEEE, 2025.

[7] M. S. Roodsari, V. Meyers, and M. Tahoori, "Lightweight concurrent out-of-distribution detection in hyperdimensional computing hardware", in *2025 31st IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, IEEE, 2025.

[8] M. S. Roodsari, V. Meyers, and M. Tahoori, "Safehdc: Concurrent uncertainty and fault detection in hyperdimensional computing", *IEEE Transactions on Device and Materials Reliability*, 2026.

[5] M. Satyanarayanan, "The emergence of edge computing", *computer*, vol. 50, no. 1, pp. 30–39, 2017.

[6] S. Majumder et al., "Smart homes for elderly healthcare—recent advances and research challenges", *Sensors*, vol. 17, no. 11, p. 2496, 2017.

[7] A. S. George, A. H. George, and T. Baskar, "Edge computing and the future of cloud computing: A survey of industry perspectives and predictions", *Partners Universal International Research Journal*, vol. 2, no. 2, pp. 19–44, 2023.

[8] B. Bajic, I. Cosic, B. Katalinic, S. Moraca, M. Lazarevic, and A. Rikalovic, "Edge computing vs. cloud computing: Challenges and opportunities in industry 4.0.", *Annals of DAAAM & Proceedings*, vol. 30, 2019.

[9] V. De and S. Borkar, "Technology and design challenges for low power and high performance", in *Proceedings of the 1999 international symposium on Low power electronics and design*, 1999, pp. 163–168.

[10] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation", *Ieee Micro*, vol. 25, no. 6, pp. 10–16, 2006.

[11] A. Rahimi and R. K. Gupta, "Hardware/software codesign for energy efficiency and robustness: From error-tolerant computing to approximate computing", in *Dependable Embedded Systems*, Springer, 2020, pp. 527–543.

[12] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors", *Cognitive Computation*, vol. 1, 2009.

[13] B. Varghese et al., "Revisiting the arguments for edge computing research", *IEEE Internet Computing*, vol. 25, no. 5, pp. 36–42, 2021.

[14] 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*, ACM, 2016, pp. 64–69.

[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*, 2019.

[16] 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)*, IEEE, 2017, pp. 1–8.

[17] R. W. Hamming, "Error detecting and error correcting codes", *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[18] H. Imai, *Essentials of error-control coding techniques*. Academic Press, 2014.

[19] B. Benjauthrit, L. Coady, and M. Trcka, "An overview of error control codes for data storage", in *Proceedings of Nonvolatile Memory Technology Conference*, IEEE, 1996, pp. 120–126.

[20] J. Samanta, J. Bhaumik, and S. Barman, "Compact and power efficient sec-ded codec for computer memory", *Microsystem Technologies*, vol. 27, pp. 359–368, 2021. DOI: 10.1007/s00542-019-04366-7.

[21] M. M. Ali, S. J. Hashim, M. A. Chaudhary, G. Ferré, F. Z. Rokhani, and Z. Ahmad, "A reviewing approach to analyze the advancements of error detection and correction codes in channel coding with emphasis on lpwan and iot systems", *IEEE Access*, vol. 11, pp. 127 077–127 097, 2023.

[22] S. Liang, Y. Li, and R. Srikant, "Enhancing the reliability of out-of-distribution image detection in neural networks", *arXiv preprint arXiv:1706.02690*, 2017.

[23] J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized out-of-distribution detection: A survey", *International Journal of Computer Vision*, vol. 132, no. 12, pp. 5635–5662, 2024.

[24] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks", *arXiv preprint arXiv:1610.02136*, 2016.

[25] D. Hendrycks et al., "Scaling out-of-distribution detection for real-world settings", *arXiv preprint arXiv:1911.11132*, 2019.

[26] R. Chan, M. Rottmann, and H. Gottschalk, "Entropy maximization and meta classification for out-of-distribution detection in semantic segmentation", in *International Conference on Computer Vision*, IEEE, 2021.

[27] W. Liu, X. Wang, J. Owens, and Y. Li, "Energy-based out-of-distribution detection", *Advances in Neural Information Processing Systems*, vol. 33, pp. 21 464–21 475, 2020.

[28] T. DeVries and G. W. Taylor, "Learning confidence for out-of-distribution detection in neural networks", *arXiv preprint arXiv:1802.04865*, 2018.

[29] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning", in *International Conference on Machine Learning*, PMLR, 2016.

[30] Y. Sun, Y. Ming, X. Zhu, and Y. Li, "Out-of-distribution detection with deep nearest neighbors", in *International Conference on Machine Learning*, PMLR, 2022, pp. 20 827–20 840.

[31] D. Kleyko, D. Rachkovskij, E. Osipov, and A. Rahimi, "A survey on hyperdimensional computing aka vector symbolic architectures, part ii: Applications, cognitive models, and challenges", *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–52, 2023.

