UNIVERSITY OF CALIFORNIA,
IRVINE

Extending HyperDimensional Computing to Reinforcement Learning for Sustainable and Transparent Learning

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Mariam Ali Issa

Dissertation Committee:
Assistant Professor Mohsen Imani, Chair
Professor Elaheh Bozorgzadeh
Assistant Professor Hamidreza Aghasi

2026

DEDICATION

To my older brother,
Alaa Ali Issa,
who inspired this entire journey.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor, Prof. Mohsen Imani, for his invaluable mentorship throughout the five years of this program. He has opened up incredible research opportunities and collaborations and has guided me through all the challenges and setbacks that come with a doctoral program. I am deeply grateful for his patience and insight that have allowed me to reach this final stage. I would also like to thank Eli Bozorgzadeh, Hamidreza Aghasi, and Sang-woo Jun, for their guidance, enthusiasm, and energy in serving as members of my doctoral committee.

I would also like to thank a number of individuals from the Bio-inspired Architecture Lab. To Yang Ni: your mentorship in the first year of the program has been invaluable as it allowed me to quickly get up to speed on brain-inspired computing and reinforcement learning. To Zhuowen Zou: your steadfast mentorship, especially as I transitioned my research into the theoretical realm of our group's research, has pushed me past my preconceived limits. I would also like to thank my lab mates: Hanning Chen, Hamza Errahmouni Barkam, and Ali Zakeri, for their research collaborations and support in navigating the many milestones we worked on and achieved in parallel. Additionally, I would like to thank Pietro Mercati, Ranganath Krishnan, and Raid Ayoub from Intel Research Labs for their mentorship throughout this journey.

To the friends I made along the way during my time at UCI: Marshall Alexander Campbell, Sakshi Agarwal, Raquel Romero Buena, Lena Gerritz, Federica Zoe Ricci, Maia Tarnas, Amy Wu—I have been so lucky to be surrounded by such incredible individuals and am consistently inspired by all that each of you do! To my gem of a friend, Shanshan (Winnie) Tran: your kindness and positive outlook on life have grounded me so much during this time, and our conversations and time spent together have brought me so much joy. Thank you for being my cheerleader through it all.

Lastly, the foundation for all that I do: my family. My parents, who were immigrants and Palestinian refugees: you worked tirelessly and made endless sacrifices to ensure that my four siblings and I could attend and be the first generation to graduate from the University of California (Berkeley, San Diego, Davis, and, now, Irvine). To my siblings: Amal, Reema, and Nasser—not everyone lucks out to grow up with such loving and supportive siblings who are there for each other in every milestone and challenge that we encounter and overcome in life. To my older brother, Alaa, who passed away at the very start of my graduate studies: you are the reason I took my first Computer Science class, the reason I decided to switch my major to Math and Computer Science in my undergraduate studies, and the reason I was first exposed to the exciting world of artificial intelligence and machine learning—you are the reason this dissertation exists. You are sorely missed. Despite the challenges and doubts I encountered on this journey, I persevered because I knew I wanted to dedicate my work and research to you. This dissertation is as much a part of your legacy as it is of mine.

I would also like to thank the National Science Foundation for its generous Graduate Research Fellowship, which funded a majority of my graduate studies, as well as the Donald Bren

School of Computer Science and Informatics for awarding me both the Dean's and Excellence Fellowships to support my first year of graduate studies upon admission. I would also like to thank the Semiconductor Research Corporation (SRC) for its support as an SRC Research Scholar.

Finally, this dissertation is derived from the following published works. Each is reprinted with permission from the publisher. The coauthors are listed in full. Professor Mohsen Imani directed and supervised the research that forms the basis for this dissertation.

The text of Chapter 2 of this dissertation is a reprint of the material as it appears in Y. Ni, D. Abraham, M. Issa, Y. Kim, P. Mercati, M. Imani. Efficient Off-Policy Reinforcement Learning via Brain-Inspired Computing. In the Proceedings of the Great Lakes Symposium on VLSI 2023, pages 449-453. ACM, 2023. The coauthors in this publication are Yang Ni, Danny Abraham, Mariam Issa, Yeseong Kim, Pietro Mercati, and Mohsen Imani.

The text of Chapter 3 of this dissertation is a reprint of the material as it appears in M. Issa, S. Shahhosseini, Y. Ni, T. Hu, D. Abraham, A.M. Rahmani, N. Dutt, M. Imani. Hyperdimensional Hybrid Learning on End-Edge-Cloud Networks. In the Proceedings of the 40th IEEE International Conference on Computer Design (ICCD), pages 652-655. IEEE, 2022. The coauthors in this publication are Mariam Issa, Sina Shahhosseini, Yang Ni, Tianyi Hu, Danny Abraham, Amir M. Rahmani, Nikil Dutt, and Mohsen Imani.

The text of Chapter 4 of this dissertation is a reprint of the material as it appears in M. Issa, H. Chen, J. Wang, M. Imani. CyberRL: Brain-Inspired Reinforcement Learning for Efficient Network Intrusion Detection. In IEEE's 44th volume of Transactions on Computer-Aided Design of Integrated Circuits and Systems, pages 241-250. IEEE, 2025. The coauthors in this publication are Mariam Issa, Hanning Chen, Junyao Wang, and Mohsen Imani.

VITA

Mariam Ali Issa

EDUCATION

Doctor of Philosophy in Computer Science **2026**
University of California, Irvine *Irvine, CA*

Bachelor of Science in Math-Computer Science **2019**
University of California, San Diego *La Jolla, CA*

RESEARCH EXPERIENCE

Graduate Research Assistant **2021–2026**
University of California, Irvine *Irvine, CA*

TEACHING EXPERIENCE

Teaching Assistant: Machine Learning and Data Mining **2026**
University of California, Irvine *Irvine, CA*

Teaching Assistant: Machine Learning and Data Mining **2025**
University of California, Irvine *Irvine, CA*

WORK EXPERIENCE

Machine Learning Engineering Intern **2024**
Meta *New York, NY*

Software Engineer: Risk Engineering **2019–2021**
Gusto *San Francisco, CA*

REFEREED JOURNAL PUBLICATIONS

Dynamic MAC Protocol for Wireless Spectrum Sharing via Hyperdimensional Self-Learning **2024**
IEEE Access

CyberRL: Brain-Inspired Reinforcement Learning for Efficient Network Intrusion Detection **2025**
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

REFEREED CONFERENCE PUBLICATIONS

Robust In-Memory Computing with Hyperdimensional Stochastic Representation **November 2021**
IEEE/ACM International Symposium on Nanoscale Architectures

HDPG: Hyperdimensional Policy-based Reinforcement Learning for Continuous Control **July 2022**
Proceedings of the IEEE/ACM Design Automation Conference (DAC)

Distributed Reconfigurable Accelerator for Hyperdimensional Reinforcement Learning **October 2022**
Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)

Hyperdimensional Hybrid Learning on End-Edge-Cloud Networks **October 2022**
Proceedings of the 40th IEEE International Conference on Computer Design (ICCD)

Beyond von Neumann Era: Brain-inspired HyperDimensional Computing to the Rescue **January 2023**
Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC)

Value-based Reinforcement Learning using Efficient HyperDimensional Computing **April 2023**
Proceedings of the IEEE/ACM Design Automation and Test in Europe Conference (DATE)

Efficient Off-Policy Reinforcement Learning via Brain-Inspired Computing **June 2023**
Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)

Late Breaking Results: Scalable and Efficient HyperDimensional Computing for Network Intrusion Detection **July 2023**
Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)

ABSTRACT OF THE DISSERTATION

Extending HyperDimensional Computing to Reinforcement Learning for Sustainable and Transparent Learning

By

Mariam Ali Issa

Doctor of Philosophy in Computer Science

University of California, Irvine, 2026

Assistant Professor Mohsen Imani, Chair

Reinforcement Learning (RL) is a field of dynamic control that leverages Machine Learning (ML) and has been employed in diverse areas such as robotics, games, navigation, and industrial automation. Currently, many state-of-the-art RL algorithms rely on deep neural networks. However, deep learning faces significant challenges: high computational demands due to costly backpropagation and gradient-based methods, extreme sensitivity to noise in data, networks, and underlying hardware, and a lack of human-like cognitive support for long-term memorization and transparency. To address these limitations, this dissertation turns to brain-inspired learning paradigms, specifically HyperDimensional Computing (HDC), which is gaining traction for its high efficiency, robust operation, and ability to be implemented on lightweight hardware. We develop a suite of HDC-based RL algorithms that demonstrate the viability of this approach. Our contributions include an HDC-based Q-learning algorithm for intrusion detection in cybersecurity (CyberRL); a HyperDimensional Hybrid Learning (HDHL) model for resource management; and an HDC-based Soft Actor-Critic (HDCyberSAC) algorithm. Our results show that these methods offer a more efficient and robust alternative to deep neural network-based RL, paving the way for next-generation edge computing applications.

Chapter 1

Introduction

1.1 Overview

The proliferation of Internet of Things (IoT) devices has created an unprecedented demand for intelligent processing at the edge [13]. Applications in autonomous systems, real-time cybersecurity, and personalized healthcare require low latency, energy efficiency, and robust decision-making without reliance on continuous cloud connectivity [77, 1]. This necessitates a paradigm shift from cloud-offloaded learning to true learning on the edge, where devices must adapt and reason within severe constraints on power, computation, and memory.

Reinforcement Learning (RL) has emerged as a critical framework for such edge-based intelligence, enabling agents to learn optimal policies through direct interaction with their environment. However, a fundamental limitation of classical RL is its poor scalability to complex problems, a challenge known as the "state explosion problem" [2]. Deep learning has addressed this by serving as a powerful function approximator, enabling remarkable successes in deep reinforcement learning (DRL). These DRL algorithms can tackle high-dimensional state spaces, from playing complex games to controlling robotic systems [62, 84].

However, the reliance on deep neural networks comes at a significant cost and introduces well-studied limitations, including massive computational demands from backpropagation and gradient-based optimization, extreme sensitivity to noise in data, network parameters, and underlying hardware, and a lack of human-like cognitive attributes such as efficient long-term memorization and interpretability [85]. For edge devices, these challenges are prohibitive. Offloading computation to the cloud is suboptimal due to unreliable network connectivity, critical latency requirements in time-sensitive applications, and serious privacy concerns surrounding raw data transmission.

A recently pioneered alternative is HyperDimensional Computing (HDC), a computational model inspired by the brain's manipulation of high-dimensional distributed representations [52]. HDC was initially developed to address "the binding problem" of connectionist models by providing a structured, symbolic framework [28]. This paradigm is inspired by how the human brain processes sensory data in high dimensions due to its extensive circuitry. HDC abstracts this process by operating on dense, holistic representations called *hypervectors*, and by defining key cognitive operations like binding and bundling using efficient, hardware-native algebraic operations (e.g., element-wise XOR/Multiply and ADD) [28]. Consequently, HDC has gained interest due to its computational efficiency, enhanced learning quality, robustness to hardware errors, and lightweight hardware implementation on platforms like FPGAs [28, 76, 96, 21, 57].