[32] A. Hernández-Cano, N. Matsumoto, E. Ping, and M. Imani, "Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system", in *2021 DATE*, IEEE, 2021, pp. 56–61.

[33] P. Vergès, T. Givargis, and A. Nicolau, "Refinehd: Accurate and efficient single-pass adaptive learning using hyperdimensional computing", in *2023 IEEE International Conference on Rebooting Computing (ICRC)*, IEEE, 2023, pp. 1–8.

[34] J. Wang, S. Huang, and M. Imani, "Disthd: A learner-aware dynamic encoding method for hyperdimensional classification", in *2023 60th ACM/IEEE DAC*, IEEE, 2023, pp. 1–6.

[35] 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*, 2019.

[36] S. Salamat, M. Imani, and T. Rosing, "Accelerating hyperdimensional computing on fpgas by exploiting computational reuse", *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1159–1171, 2020.

[37] B. Khaleghi, S. Salamat, A. Thomas, F. Asgarinejad, Y. Kim, and T. Rosing, "SHEARer: Highly-efficient hyperdimensional computing by software-hardware enabled multifold approximation", in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020.

[38] M. Imani et al., "Revisiting hyperdimensional learning for FPGA and low-power architectures", in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021.

[39] T. Zhang, S. Salamat, B. Khaleghi, J. Morris, B. Aksanli, and T. S. Rosing, "HD2FPGA: Automated framework for accelerating hyperdimensional computing on FPGAs", in *2023 24th International Symposium on Quality Electronic Design (ISQED)*, IEEE, 2023.

[40] A. Safa, I. Ocket, F. Catthoor, and G. Gielen, "SupportHDC: Hyperdimensional computing with scalable hypervector sparsity", in *Proceedings of the 2023 Annual Neuro-Inspired Computational Elements Conference*, 2023.

[41] 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*, 2019, pp. 493–498.

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

[43] S. Datta, R. A. Antonio, A. R. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric IoT", *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, 2019.

[44] J. Liu, M. Ma, Z. Zhu, Y. Wang, and H. Yang, "Hdc-im: Hyperdimensional computing in-memory architecture based on rram", in *26th IEEE ICECS*, IEEE, 2019, pp. 450–453.

[45] H. Li et al., "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)*, IEEE, 2016, pp. 16–1.

[46] A. Kazemi et al., "Achieving software-equivalent accuracy for hyperdimensional computing with ferroelectric-based in-memory computing", *Scientific reports*, vol. 12, no. 1, p. 19 201, 2022.

[47] Y. H. Oh, C. Hwang, J. Kim, K. E. Jeon, J. H. Ko, et al., "Memhd: Memory-efficient multi-centroid hyperdimensional computing for fully-utilized in-memory computing architectures", in *2025 Design, Automation & Test in Europe Conference (DATE)*, IEEE, 2025, pp. 1–7.

[48] M. Heddes et al., "Torchhd: An open source python library to support research on hyperdimensional computing and vector symbolic architectures", *Journal of Machine Learning Research*, vol. 24, 2023. [Online]. Available: `http://jmlr.org/papers/v24/23-0300.html`.

[49] X. Wang, B. Wang, Y. Wu, Z. Ning, S. Guo, and F. R. Yu, "A survey on trustworthy edge intelligence: From security and reliability to transparency and sustainability", *IEEE Communications Surveys & Tutorials*, vol. 27, no. 3, pp. 1729–1757, 2024.

[50] W. Xu, S. Fuhrman, K. Fan, S. Pinge, W.-C. Chen, and T. Rosing, "Hypermetric: Efficient hyperdimensional computing with metric learning for robust edge intelligence", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[51] S. Zhang, R. Wang, J. J. Zhang, A. Rahimi, and X. Jiao, "Assessing robustness of hyperdimensional computing against errors in associative memory", in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2021, pp. 211–217.

[52] T. Iwasaki and M. Shintani, "Lifetime improvement method for memristor-based hyperdimensional computing accelerator", in *2023 IEEE IMFEDK*, IEEE, 2023, pp. 1–2.

[53] S. Zhang, M. Imani, and X. Jiao, "Scalehd: Robust brain-inspired hyperdimensional computing via adaptive scaling", in *Proceedings of the 41st IEEE/ACM ICCAD*, 2022, pp. 1–9.

[54] D. Liang, H. Awano, N. Miura, and J. Shiomi, "Dependablehd: A hyperdimensional learning framework for edge-oriented voltage-scaled circuits", in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023, pp. 416–422.

[55] M. Mejri, C. Amarnath, and A. Chatterjee, "Error resilient hyperdimensional computing using hypervector encoding and cross-clustering", in *2024 IEEE 42nd VTS*, IEEE, 2024, pp. 1–7.

[56] M. Mejri, C. Amarnath, and A. Chatterjee, "Deeper-hd: An error resilient hyperdimensional computing framework with dnn front-end for feature selection", in *2024 IEEE 25th LATS*, IEEE, 2024, pp. 1–6.

[57] Z. Liang et al., "Conformalhdc: Uncertainty-aware hyperdimensional computing with application to neural decoding", *arXiv preprint arXiv:2602.21446*, 2026.

[58] F. Yang and S. Ren, "On the vulnerability of hyperdimensional computing-based classifiers to adversarial attacks", in *International Conference on Network and System Security*, Springer, 2020, pp. 371–387.

[59] D. Ma, T. Š. Rosing, and X. Jiao, "Testing and enhancing adversarial robustness of hyperdimensional computing", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 4052–4064, 2023.

[60] R. Wang and X. Jiao, "Poisonhd: Poison attack on brain-inspired hyperdimensional computing", in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 298–303.

[61] L. Chen, J. Wang, T. Mortlock, P. Khargonekar, and M. A. Al Faruque, "Hyperdimensional uncertainty quantification for multimodal uncertainty fusion in autonomous vehicles perception", in *Proceedings of the Computer Vision and Pattern Recognition Conference*, 2025, pp. 22 306–22 316.

[62] S. Wilson, T. Fischer, N. Sünderhauf, and F. Dayoub, "Hyperdimensional feature fusion for out-of-distribution detection", in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2023, pp. 2644–2654.

[63] Z. A. Daniels et al., "Efficient model adaptation for continual learning at the edge", *arXiv preprint arXiv:2308.02084*, 2023.

[64] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[65] A. Krizhevsky, G. Hinton, et al., "Learning multiple layers of features from tiny images", *University of Toronto, Canada*, 2009.

[66] A. Krizhevsky, "Learning multiple layers of features from tiny images", Tech. Rep., 2009.

[67] R. Cole and M. Fanty, *ISOLET*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C51G69, 1991.

[68] J. Reyes-Ortiz, D. Anguita, A. Ghio, L. Oneto, and X. Parra, *Human Activity Recognition Using Smartphones*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C54S4K, 2013.

[69] N. Krilova, I. Kastalskiy, V. Kazantsev, V. Makarov, and S. Lobov, *EMG Data for Gestures*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C5ZP5C, 2018.

[70] D. Campos and J. Bernardes, *Cardiotocography*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C51S4N, 2000.

[71] J. Yang et al., "Medmnist v2-a large-scale lightweight benchmark for 2d and 3d biomedical image classification", *Scientific Data*, vol. 10, no. 1, p. 41, 2023.

[72] N. Mu and J. Gilmer, "MNIST-C: A robustness benchmark for computer vision", *arXiv preprint arXiv:1906.02337*, 2019.

[73] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning", *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017. [Online]. Available: `http://jmlr.org/papers/v18/16-365`.

[74] M. Damiani, P. Olivo, M. Favalli, S. Ercolani, and B. Ricco, "Aliasing in signature analysis testing with multiple input shift registers", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 12, 1990.

[75] P. Vergès, M. Heddes, I. Nunes, T. Givargis, and A. Nicolau, "HDCC: A hyperdimensional computing compiler for classification on embedded systems and high-performance computing", *arXiv preprint arXiv:2304.12398*, 2023.

[76] J. Krautter, P. Genssler, G. Sepanta, H. Amrouch, and M. Tahoori, "Stress-resiliency of AI implementations on FPGAs", in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023.

[77] M. Imani et al., "Quanthd: A quantization framework for hyperdimensional computing", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2268–2278, 2019.

[78] Y. Umuroglu et al., "Finn: A framework for fast, scalable binarized neural network inference", in *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 65–74.

[79] D. H. Lee, C. H. Kim, J. Youn, and J. Jeong, "Evaluation methods for long-term reliability of polymer-based implantable biomedical devices", *Biomedical Engineering Letters*, vol. 11, pp. 97–105, 2021.

[80] Z. Cai, Z. Zhou, H. Yu, and Z. Wang, “Low voltage memory test algorithm and fault injection technology”, in *2021 IEEE 4th ICET*, IEEE, 2021, pp. 299–302.

[81] J.-Y. Li and W.-C. Fang, “An edge ai accelerator design based on hdc model for real-time eeg-based emotion recognition system with risc-v fpga platform”, in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2024, pp. 1–5.

[82] T. Saito and M. Rehmsmeier, “The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets”, *PloS one*, vol. 10, no. 3, e0118432, 2015.