Given these distinctive advantages, HDC presents itself as a promising solution for deploying intelligent algorithms on low-power IoT devices [16, 18]. While state-of-the-art ML predominantly relies on computationally expensive deep learning, making it impractical for the edge [61], HDC-based algorithms offer a path toward sustainable and robust edge learning [97].

Therefore, in this dissertation, we argue for HDC as a foundational framework for reinforcement learning on resource-constrained devices. We introduce and validate a suite of HDC-based RL algorithms designed for applications in cybersecurity and resource management.

We demonstrate that the brain-inspired properties of HDC offer competitive advantages over deep learning in a number of environments.

The following is a summary of contributions for each chapter:

- Chapter 2 introduces a novel positional encoding scheme for HDC that serves as the foundational representation for value-based RL algorithms.
- Chapter 3 presents HyperDimensional Hybrid Learning (HDHL) [47], a hybrid RL approach that combines model-free and model-based learning for dynamic resource optimization in an End-Edge-Cloud Network. HDHL utilizes HDC for both direct policy learning and classification tasks within its model-based component.
- Chapter 4 presents CyberRL [48], an HDC-based reinforcement learning algorithm for developing offensive and defensive strategies in a simulated network intrusion environment, showcasing HDC's efficiency for security tasks.
- Chapter 5 proposes an HDC-based Soft Actor-Critic (HDCyberSAC) algorithm, which is designed to solve non-stationary, partially observable RL problems, extending the benefits of HDC to complex tasks.

1.2 Background: HyperDimensional Computing

HyperDimensional Computing (HDC) is motivated by the theoretical neuroscience observation that the human brain processes information by operating on high-dimensional data representations. This concept informs the core algorithmic design behind HDC, which is to map objects into their high-dimensional representations.

This is done by encoding the original data into *hypervectors*, which are vectors comprising thousands of elements. Abstractly, each dimension of a hypervector represents the func-

tionality of a neuron in processing external stimuli [52]. Once an HDC model is trained, the encoded hypervector captures the important patterns of its respective class across its dimensions. This encoding allows HDC to be holographic, since information is evenly distributed throughout the hypervector. This property is ideal for dealing with hardware errors; therefore, HDC models remain robust even in the presence of bit flips [22, 94].

To explain how HDC works in greater detail, let us assume that $\vec{\mathcal{H}}_1$ and $\vec{\mathcal{H}}_2$ are two randomly initialized hypervectors, where $\vec{\mathcal{H}}_1, \vec{\mathcal{H}}_2 \in \{0,1\}^d$ and d is the dimensionality (e.g., $d = 10,000$). Due to random generation and high dimensionality, these two vectors are nearly orthogonal: $\delta(\vec{\mathcal{H}}_1, \vec{\mathcal{H}}_2) \approx 0$, where δ is a defined similarity function between the two hypervectors.

By operating in high-dimensional space, the multitude of nearly orthogonal hypervectors provides a foundational basis for the HDC model. The HDC model defines operations, including *bundling* and *binding*, to combine hypervectors while maintaining their respective information with high probability. The training process involves encoding the original input data and bundling those of the same class to produce a class hypervector, which is the representation of the class in high-dimensional space.

The HDC operation *binding* performs the cognitive computation and learning over the encoded data during the training phase to produce highly accurate, high-dimensional representations. Given a new data point to classify, the inference algorithm performs a similarity search between the class hypervectors and the encoded new data point to inform the model's decision-making. The HDC operations are described in greater detail below:

Bundling is the component-wise additive operation of hypervectors, where the information from multiple hypervectors is stored in a single hypervector. This constitutes the memorization capability of HDC, since the bundled *hypervector* is similar to each of the component hypervectors of which it is comprised (i.e., $\delta(\vec{\mathcal{H}}_1 + \vec{\mathcal{H}}_2, \vec{\mathcal{H}}_1) \approx 1$).

The **Binding** operation associates items in hyperspace. Given the hypervectors $\vec{\mathcal{H}}_1$ and $\vec{\mathcal{H}}_2$, component-wise multiplication (XOR in binary) denoted as $(\cdot)$ is conducted to produce a new hypervector that is dissimilar to its constituent vectors (i.e., $\delta(\vec{\mathcal{H}}_1 \cdot \vec{\mathcal{H}}_2, \vec{\mathcal{H}}_1) \approx 0$). This is ideal for constructing hypervectors for variable-value association.

The combination of these properties and operations has made HDC a powerful and versatile tool. In recent years, HDC has been utilized for various machine learning tasks given its desirable properties, including its fast convergence in learning, power and energy efficiency [17, 96, 71], robustness to noise and hardware errors, and acceleration on edge devices [90, 7].

1.2.1 Background: Reinforcement Learning

Reinforcement Learning (RL) is a field of dynamic control that leverages ML. An RL problem is formulated as a goal-oriented agent interacting with a defined environment to accomplish its designated task. To model this interaction, the Markov Decision Process (MDP), a discrete, stochastic framework for modeling systems influenced by external control, is employed. It is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where:

- $\mathcal{S}$: the set of all possible states that the environment can be configured in,
- $\mathcal{A}$: the set of actions that the agent may take to interact with the environment,
- $\mathcal{T}$: $P(\mathcal{S}'|\mathcal{S}, \mathcal{A})$, the transition probability of progressing to state $\mathcal{S}'$ given state $\mathcal{S}$ and action $\mathcal{A}$,
- $\mathcal{R}$: the reward function, which is commonly defined as $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ or $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}' \to \mathbb{R}$. It quantifies the desirability of the agent's actions and progress toward its goal [86].

Given this framework, RL can be expressed as an agent selecting actions to transition to

different states in its environment until it reaches a terminal state, signifying the success or failure of its objective. The agent learns by utilizing feedback from the reward function $\mathcal{R}$ to refine its strategy with the goal of maximizing the expected cumulative return $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, where $\gamma \in [0, 1]$ is the discount factor, which weights the importance of immediate versus future rewards.

There are a number of approaches within RL. One collection of algorithms, known as model-based learning, focuses on learning the environment dynamics (e.g., the reward function and transition probabilities) to build a model that emulates the environment. Alternatively, within model-free RL, which learns through direct interaction with the environment, there are policy-based and value-based approaches. The latter, value-based methods, learn by evaluating states in the environment, favoring those that are likely to yield high future rewards. Meanwhile, policy-based RL focuses on learning a policy for action selection.

While deep neural networks have become the dominant function approximators in modern RL, they introduce significant challenges, including high computational cost, sensitivity to hyperparameters and noise, and poor sample efficiency. This work explores an alternative paradigm by leveraging Hyperdimensional Computing (HDC) as a more efficient, robust, and brain-inspired framework for function approximation within RL algorithms. We hypothesize that HDC can mitigate these challenges while maintaining competitive performance, particularly in resource-constrained environments.

Chapter 2

HDC Posisional Encoding for Value-Based RL

2.1 Introduction

Reinforcement Learning (RL) has opened up new opportunities to solve a wide range of complex predictions and decision-making tasks that were previously out of reach for a machine [10, 36]. Compared to supervised and unsupervised learning methods, RL does not have direct access to labeled training data. Instead, it trains a self-learning agent using the observations of states and feedback rewards from the environment [27]. The learning process of RL is similar to how human beings learn to perform a new task. Initially, the RL agent interacts with the environment and tries to take action with no prior knowledge or past experience. Learning through the interaction of an environment makes RL appealing to the field of dynamic control and automated system optimization, where the optimal policy is hard to define and is constantly changing with its environment [10].

Q-learning is one of the most popular model-free reinforcement learning methods. With

the advancement in Deep Neural Networks (DNNs), the traditional Q-learning methods are changing to Deep Q-Networks (DQN) [38]. Unlike Q-learning that relies on large tables, DQN exploits DNN to learn an approximation of Q-value for every pair of action and state. In DQN, the actions and states are inputs to a DNN and the outputs are Q-values for each possible action. Recently, there has been active developments for various DQN applications such as playing GO [81] and computer games [62], system optimization [50], Genomics [43, 44, 42], and dynamic resource management [35]. DQN is capable of learning complex tasks without modeling the environment, but its power comes at a price, i.e., the huge computation cost and long learning time. This makes it only suitable for powerful computers in the cloud rather than mobile devices at the edge. However, edge devices play an increasingly important role in daily life, and they directly interact with the environment and collect feedback. Edge devices are natural platforms for RL, except that they have a limited energy budget and computation power. Offloading RL to the cloud not only leads to extra communication overhead but also causes security and privacy concerns.

To apply RL at the edge for mobile and sensor-like devices, we redesign the RL algorithm by exploiting the brain-inspired HyperDimensional Computing (HDC) [52]. Compared to DNN, HDC is highly efficient and robust against noise. HDC is motivated by how human brains process different kinds of inputs, i.e., brains express information using a vast number of neurons. The information is then processed and memorized in a holistic and high-dimensional way. HDC is powered by a set of well-defined HDC operations that mimic the functionalities of human brains and enable learning. For inputs in the lower-dimensional space, HDC usually encodes them to vectors of several thousand dimensions, i.e., hypervectors. The learning process is based on highly-parallelizable operations of hypervectors, such as element-wise addition and multiplication. Thus, HDC has been applied as a lightweight machine learning solution to multiple applications, e.g., bio-signal classification [63, 75], speech recognition [41], reasoning [73, 97]. In these applications, HDC is capable of achieving comparable accuracy to DNN with significantly higher efficiency. HDC also possesses an

intriguing beneficial advantage from its brain-like memorization ability, i.e., HDC achieves one-shot or few-shot learning with high quality [37].

However, current HDC solutions mainly focus on traditional classification and clustering. In contrast, in this paper, we propose QHD, a Hyperdimensional Reinforcement Learning, that mimics brain properties towards robust and real-time learning. The main contributions of the paper are listed as follows:

- To the best of our knowledge, QHD is the first hyperdimensional reinforcement learning algorithm. QHD relies on lightweight HDC models to learn an optimal policy in an unknown environment. We first develop a novel mathematical foundation and encoding module that maps state-action space into high-dimensional space. We accordingly develop a hyperdimensional regression model to approximate the Q-value function. QHD-powered agent makes decisions by comparing Q-values of each possible action. Our neural-inspired QHD uses operations that are extremely hardware-friendly for edge devices.

- We evaluate the effect of the different RL training batch sizes and local memory capacity on the QHD quality of learning. Thanks to the brain-like hyperdimensional operations, QHD is able to utilize even a small amount of available training data. It thereby supports a much smaller training batch size than DQN while still providing high-quality results. Our QHD is also capable of online learning with tiny local memory capacity, which can be as small as the training batch size. QHD provides real-time learning by further decreasing the memory capacity and the batch size. This makes QHD suitable for highly-efficient reinforcement learning in the edge environment, where it is crucial to support online and real-time learning.

We compare our QHD accuracy and efficiency with the state-of-the-art DQN algorithms for multiple dynamic control tasks. Our evaluation shows that QHD achieves significantly better

overall efficiency than DQN while providing higher or comparable rewards. Our solution also supports a small experience replay batch size that provides 12.3× speedup compared to DQN while ensuring minimal quality loss. Our evaluation shows QHD capability for real-time learning, providing 34.6× speedup and significantly better quality of learning than state-of-the-art deep RL algorithms.

2.2 QHD: Hyperdimensional Q-learning

2.2.1 Overview

Figure 4.1 shows an overview of QHD supporting hyperdimensional reinforcement learning. In our RL task, there are two components (Agent and Environment) and three variables (Action, State and Reward). Figure 4.1(a) exploits a Cartpole example to illustrate these components and variables. The cartpole is the agent, and the space around it is the environment. The cart either goes right or left, which forms the action space of the agent. The position and velocity of the cart as well as the pole angle and angular velocity, form a state vector at each time step. For each step, the cart chooses an action, then the state of the cart and pole will be updated accordingly. Depending on the pole being balanced or not, the cart gets either a positive reward or a zero reward as the feedback from the environment. As shown in Figure 4.1, the interaction between the agent and environment form a loop in which the action taken based on the current state leads to the next state and reward. On the other hand, we have a trajectory composed of different states, rewards, and actions from each time step until the pole loses balance. The trajectory of each episode is saved in local memory for later learning. In Figure 4.1(b), we provide an overview of QHD algorithm guiding the agent in the decision-making process. In each time step, we map the state vector to a holographic hypervector using the HDC encoder. The hyperdimensional

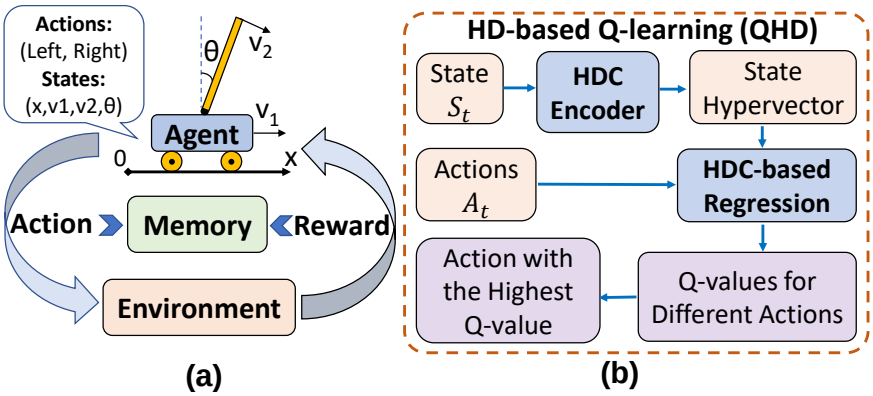


Figure 2.1: Overview of QHD reinforcement learning.

regression algorithm then predicts the Q-value for each possible action based on the input state hypervector. The final step of QHD chooses an action with the highest Q-value. In the remaining parts of this section, we introduce the HDC encoder, HDC regression, and the training of the QHD.

2.2.2 Hyperdimensional Encoding

In QHD algorithm, the learning process starts by mapping the current state vector from the original to high-dimensional space, i.e., hypervector encoding. One characteristic of this high-dimensional space is that we can create a large number of nearly orthogonal hypervectors through random sampling. Unlike the original HDC encoder that operates over binary or bipolar representation, our solution generates hypervectors with random exponential elements, e.g., $\vec{\mathcal{H}}_1$ belongs to $\{e^{i\theta} : \theta \in [-\pi, \pi]\}^D$.

For example, if we create two randomly generated hypervectors, $\vec{\mathcal{H}}_1$ and $\vec{\mathcal{H}}_2$, then we have the cosine similarity $\delta(\vec{\mathcal{H}}_1, \vec{\mathcal{H}}_2) = (\vec{\mathcal{H}}_1 \cdot \vec{\mathcal{H}}_2)/(|\vec{\mathcal{H}}_1||\vec{\mathcal{H}}_2|) \approx 0$. This allows the HDC encoder to express the input using a holographic hypervector through well-defined hypervector operations. In other words, the original information is evenly distributed across all hypervector elements. The advantage of holographic representation is that we can accumulate informa-

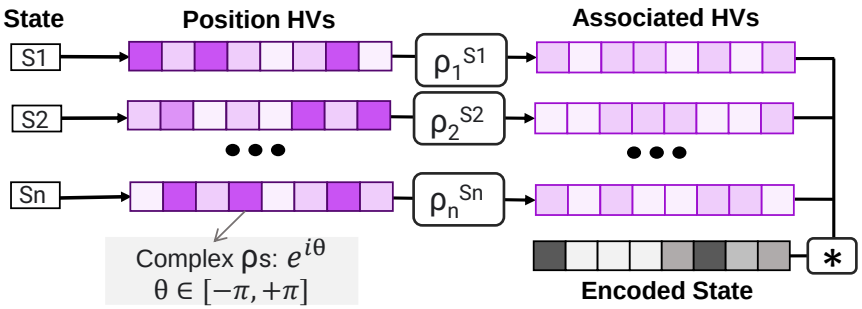


Figure 2.2: Hyperdimensional encoding with complex bases.

tion by combining two hypervectors, which is also reversible. Next, assuming the same two random hypervectors $\vec{\mathcal{H}}_1$, $\vec{\mathcal{H}}_2$ as above, we define the following HDC mathematics to model brain functionalities:

Bundling: This operation stands for component-wise addition of hypervectors. Bundling operation is the core of memorization for HDC models, in which the information from multiple hypervectors is saved into one single hypervector. The bundled hypervector is similar to every component hypervector, i.e., $\delta(\vec{\mathcal{H}}_1 + \vec{\mathcal{H}}_2, \vec{\mathcal{H}}_1) >> 0$. Thus, we represent sets using bundling operation.

Continuous Binding: The goal of binding is to associate items in hyperspace. Figure 4.2 shows the functionality of our continous binding. Assuming we have an n-element state vector $S_t = \{s_1, s_2, \ldots, s_n\}$ at time step t. For tasks with continuous state, we have $S \in \mathcal{R}^n$.

Our encoding generates a random exponential hypervector for each feature position $\{\vec{\mathcal{P}}_1, \vec{\mathcal{P}}_2, \cdots, \vec{\mathcal{P}}_n\}$ and then performs encoding as:

$$\vec{\mathcal{H}}_1 = \vec{\mathcal{P}}_1^{s_1} \vec{\mathcal{P}}_2^{s_2} \cdots \vec{\mathcal{P}}_n^{y_n}$$

We define $\vec{\mathcal{P}}_k^{s_k}$ to be the component-wise exponential of $\vec{\mathcal{P}}_k$. Bundling of several encoded states results in generating a memory hypervector: $\vec{\mathcal{R}} = \vec{\mathcal{H}}_1 + \vec{\mathcal{H}}_2 + \cdots + \vec{\mathcal{H}}_m$. Our solution is

extremely flexible to move, rotate, or decode any state in high-dimension space. For example, an encoded state can be moved by Δs to left by binding the high-dimensional representation to $\vec{\mathcal{P}}_k^{-\Delta s}$. In other words, $\vec{\mathcal{H}}\vec{\mathcal{P}}_1^{-\Delta s} = \vec{\mathcal{P}}_1^{(s_1-\Delta s)} \cdots \vec{\mathcal{P}}_n^{s_n}$. We exploit this adaptive and flexible representation to also decode an object from high-dimension. For instance, we can extract s_1 value by exploring Δs value until $s_1 - \Delta s = 0$, therefore, $\vec{\mathcal{P}}_1^{(s_1-\Delta s)} = \vec{\mathcal{P}}_1^0 = 1$. This can be detected by looking at the similarity of shifted vectors with the original encoded data. Another advantage of our mathematics is its capability to decode or extract information independent of the number of features. As a result, our solution can explore the space, decode information from desired features, and potentially provide analogy and reasoning.

To improve the existing HDC encoding, we introduce the idea of a continuous complex vector method. The idea is to generate a position hypervector, $\vec{\mathcal{P}}$, for each position/feature, which is correlated based on the distance between them. That is, the $\vec{\mathcal{P}}$s that represent nearby locations should be similar to each other, and the $\vec{\mathcal{P}}$s which represent far locations, should have almost orthogonal distribution: $\delta(\vec{\mathcal{P}}^x, \vec{\mathcal{P}}^y) = \sum_i e^{i\theta_i(y-x)} \approx \mathbb{E}(e^{i\theta(y-x)})_{p(\omega)} = k(x-y)$, where, $k(x-y)$ is a shift-invariant kernel, which is the Fourier transform of the probability distribution.

To evaluate the similarity for Norm encoding, we first consider the hypervectors $\rho^x\vec{\mathcal{R}}_1$ and $\rho^y\vec{\mathcal{R}}_2$, since they form the building blocks of all our encoding. Their similarity is given by

$$\delta(\vec{\mathcal{P}}^x\vec{\mathcal{R}}_1, \vec{\mathcal{P}}^y\vec{\mathcal{R}}_2) \approx \mathbb{E}(\vec{\mathcal{P}}_i^{y-x}\vec{\mathcal{R}}_{1,i}^{\dagger}\vec{\mathcal{R}}_{2,i}) \approx \delta(\vec{\mathcal{P}}^x, \vec{\mathcal{P}}^y)\delta(\vec{\mathcal{R}}_1, \vec{\mathcal{R}}_2)$$

The error in this approximation will be due to random covariances, which will tend to 0 as we increase the dimension D (which corresponds to the number of samples while taking the expectation values). Since both these similarities are real, we need to consider only their real parts. First term $\delta(\vec{\mathcal{R}}_1, \vec{\mathcal{R}}_2)$ can be written as $\frac{1}{D}\sum_{i=1}^{D}\cos(\alpha_i - \beta_i)$, where α_i and β_i are uniformly sampled from $[-\pi, \pi]$. If $\vec{\mathcal{R}}_1 = \vec{\mathcal{R}}_2$, then this similarity is 1. Otherwise, because

the angles matter only modulo 2π, we can consider $\alpha_i - \beta_i$ to be randomly sampled from $[-\pi, \pi]$. Assuming that D is large, we have: $\delta(\vec{\mathcal{P}}^x, \vec{\mathcal{P}}^y) \sim N\left(k(x-y), \frac{\sigma_{p(\omega)}}{\sqrt{D}}\right)$.