List of Acronyms

AI	Artificial Intelligence
BHV	Base Hypervector
BNN	Binary neural network
BRAM	Block Random Access Memory
CED	Concurrent Error Detection
CHV	Class Hypervector
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DSP	Digital Signal Processing
ECC	Error-Correcting Code
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
HDC	Hyperdimensional Computing
HLS	High-Level Synthesis
HV	Hypervector
ID	Identifier
IMC	In-Memory Computing
IoT	Internet of Things
IP	Intellectual Property
LFSR	Linear Feedback Shift Register
LUT	Look-Up Table
MISR	Multiple Input Signature Register
ML	Machine Learning
MLP	Multilayer Perceptron

NN	Neural Network
OOD	Out-of-Distribution
PopCount	Population Count
QHV	Query Hypervector
RCEHDC	Resource-Constrained Edge Hyperdimensional Computing
ReRAM	Resistive Random Access Memory
ROC	Receiver Operating Characteristic
RTL	Register Transfer Level
PR	Precision-Recall
SEC-DED	Single-Error Correction and Double-Error Detection

List of Figures

1.1. Overview of the thesis structure and objectives 5

2.1. Overview of Hyperdimensional Computing 10
2.2. Overview of OOD detection in an HDC-based classifier. 15

3.1. Base-level encoding in HDC 31
3.2. On-the-fly memory-to-logic encoding 32
3.3. LFSR, MISR, and BHV-Gen comparison 34
3.4. E^3HDC accelerator architecture 35
3.5. End-to-end FPGA mapping flow 37
3.6. HDC accelerator floorplan on PYNQ-Z2 40

4.1. Column-wise access pattern to CHV memory 44
4.2. Non-uniform ECC framework 45
4.3. Column-Hamming-weight fault sensitivity 46
4.4. Fault impact in sensitive and less-sensitive regions 47
4.5. Non-uniform ECC architecture 47
4.6. Accuracy comparison of non-uniform ECC with state-of-the-art 50
4.7. FPGA floorplan of non-uniform ECC 51

5.1. Memory-free ECC framework 54
5.2. CHV and ECC memory layout for memory-free error correction 55
5.3. Memory-free ECC architecture for the HDC accelerator 56
5.4. Accuracy preservation under CHV fault probability with memory-free ECC 59
5.5. Multi-bit error correction accuracy across datasets 60
5.6. Memory utilization of baseline and memory-free ECC 61

6.1. ID threshold calibration for OOD detection 65
6.2. ID and OOD score distributions 66
6.3. OOD detector hardware architecture 67
6.4. Effect of HDC parameters on OOD detection 68
6.5. OOD detection results across datasets 69

7.1. Accuracy degradation under runtime uncertainty 76
7.2. Fault detection precision across uncertainty scenarios and different metrics 77
7.3. Similarity-threshold calibration 79
7.4. Similarity score distributions under faults and uncertainty 80
7.5. SafeHDC hardware integration 81
7.6. Precision-recall curves for reliability detection 82
7.7. Accuracy drop across fault and uncertainty types 84
7.8. Maximum similarity score distributions for different encodings 85

8.1. CED-HDC detection flow 90
8.2. Accuracy degradation under different fault probabilities 91

8.3. Score distributions under CHV faults 92
8.4. CED metrics under fault injection 93
8.5. CED-HDC fault detection architecture 95
8.6. Fault coverage across HDC configurations 96
8.7. Window size impact on fault coverage 97

List of Tables

2.1. Comparison of ML paradigms for edge deployment 10
2.2. Comparison of HDC encoding methods 12
2.3. Dataset descriptions 25

3.1. OTFGEncoder accuracy impact across datasets 38
3.2. Comparison of HDC encoding techniques on hardware consumption 39
3.3. E^3HDC hardware consumption other MLs 39

4.1. ECC codewords and databits 49
4.2. Comparison of non-uniform ECC with other state-of-the-art 50
4.3. Non-uniform hardware overhead compared to HDC 51
4.4. Non-uniform resource utilization compared to state-of-the-art 51

5.1. MF-ECC Accuracy improvement compared to standard ECC 61
5.2. Accuracy comparison of MF-ECC with state-of-the-art 61
5.3. MF-ECC hardware overhead compared to HDC 62
5.4. MF-ECC resource utilization compared to state-of-the-art 62

6.1. HDC configuration factors for OOD detection 68
6.2. OOD detection results over different configuration 71
6.3. Comparison of OOD detection with other methods 71
6.4. OOD resource utilization 71
6.5. Hardware OOD detection rate on MNIST 72

7.1. HDC configuration factors for uncertainty detection 83
7.2. Uncertainty detection rate over different configurations 85
7.3. Uncertainty detection hardware consumption 86

8.1. CED-HDC resource utilization compared to HDC 97
8.2. CED-HDC resource utilization compared to state-of-the-art 98