2.2.3 QHD Hyperdimensional Regression

We develop a regression model based on hyperdimensional mathematics. Recall Figure 4.1(b), the input to our hyperdimensional regression model is the action for evaluation and the encoded state hypervector. The output is the action-state value or Q-value corresponding to the input. Our regression consists of multiple model hypervectors $\{\vec{\mathcal{M}}_1, \vec{\mathcal{M}}_2, \ldots, \vec{\mathcal{M}}_n\}$, where n is the size of the action space. For evaluation of each action at time step t, we only select one of the model hypervectors $\vec{\mathcal{M}}_A$ that corresponds to a certain action A, and the regression is operated on the current-step state hypervector $\vec{\mathcal{S}}_t$. These model hypervectors are initialized to all zero elements and have the same dimensionality as the encoded state hypervector, i.e., $\vec{\mathcal{M}}_A \in \{0\}^{\mathcal{D}}$. Even though RL is not a typical supervised learning task, the regression part of it trains under supervision. In the context of QHD, the true value is given by the ideal Q-function, and we use hyperdimensional regression to approximately calculate the Q-value. We explain the ground truth for Q-value in Section 2.2.4 when we introduce QHD. On the other hand, the approximated Q-value for action A is given through the dot product between the model hypervector and the encoded state hypervector: $q_{pred} = \vec{\mathcal{M}}_A \cdot \vec{\mathcal{S}}^T$, where $\vec{\mathcal{S}}^T$ is a conjugate of the encoded query with complex elements. As for the regression model update, we use the error between q_{pred} and q_{true} (ground truth). We either add or subtract a portion of the state hypervector to the model, weighted by the regression error.

$$\vec{\mathcal{M}}_A = \vec{\mathcal{M}}_A + \beta(q_{true} - q_{pred}) \times \vec{\mathcal{S}}$$

This equation ensures that the model gets updated more aggressively for higher prediction error rates ($q_{true} - q_{pred} \gg 0$). In order to have a smooth training process, we control the

model update speed by multiplying a learning rate β. Thus, we gradually update the regression model with iterative training, and the approximation of Q-values becomes more accurate. The lightweight operations in our regression design, such as component-wise addition, contribute to the fast learning process for QHD.

2.2.4 Hyperdimensional Value-based Reinforcement Learning

In this section, we present the details for our QHD, a hyperdimensional Q-learning algorithm. We start our introduction with how agents with QHD make decisions at each time step. Current Q-learning methods, such as DQN, are value-based RL, meaning they do not directly learn a policy. Instead, they apply an indirect policy backed by the value function. In QHD, we use a greedy policy that prefers actions with higher Q-values. However, it is crucial to balance the exploration of the environment and the exploitation of the learned model. For example, always selecting actions with the highest Q-value will easily result in a sub-optimal result that gets stuck in a local minimum. This is because agents have not explored enough, and their decisions are short-sighted. On the other hand, if agents spend too much time trying new actions instead of following the learned model, we again have lower quality results. Therefore, we combine a random exploration strategy with the greedy policy, i.e., ϵ-decay policy. Assuming the action space $\mathcal{A}$ and time step t:

$$A_t = \begin{cases} \text{random action } A \in \mathcal{A}, & \text{with probability } \epsilon \\ argmax_{A \in \mathcal{A}} Q(S_t, A), & \text{with probability } 1 - \epsilon \end{cases}$$

The probability of selecting random actions will gradually drop after the agent explores and learns for several episodes. In experiments, we use a rate of changing ϵ-decay less than 1; this ensures that QHD agents start to trust their learned model more and lessen the importance of exploration. In the equation above, $Q(S_t, A)$ is a hyperdimensional regression model that

returns approximated Q-values for input action-state pairs. Once an action A_t is chosen by QHD, the agent interacts with the environment. We then obtain the new state S_{t+1} for the agent and the feedback reward R_t from the environment. At the next time step $t+1$, QHD selects another action A_{t+1} according to the ϵ-decay policy based on the updated state. This chain of actions and feedbacks form a trajectory or an episode until some termination conditions are met. To train an RL algorithm, these episodes or past experiences are usually saved to local memory as training samples. More specifically, we save a tuple of four elements for each step: (S_t, A_t, R_t, S_{t+1}).

In DQN, the RL training process and parameter update are based on DNN back-propagation, while the training in QHD utilizes more efficient hypervector operations. The regression model in QHD is trained at the end of each time step after saving current information to the local memory. As in the DQN training process, we apply a strategy called experience reply. Unlike offline supervised learning, where the training dataset is fixed, it is common to have a stream of training samples in RL. Thus, it is important to refresh the memory of past learning experiences while processing the new inputs. However, in ideal settings, i.e., the capacity of local memory is unlimited, the number of training samples grows so large in the end that it becomes time-consuming to train on the full dataset. Thus, the experience replays in QHD samples a training batch from the local memory and uses it for the regression model update. The training batch includes multiple tuples of past experiences.

[t!]

Episode i Time step t Get current state vector S_t ϵ-greedy algorithm for choosing action A_t Get reward R_t and new state S_{t+1} Record $\{S_t, A_t, R_t, S_{t+1}\}$ to memory Call function **TrainQHD** for QHD training Exploration rate decays After τ episodes Copy QHD model $\mathcal{Q}$ to delayed QHD model $\mathcal{Q}'$ TrainQHD Sample an experience batch E from memory $\{S_t, A_t, R_t, S_{t+1}\}$ in E Calculate predicted q-value q_{t_pred} using (4.2) Calculate true q-value q_{t_true} using (4.1) Update hyperdimensional regression model $\mathcal{Q}$ with (4.3)

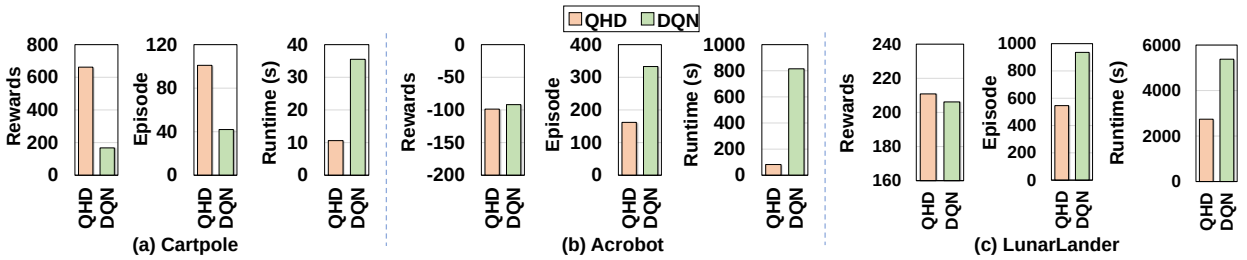


Figure 2.3: Performance comparison between DQN and QHD: In (a), the Cartpole reward is averaged over the last 50 episodes; the number of episodes and runtime are recorded when the episodic reward surpasses 200. In (b), the Acrobot reward is averaged over the last 100 episodes; the rest is recorded when the past-100-episode average reward reaches -120. In (c), the LunarLander reward is averaged over the last 100 episodes; the rest is recorded when the past-100-episode average reward reaches 200.

Now assume we sample a one-step experience tuple from past trajectories to train our QHD, i.e., (S_t, A_t, R_t, S_{t+1}). In Section 2.2.3, we introduce the regression model update based on the approximation error. We first encode the input state S_t to the hypervector $\vec{\mathcal{S}}_t$ and the predicted value q_{t_pred} is simply calculated as:

$$q_{t\_pred} = \mathcal{Q}(S_t, A_t) = \vec{\mathcal{S}}_t^T \cdot \vec{\mathcal{M}}_{A_t} \tag{2.1}$$

For supervised regression training, we need a ground truth q_{t_true}. We cannot directly obtain the true Q-value because RL is not typical supervised learning. For time step t, the feedback is the one-step reward R_t while the Q-value is the expectation of accumulated rewards. The method to connect these two values is called the Bellman Equation or Dynamic Programming Equation [9]. Since most RL tasks can be viewed as a Markov Decision Process (MDP), the Bellman equation gives a recursive expression for the Q-value at step t, the expected sum of current rewards, and the Q-value for step $t+1$. To learn an optimal Q-function, we use the Bellman optimality equation as shown below:

$$q_{t\_true} = R_t + \gamma max_A \mathcal{Q}'(S_{t+1}, A) \tag{2.2}$$

Recall that our objective in QHD is to achieve optimal policy and maximize the accumulated rewards within one episode. The Bellman optimality equation states that, in order to achieve optimal results for the whole task, we need to optimize each sub-task. Thus, the true value q_{t_true} is the sum of R_t and the max next-step Q-value. Instead of using model $\mathcal{Q}$ to calculate the maximum next-step Q-value, we use a delayed model $\mathcal{Q}'$ which gets updated periodically using parameters in $\mathcal{Q}$. This method is called Double Q-learning [34]; it stabilizes the learning process and avoids the overestimation of Q-value caused by the maximization in the Bellman equation. We also include a reward decay term γ that adjusts the effect of future rewards on the current step Q-value. If γ is close to 1, QHD takes more long-term effects into consideration, while when γ is close to 0, the model considers mostly the effect of immediate rewards and thus being short-sighted. It is worth noting that a smaller γ is not necessarily bad because some tasks do not have a long-term effect.

After obtaining the predicted Q-value and true Q-value, we perform regression model update. We update the model corresponding to the action taken, using the regression error $q_{t\_true} - q_{t\_pred}$ and the encoded state hypervector. The learning rate is β.

$$\vec{\mathcal{M}}_{A_t} = \vec{\mathcal{M}}_{A_t} + \beta(q_{t\_true} - q_{t\_pred}) \times \vec{\mathcal{S}}_t \tag{2.3}$$

In Algorithm 2.2.4, we conclude how QHD learns in an episodic RL environment using the equations and procedures introduced above.

2.3 Experimental Result

2.3.1 Experiment Settings

We implement our QHD algorithm on CPU hardware platform Intel Core-i7 10700 using Python. We validate the functionality of QHD with multiple control tasks in the OpenAI Gym [12] and a resource management task. For the control tasks, we selected Cartpole, Acrobot and LunarLander. The objective of Cartpole is to keep the balance of an uprising pole that loosely connects to a moving cart. Acrobot is a system with two links and two joints, where force is applied to the joint between the two links. The objective of Acrobot is to swing the lower part of the hanging link until it reaches the target height. The target in the LunarLander task if to safely land the moon vehicle on the uneven ground as fast as possible by controlling three engines. For comparison, we use the DQN algorithm for the same tasks in our evaluation. In the following subsections, we compare these two methods' learning performance and efficiency in all tasks. We run both methods for 200 episodes for the Cartpole task, 500 episodes for Acrobot and 1000 episodes for LunarLander. The maximum number of steps for each episode is 1000 for Cartpole/LunarLander and 500 for Acrobot.

The regression model we used in QHD has dimensionality $\mathcal{D} = 6000$ unless stated otherwise. The DQN is powered by a neural network with two hidden layers. The first layer has 128 neurons, and the second one has 256; except in the LunarLander task, where we use 64 neurons for the first layer and 128 for the second layer. The experience replay is enabled for model training in both methods, and we assume unlimited memory for rewards and runtime comparison. We select different parameters for sampling training batches to ensure the best learning quality for both methods. The QHD training batch size is 4 for Acrobot/Cartpole and 10 for LunarLander, and the DQN training batch size is 64 for all tasks. Rewards and runtime results for both methods are averaged over multiple trials.

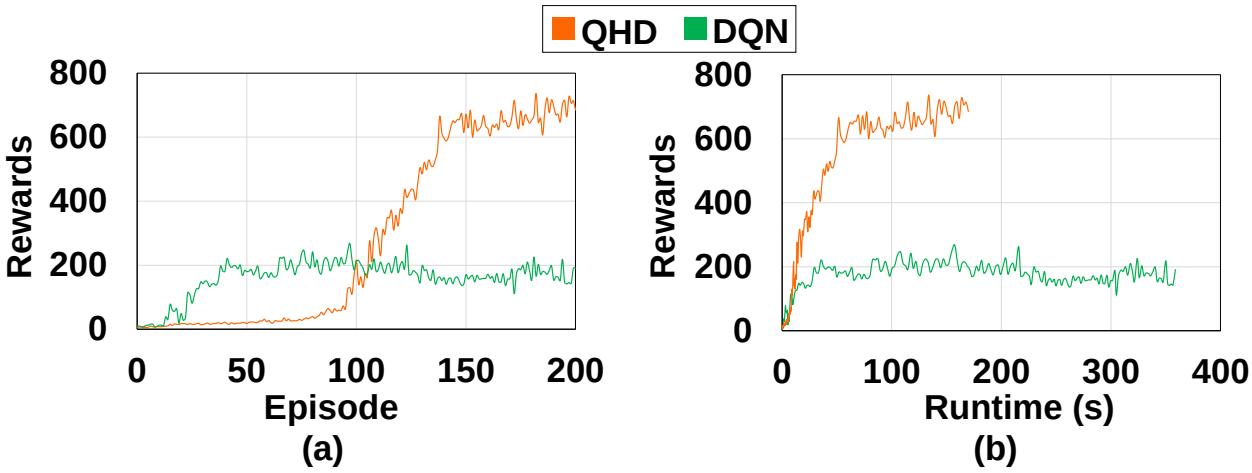


Figure 2.4: QHD and DQN Cartpole rewards comparison: In (a), average rewards trajectory is plot with episode index; (b) presents the trajectory with runtime as x-axis.

2.3.2 RL Rewards & Runtime comparison

Figure 2.3 compares the performance of DQN and QHD over three popular OpenAI control tasks. The bar graphs compare the final rewards achieved by both methods and the runtime needed for reaching preset goals. We set the goal of Cartpole as keeping balance for more than 200 steps, i.e., the episodic reward is larger than 200. As for Acrobot, the goal is to reach the target height within 120 steps, which means the reward is larger than -120. The goal in LunarLander task is 200 rewards. In this figure, we use slightly different criteria for achieving goals in three tasks. For Cartpole, we focus on the immediate episodic reward in runtime comparison because the DQN method cannot stabilize above 200 rewards. For Acrobot and LunarLander, we collect the average reward over the past 100 episodes as the standard. As shown in Figure 2.3(a) and 2.4, QHD achieves significantly higher final rewards in Cartpole compared to DQN. Within 200 episodes, QHD provides an averaged episodic reward over 660, which is nearly 4× higher than DQN. Note that QHD may need a higher number of episodes than DQN. For example, QHD reaches 200 rewards after 101 episodes while DQN only needs 42 episodes. However, considering the total execution time (shown in Figure 2.4b), QHD is significantly faster than DQN. This efficiency comes from the lightweight nature of QHD.

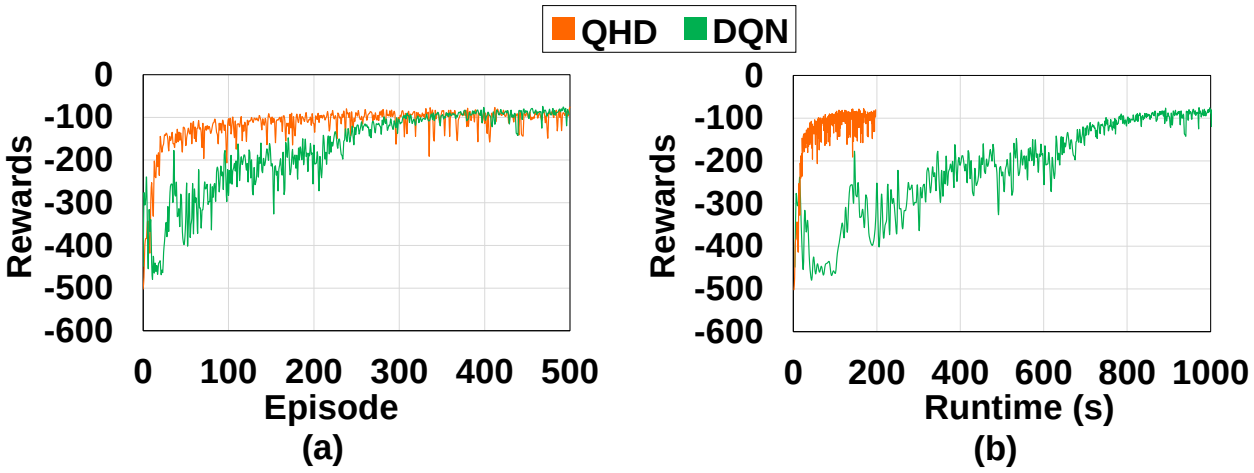


Figure 2.5: QHD and DQN Acrobot rewards comparison with both episode index and runtime index.

We also present the result comparison for the Acrobot task in Figure 2.3(b) and 2.5. As for the quality of learning, QHD achieves similar results compared to DQN. The averaged reward of QHD is slightly lower than that of DQN due to fluctuation at the end of the trial. However, our QHD provides significantly better learning efficiency compared to DQN. QHD is about 10× faster than DQN in terms of runtime and uses 2× fewer number of episodes. In the figure, we show that our QHD converges to the goal much faster at the beginning of the learning process. The ability of QHD to rapidly capture patterns in the environment and accurately memorize past experience is very appealing to RL tasks.

As for the LunarLander task, we compare the RL performance and runtime in Figure 2.3. It shows that, comparing to DQN, our QHD provides significantly higher efficiency in terms of both number of episodes and actual runtime while also achieving slightly higher rewards. QHD achieves the goal nearly 400 episodes earlier than DQN and about 2600 seconds (2 times) faster in runtime.

2.3.3 Evaluate the effect of training batch size

As we mentioned in 2.2.4, both our QHD and DQN relies on experience replay that periodically provides the learning agent with the past experience. Since the memory is assumed to be infinite and the reinforcement learning process can be extended to infinite episodes, we need to sample the training dataset from the large memory with preset batch size. This parameter is rather crucial because it controls how much past experience is available for the agent to learn from, thereby deeply influencing the learning quality. Usually, a larger batch size provides the agent with a better view of the environment and prevents it from forgetting past experience. However, increasing the number of training samples also brings greater computation and communication costs, which is not ideal for efficient learning at the edge. Our QHD, on the other hand, aims to fully utilize the provided training samples at each step; and we observe from the following results that it relieves the requirement for large batch sizes and saves the runtime for RL training.

In Figure 2.6, we explore the effect of replay batch size on both methods using the Acrobot task. Figure 2.6a compares the average rewards for the last 100 episodes, and it is clear that our QHD performs significantly better than DQN for different batch size settings, especially for smaller batch sizes. For example, when we only provide two random samples for each model training step, QHD can still achieve the goal with an average of -102.9 rewards. On the other hand, DQN performs poorly with the reward of -480.9, which is close to the lowest possible reward of -500. This means that DQN fails to learn the Acrobot task, and it does not efficiently utilize the limited available training samples. We also observe that increasing the batch size by a large amount from 2 to 30 helps DQN achieves better average rewards of -215.9. However, it is still unable to reach the goal of -120 rewards even with 15× larger batch size than the minimum setting of QHD. When we increase the batch size for QHD, the figure shows a steady increase in average rewards. When the batch size of QHD is set to 30, it achieves higher rewards than the DQN results shown in the last section with a batch

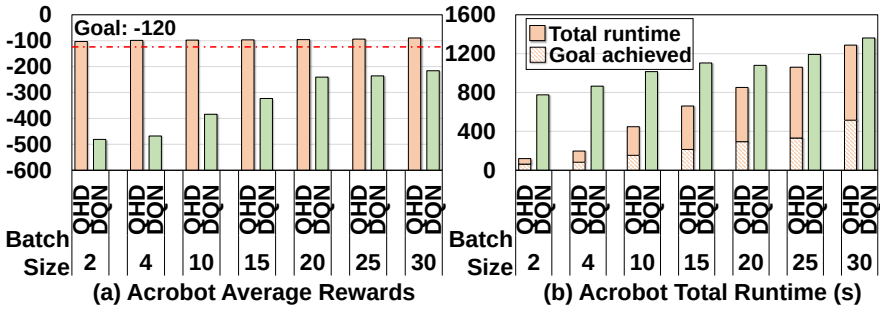


Figure 2.6: Explore the effect of batch size on both DQN and QHD using Acrobot task: In (a), we record average rewards of the last 100 episodes. In (b), we record the total runtime for 500 episodes and collect the goal-achieved runtime when QHD reaches the average reward of -120.

of 64.

Apart from better performance, our QHD also provides higher efficiency. In Figure 2.6(b), We compare the total runtime of QHD and DQN with different batch sizes. For QHD, we provide both the runtime for 500 episodes and the runtime when the goal is achieved. For DQN, only total runtime is provided because DQN cannot achieve the goal with small batch sizes from 2 to 30. Our QHD is constantly faster than DQN for these batch size settings. With the batch size of 2 (15), our QHD is about 6.5× (1.7×) faster than DQN in terms of total runtime. Focusing on the actual runtime when achieving the target, QHD shows an even larger improvement, e.g., the speedup is about 12.3× (2.6×) with the batch size of 2 (30).

2.3.4 QHD vs. DQN with Limited Memory

In the above sections, we assume the local memory for RL experience replay has infinite capacity, i.e., the agent has access to all previous experience during the training. Even though we use sampling to select training sets, each data point in memory can possibly be used for training. However, in practical implementations of RL algorithms, the memory capacity is limited due to energy and space budgets, especially in the low-power edge environment.

Thus, in this section, we evaluate the performance of our QHD with limited memory size and compare it to the DQN results.

In Figure 2.7(a), we present the average reward achieved by both methods under different memory capacity limitations. The reward is averaged over the last 100 episodes. When collecting these results, we fix the training batch size; the batch size is 4 for QHD and 64 for DQN. We vary the memory size from 1 to 1024 for QHD, and the minimum memory size setting for DQN is 64 because of its higher requirement on training batch size. The figure shows that DQN performs poorly when the memory size is 64 and 128, with an average reward of -500. When we increase the memory size to at least 256, i.e., 4× of the batch size, DQN starts to learn but ends without reaching the -120 target. However, our QHD can reach that goal even with a memory size as large as its batch size. These results show that QHD can perform RL tasks with online learning, i.e., a tiny local memory.

We also take one step further to explore the QHD capability of real-time learning. We set both the batch and memory sizes to 1, which means the agent will learn based on only the current sample, without any access to previous experiences. To handle this real-time learning setting, the RL algorithm should have the ability to fully memorize the current sample and efficiently utilize this one-time training data point. We use DQN with 256 memory size and 64 batch size as an online-learning comparison since DQN does not learn with a real-time setting. As shown in Figure 2.7(b), even with larger memory size and batch size, DQN achieves significantly lower rewards (-345.4). For a 500-episode training, our QHD achieves average rewards of -113.7 using 83 seconds, which leads to a 34.6× speedup in total runtime.

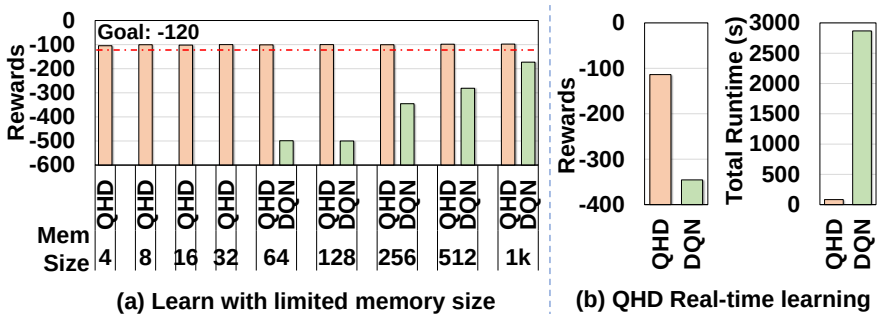


Figure 2.7: QHD learning efficient with tiny local memory.

2.4 Conclusion

We propose a novel lightweight RL algorithm based on brain-inspired hyperdimensional computing. QHD utilizes hardware-friendly hyperdimensional operations for high-quality Q-value approximation and self-learning agent training. Our evaluation on several control tasks shows that QHD provides significantly higher efficiency and better learning quality than existing DQN approaches. Our solution also supports small experience replay batch sizes that provide 12.3× speedup compared to DQN while ensuring minimal quality loss.

Chapter 3

HyperDimensional Hybrid Learning for End-Edge-Cloud Computing

3.1 Introduction

With the rise of distributed systems and smart devices, Resource Management is an increasingly prevalent orchestration problem to optimize. When these systems involve computationally expensive services, such as Deep Learning (DL) kernels, resource-constrained end-user devices heavily rely on cloud infrastructures to execute these tasks [88]. However, this is not a robust solution since network conditions are prone to inconsistencies and affect the real-time of providing these services [54]. To enable a low-latency infrastructure, the introduction of edge computing has brought compute capacity closer to end-user devices [92]. Furthermore, the collaborative end-edge-cloud architecture has allowed instantaneous computational offloading to higher equipped cloud and edge nodes instead of the resource-constrained end-user device [64].

Resource management is a multi-dimensional optimization problem since it requires finding

the appropriate device to offload the computation while minimizing the latency in aberrant network conditions [64]. Configuring this system is extremely complex and requires machine learning algorithms run-time decision-making capabilities. Reinforcement learning (RL) algorithms have shown great potential in solving dynamic resource management and scheduling problems [93, 91]. RL methods are developed to interact with the environment by trying different actions and reinforcing those that provide higher rewards [27]. In the case of optimizing the network resources for an end-edge-cloud system containing numerous low-powered devices, the use of lightweight and robust artificially intelligent models is critical for performance. The two classes of RL algorithms are model-free and model-based learning. The former includes popular algorithms, such as Q-Learning, and involves an agent directly interacting with its environment to accumulate experience, which is data that is then used to train the model. A notable trade-off in using these algorithms include the costly exploratory interactions with the network environment [56], which is intensified if the state and action spaces are expansive. In the application of distributed computer systems, this is extremely impractical since these executions are expensive and result in higher latency and resource consumption [56]. The second class of RL algorithms, model-based learning, avoids this costly training time since it trains an agent by simulating the agent's environment in order to predict the system behavior and enable real-time learning. The advantage of doing so includes improved generalization and a significant reduction in the number of system execution runs to realize the optimal solution. However, these algorithms are prone to model-bias, which consequently leads to trained models often reaching sub-optimal results.

Hybrid Learning is an approach that utilizes both of these RL algorithms to exploit their strengths and lessen their respective disadvantages, e.g., costly real-time exploration and model-bias. To mitigate the trial-and-error phase of model-free learning, a model-based RL system model is incorporated to aid the model-free agent by producing data samples from the simulated environment, which consequently reduces the number of direct interactions with the system to enable faster real-time learning. Despite the success, the existing RL

algorithms face several difficulties in dealing with large state spaces, which are common in resource management problems. In addition, the state-of-the-art RL algorithms are based on deep neural network models. Therefore, they have the following computational challenges: (1) demand unreasonable resources to train as neural network models rely on costly back-propagation and gradient-based methods [52], (3) are extremely sensitive to noise in data, network, or underlying hardware, and (4) lack human-like cognitive support for long-term memorization and transparency.

To address the above listed challenges, recent learning algorithms aim to model the human brain more closely. Brain-inspired Hyperdimensional Computing (HDC) is gaining traction for its impressive learning ability, lightweight hardware implementation, and computational efficiency [28, 76]. The origin of this field stems from neuroscience research on how the human brain, specifically the cerebellum cortex, processes information in high dimensions due to the brain's extensive brain circuitry [52]. HDC abstractly extends this concept as a learning paradigm by modeling and operating on high-dimensional data representations [52].

HDC achieves this by encoding input data into high-dimensional space, outputting what is called *hypervectors* and using well-defined HDC operators to train and execute inference tasks [28]. Each dimension of this hyperspace abstractly models a neuron's functionality in processing external stimuli [52]. The HDC operators are simple arithmetic operations over these dimensions that can be supported on devices with limited resources and computational power [45]. In addition, this class of algorithms is able to accomplish both classification and regression machine learning tasks [29, 69] and have been shown to achieve comparable results to neural networks offering higher learning quality, e.g, faster convergence, learning speed, and computational efficiency [28].

We utilize this hybrid learning approach for orchestrating an end-edge-cloud architecture for DL inference tasks and utilize an HDC version of the Deep Dyna-Q for the RL agent [83, 72]. As a result, our algorithm significantly speeds up training by reducing the number

of real-time interactions with the system and enables computationally efficient learning.

In this paper, we propose HDHL, a novel hyperdimensional hybrid learning model to optimize resource management applications. Our contributions in this paper are listed below:

- To the best of our knowledge, HDHL is the first hyperdimensional hybrid learning algorithm to successfully and efficiently orchestrate an edge-edge-cloud system. Compared to Deep Q-Learning, HDHL is magnitudes more computationally efficient in training the reinforcement learning agent compared to the state-of-the-art DQN algorithm.
- We also demonstrate HDHL's ability to produce higher quality learning with a faster convergence to the optimal result compared to the DQN algorithm.
- We also present the notable application of Hyperdimensional Q-Learning (QHD) to demonstrate its computationally efficient advantage over its Neural Network counterpart.

In this paper, we tackle the communication-computation co-optimization offloading orchestration problem to an end-edge-cloud network. We show that the novel HDHL algorithm is computationally more efficient than DQN by 21.0× for three devices and by 9.5× for four devices. In addition, we show HDHL's improved learning quality by the faster convergence to the near optimal result and the significant reduction to the number of direct interactions with the system environment by 4.8× for three devices and by 5.7× for four devices.

3.2 Network Architecture Overview

We introduce the problem by first laying out the collaborative End-Edge-Cloud (EEC) architecture. Our objective is to optimize the orchestration of an intelligent service, a Deep Learning inference application that is run periodically, with the constraint of minimizing the

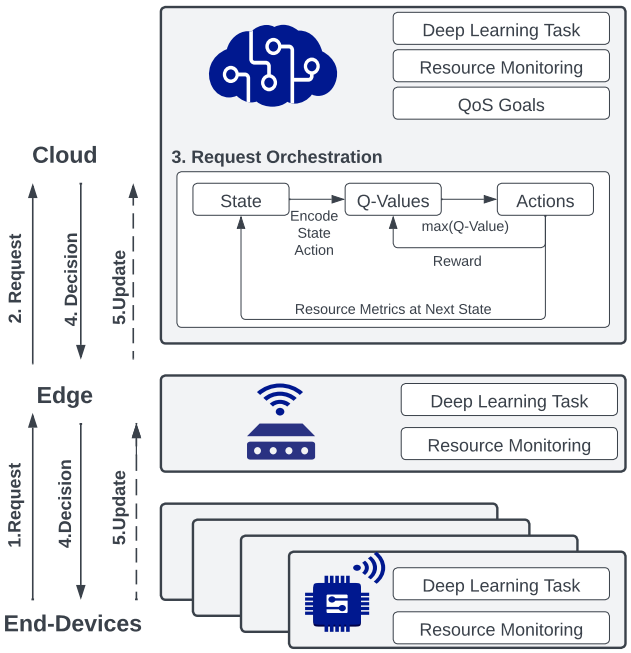


Figure 3.1: Multi-user Deep Learning inference orchestration framework.

latency and maintaining the accuracy threshold while also optimizing the various network resources. We use the following notation (S, E, C) to represent the components in the network: S refers to a user end-device, E denotes the edge device, and C is the cloud node that hosts the main RL agent i.e., the intelligent orchestrator, where the system's offloading decision-making occurs. In addition, each device's resource metrics are measured to inform the orchestrator of the network conditions. This includes the processor utilization P_i, the network connection status C_i between devices, and the available memory M_i for node i in the network.

The orchestration optimization component is the decision of whether the intelligent task needs to be offloaded to a node with a higher compute capacity i.e., the cloud or edge device, or to perform the task locally on an end-device. The metric that we aim to optimize is the comprehensive time of deciding which device to offload to, making the request to the service, and receiving the results; we refer to this as the *response time*, $T_{response}$. Using this notation, we define our objective as minimizing the average $T_{response}(o_i, m_k)$ of offloading

decision o_i and inference model m_k for the i^{th}node.

To learn the optimal configuration in automating the intelligent orchestration given the network conditions, we employ reinforcement learning to learn the environment. To accomplish this, an agent interacts with the environment to collect experience, which is then used to formulate a policy to learn the task at hand. In this problem, the agent is the intelligent orchestrator that executes offloading decisions. The ***State Space:*** is the system dynamics at a given time step, which are the available processing cores, network stability, and memory availability. The measures of each of these components impact the optimal offloading decisions. Furthermore, the ***Action Space:*** is comprised of the offloading decisions made at each time step. The size of the action space depends on the number of end-devices and the number of deep learning models stored in each node since each end-device either offloads the intelligent task to a high computational capacity node or executes the task locally, which requires a selection between the l models. To simplify the action space, we restrict the cloud and edge devices to only store one model.

Table 3.1: State Space Discrete Values

State	Discrete Values	Description
P^C	Nine discrete levels	Cloud CPU Utilization
M^C	Available, Busy	Cloud Memory Availability
C^{D_i}	Regular, Weak	Cloud Available Bandwidth
P^E	Nine discrete levels	Edge CPU Utilization
M^E	Available, Busy	Edge Memory Availability
C^E	Regular, Weak	Edge Available Bandwidth
P^{S_i}	Available, Busy	End-Node CPU Utilization
M^{S_i}	Available, Busy	End-Node Memory Availability
C^{S_i}	Regular, Weak	End-Node Available Bandwidth

To complete this reinforcement learning task definition, we define the ***Reward Function*** as the observed average response time of the agent's consequent offloading decisions given the network conditions at each time step. This is the feedback from the environment used

for optimizing the system's orchestration. To incorporate the accuracy threshold constraint, we penalize the agent if this constraint is violated; otherwise, the reward remains the average response time.

The EEC architecture is presented in Figure 3.1, along with the resource monitoring, service requests, and agent components. Each end-device include two software components: (i) the resource monitoring service which periodically collects the device system's parameters and broadcasts these metrics to the cloud and edge nodes; and (ii) the collection of deep learning models, which are trained for image classification. The cloud and edge nodes also contain these two software components. In addition, the cloud hosts the orchestrator, which is responsible for the computation offloading decision for each end-device in the network, as well as the model selection if the local offloading decision is chosen. Hence, this is the centralized reinforcement learning agent that will be trained to learn the optimal configuration. The agent utilizes the information communicated from the edge and end-devices, including the response time or the feedback from the environment to learn an optimal policy.

Furthermore, there is the Quality of Service (QoS) Goal, which is the accuracy constraint that must be upheld. Figure 3.1 shows the sequence of steps in this procedure: (1) the end-device makes a periodic request for the deep learning inference service, (2) the request is passed through the edge layer and (3) handed off to the cloud node for the orchestrator to process, (4) the offloading decision is made and delegates the task to the selected device, (5) once the inference task is completed and the response time is reported, each node updates the agent on network conditions i.e., the available resource information.

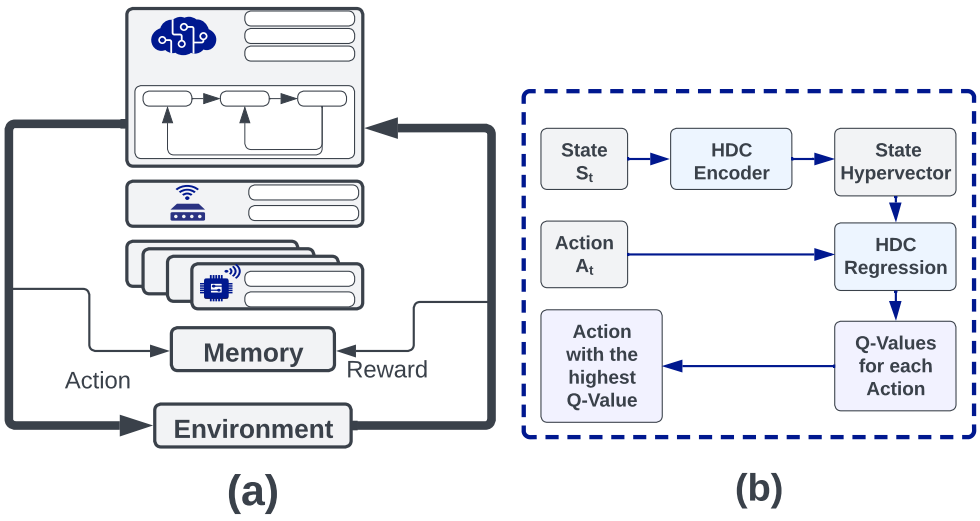


Figure 3.2: Overview of HDHL reinforcement learning framework.

3.3 Hyperdimensional Q-Learning

In this section, we detail the Hyperdimensional Reinforcement Learning algorithm employed to learn the network's optimal orchestration. The agent uses Q-Learning, a value-based Reinforcement Learning (RL) algorithm, which learns to approximate the expectation of future rewards, referred to as the *Q-Function*.

As is known with RL algorithms, when learning new environments, the agent needs to learn undiscovered areas of the state space while also utilizing past experience to learn the best action to take at any given state. Exclusively using one of these initiatives results in one of the two extremes: by only taking random actions, the model never learns the Q-Function, and on the other, the model may converge to a local maxima, resulting in a sub-optimal configuration. To balance this, we employ the Epsilon-Greedy strategy, which initializes the RL agent to prioritize exploration exclusively. However, throughout training, the agent relies more heavily on the learned Q-function until epsilon completely decays, enabling pure exploitation of the Q-Function. This is done by initializing $\epsilon = 1$ and applying a decay rate for epsilon to converge to zero. At each time step in training, if a randomly generated

number is less than ϵ, then a random action is selected. If not, the model selects the action with the best Q-value, as shown below:

$$A_t = \begin{cases} \text{random action } A \in \mathcal{A}, & \text{with probability } \epsilon \\ argmax_{A \in \mathcal{A}} Q(S_t, A), & \text{with probability } 1 - \epsilon \end{cases}$$

assuming an action space $\mathcal{A}$ and time step t.

As alluded to, the agent orchestrating the optimal network offloading decisions is implemented using HDC-based regression. Similar to Deep Q-Learning, this algorithm is powered by a machine learning model, rather than relying on a Q-table (i.e., tabular Q-Learning), to predict Q-values; however, instead of using a neural network to predict these values, the agent utilizes an HDC regression model. This is achieved by mapping the state of the network into high-dimensional space using a Radial Basis Function (RBF) encoder, as is used in other HDC algorithms [37]. The HDC model contains n class hypervectors $\{\vec{\mathcal{M}}_0, \vec{\mathcal{M}}_1, ...\vec{\mathcal{M}}_n\}$, where n is the size of the action space. In other words, each of these hypervectors corresponds to an action encoded in the hyperspace. Following the Q-Learning algorithm, in state S_t, we choose an action by selecting the one that outputs the highest Q-value. In the HDC version, we do this by binding the encoded state hypervector $\vec{\mathcal{S}}_t$ with each class hypervector $\vec{\mathcal{M}}_i$ to output the corresponding Q-value for each state-action pair. The agent selects the action with the highest Q-Value, A_t, and interacts with the environment, collecting feedback, R_t, and observing the next state in this transition, S_{t+1}. The algorithm advances to the next time step $t+1$, and continues to iterate through this algorithm until the episode terminates.

The class hypervectors of the HDC model are initialized to zero, but are updated over episodes as more experience is accumulated. After each time step t, the state, the selected action, the reward, and the next state are stored in a replay buffer as a tuple $\{S_t, A_t, R_t, S_{t+1}\}$. This experience replay strategy is used to collect additional data for training the agent in an online approach. After sufficient data samples are collected, a batch of these tuples are

sampled from the experience buffer to update the HDC regression model. Unlike classical supervised machine learning, reinforcement learning algorithms do not have data labeled with the "ground truth" readily available. Instead, the Bellman Equation or Dynamic Programming Equation [9], which is a recursive expression for the Q-value at step t, is used to label the data. We use this to define the optimal Q-function

$$q_{t\_true} = R_t + \gamma max_A \mathcal{Q}'(S_{t+1}, A)$$

and to maximize the accumulated rewards in an episode. In order to do so, the Bellman Equation states that each sub-task must be optimized to achieve the optimal results for the entire task, which means q_{true} is the sum of R_t and the maximum Q-value for the next time step. As noted in the equation above, we use $\mathcal{Q}'$, the target model, since our approach uses the Double Q-Learning method in updating the model [34]. Compared to $\mathcal{Q}$, which is updated at every time step, $\mathcal{Q}'$ is updated periodically and stabilizes the learning process to avoid overestimating the Q-Value, a common consequence of maximizing the Bellman equation. Additionally the reward decay γ is included to adjust the weight on future or near-sighted rewards: a value of 1 puts a higher weight for the long-term rewards and a value close to 0 prioritizes the more immediate rewards.

Returning to the update step: for a given tuple $\{S_t, A_t, R_t, S_{t+1}\}$, we encode the state S_t into hyperdimensional space to give its corresponding hypervector $\vec{\mathcal{S}_t}$ and bind it with action A_t's corresponding class hypervector, M_{A_t}, from the HDC model

$$q_{t\_pred} = \mathcal{Q}(S_t, A_t) = \vec{\mathcal{S}_t^T} \cdot \vec{\mathcal{M}}_{A_t} \tag{3.1}$$

We take the difference between q_{pred} and q_{true} and use it to update the HDC model where α is the learning rate.

$$\vec{\mathcal{M}}_{A_t} = \vec{\mathcal{M}}_{A_t} + \alpha(q_{t\_true} - q_{t\_pred}) \times \vec{\mathcal{S}}_t \tag{3.2}$$

Over the duration of learning, the Q-Function gradually becomes more accurate in estimating the best actions across the state space.

To further improve the HDC Q-Learning algorithm, we incorporate model-based reinforcement learning to learn a system model and include a planning phase to leverage faster training. This approach is sectioned into three phases (1) the exploratory phase used for data collection, (2) the system model training phase, and (3) the planning phase which involves training the Q-Learning agent.

3.4 Hyperdimensional Computing Hybrid Learning

3.4.1 Hybrid Learning Algorithm

An issue that arises in applying the Q-Learning algorithm to this problem is working with an action space that is extremely large. This is a challenge since a large state space requires extensive exploration to learn the best configuration of actions. As described in the previous section, each end-device has $l + 2$ choices at any time point t: it either selects one of the l locally stored models or it offloads the task to either the edge or cloud devices. Given the number of end devices, the action space quickly grows (e.g., for a network configuration of n end-devices and l local models, the number of actions is $(l + 2)^n$). For instance, in a network with three devices, each storing eight models, the action space is 1000, but for a network with five devices, this number quickly swells to $100,000$. This is difficult for any RL algorithm to efficiently tackle since the length of exploration consumes much of the time

and cost to train the agent.

A hybrid approach to this problem is to reintroduce the objective to the agent with the inclusion of a planning phase that utilizes a system model. To mitigate the expensive exploration phase, we use a system model to simulate the environment by training it to learn the next state and response times given a state-action pair. This is a computationally efficient alternative to leverage since relying solely on direct interactions with the environment for training is expensive. However, this approach still includes the Q-Learning exploration phase since it collects the initial data samples necessary for training the system model. The system model includes two models: the first is the *Time Model*, an HDC regression model used to predict the response time of taking an action at a given state, and the second is the *State Model*, which is implemented using a collection of HDC classification models to predict the next observation space.

In the first phase of this hybrid learning approach, the QHD model is deployed to the environment to collect data to populate the replay buffer, buffer_{direct}, for the subsequent phase. The algorithm detailed in the previous section is used for action selection during this phase, which includes the Epsilon-Greedy Algorithm and the calculation of Q-Values using HDC. This first phase is used solely for data collection; thus, no updates are made to either of the target or policy models. The second phase utilizes this populated replay buffer to train the system model in predicting the network behaviors, which is then used in the planning phase. This third phase uses the system model's simulated data to further train the QHD agent.

3.4.2 System Model Design

As mentioned, we train a system model in order to learn the network behavior and simulate the environment in order to accumulate more data samples. The first component of the

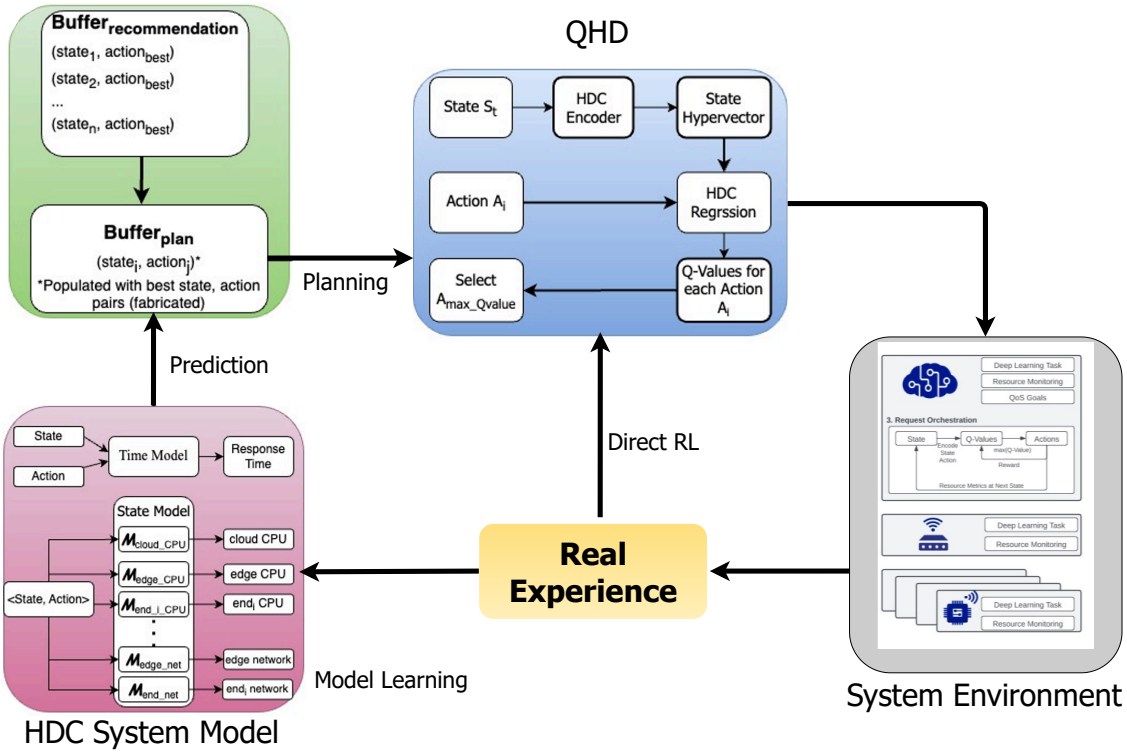


Figure 3.3: Hybrid Learning Architecture.

system model, the *Time Model*, predicts the reward e.g., response time, of selecting action A_t at state S_t. Since this is a regression model, it is quite similar to the QHD model in terms of its implementation; however, instead of using the regression model to output the Q-value of a state-action pair, the Time Model outputs the predicted response time of taking action A_t at a given state S_t. This is achieved by encoding the state S_t into hyperdimensional space and associating it to each of the model's class hypervectors $\{\vec{\mathcal{M}}_0, \vec{\mathcal{M}}_1, ...\vec{\mathcal{M}}_{N_{actions}}\}$, which outputs the predicted reward for selecting each action respectively. We sample tuples $\{S_t, A_t, R_t, S_{t+1}\}$ from the buffer$_{direct}$ to enable a supervised training of the model by updating it by the error in predicted response time, R'_t, as shown below

$$M_{A_t} = M_{A_t} + \alpha \times (R_t - R'_t) \times \vec{\mathcal{S}}_t \tag{3.3}$$

where α is the learning rate, M_{A_t} is the action's corresponding class hypervector, and $\vec{\mathcal{S}_t}$ is the state's encoded hypervector. We train this model to be leveraged in the planning phase of this approach.

The system model's second component is the State Model, which is trained to predict the next state S_{t+1} of the environment given action A_t at state S_t. This involves predicting each of the elements in the observation space, which includes each device's CPU utilization, the available memory, and available bandwidth, as elaborated in Table 1. An issue that arises in implementing this classification task is due to the large number of possible network configurations. To calculate the number of classes for classification involves exhaustively enumerating all combinations of each resource metric. Accounting for all these possible network states and translating this into a classification task would require an HDC model that contains $2^{2n} \cdot 10^2 \cdot 2^2$ class hypervectors for n end-devices. This implementation would be infeasible since each class hypervector contains thousands of elements; hence, it can be assumed that training such a model would perform far worse than the QHD model, making this hybrid learning approach entirely futile.

Instead, we break the classification into sub-tasks by training multiple HDC models, where each model is trained to only predict one of the observations in the state space. Although there is a trade-off in having to train $4 + 2 \cdot n_{devices}$ HDC models, this is a far better alternative since all, except the cloud and edge devices' CPU utilization levels, would require binary classification tasks. The input for each of these models are the concatenated state and action pairs, which are encoded using the RBF. This hypervector is then used to predict its observation feature using the cosine similarity between the encoded vector and each of the model's class hypervectors. Once each of these models output their respective predicted observation, the predictions are then concatenated to reconstruct the predicted state at the next time step, S_{t+1}. To update the model, we implement an adaptive iterative training approach, which updates the correct class hypervectors based on the cosine similarity of

the encoded state vector of the correct class in order to increase their similarity in hyperspace. Conversely, if the class is predicted incorrectly, we update the incorrectly predicted class hypervector in the negative direction to decrease this similarity measure of these two hypervectors. The equations for updating this model are shown below:

$$M_{correct} = M_{correct} + \alpha \times \delta(\vec{\mathcal{H}}_{<S_t,A_t>}, M_{correct}) \times \vec{\mathcal{H}}_{<S_t,A_t>}$$
$$M_{wrong} = M_{wrong} - \alpha \times \delta(\vec{\mathcal{H}}_{<S_t,A_t>}, M_{wrong}) \times \vec{\mathcal{H}}_{<S_t,A_t>}$$

where α is the learning rate, δ is the cosine similarity, $\vec{\mathcal{H}}_{<S_t,A_t>}$ is the encoded state-action pair hypervector, and $M_{correct}$ and M_{wrong} are the correctly and incorrectly predicted class hypervectors respectively. By training each of these HDC classification models separately, we are able to predict with high accuracy the overall next state of the network.

3.4.3 Planning Phase

The third phase is broken down into two parts: the first involves populating a replay buffer, buffer$_{recom}$, with the best recommended state-action pairs, which is then used for its subsequent counterpart of training and updating the QHD agent. To populate the buffer$_{recom}$, the trained system model from the previous phase is used to predict the next state and reward given any state-action pair. Specifically, for a given state, we utilize the Time Model to predict the action that yields the highest reward and use this selected action to predict the next state of the environment using the State Model. We use this collected data of the paired state and action with the predicted reward and next state values to populate a prioritized replay buffer, buffer$_{plan}$, which will be used later in this phase.

To tackle the challenge of exploring a large space, we also select a random neighboring action for each of the top action's adjacent action configurations to populate the buffer$_{plan}$ with. This enables a strategic exploration of the large state-action space. Furthermore, the

buffer stores exactly one state-action pair for each respective state. In the scenario where a state exists in the buffer_{plan} and the same state is populated with a new action, the buffer will replace the previously matched action with the new one along with its corresponding predicted reward and next state. By employing these two models in tandem, we collect additional data samples without needing to directly interact with the environment, saving time and computational overhead.

The next step in planning involves training the QHD agent further, but by instead sampling from the buffer_{plan}. By doing so, only the states that are most likely to yield the highest rewards are strategically used to train the agent. It uses the QHD model to select the best action for each state in the sampled buffer and updates the QHD model with the correct and incorrect action values. After iterating through this step a number of times, the target model is updated at the end of the epoch. Additionally, this step includes updating the policy QHD model with the Q-value errors. This final phase of the hybrid learning concludes and returns to the exploratory first phase to continue iterating through this algorithm.

3.5 Evaluation

3.5.1 Experimental Setup

For the DL orchestration framework, we first consider the MobileNet image classification application as the benchmark [39]. We consider eight different MobileNet models [39] ($d0$ through $d7$) with varying levels of accuracy and performance. Table 3.3 summarizes the MobileNet models $d0$ through $d7$, with each model having different response times and accuracy levels. Our framework supports up to four end-device nodes, networked with edge and cloud layers.

Table 3.2: Experiment Environment Setup. *R* and *W* represent *Regular* and *Weak* network condition, respectively.

Exp	S1	S2	S3	S4	E
A	R	R	R	R	R
B	R	W	R	W	W
C	W	W	W	R	R
D	W	W	W	W	W

Each end-user device is connected to a single edge device, and can request a DL inference service to the cloud layer. The cloud layer hosts the orchestrator that contains the RL agent, which handles the inference orchestration. Upon each service request, the RL agent is invoked to determine: (i) where the request should be processed and (ii) what DL model should be executed for the corresponding request. The platform consists of five AWS a1.medium instances with single ARM-core (as the end-device nodes), connected to an AWS a1.large instance with two CPUs (as edge device), and an AWS a1.xlarge instance with four CPUs (as the cloud node). In this work, we conduct experiments under four unique scenarios with varying network conditions (See Table 4.2). Each scenario represents a combination of regular (R) and weak (W) network signal strength over four end-user devices (S1-S4) and 1 edge device (E). The regular network has no transmission delay, while we add 20ms delay to all outgoing packets for the weak connection.

3.6 Experimental Results

We demonstrate the efficacy of HDHL for RL-based orchestration compared with the state-of-the-art Deep Reinforcement Learning algorithm. We evaluate the performance of the agent on a multi-user end-edge-cloud framework. We also report the overhead incurred by the agent learning process compared with state-of-the-art.

Table 3.3: MobileNet Models [39]

#	Model	Million MACs	Type	Accuracy (%)
$d0$	MobileNetV1-224 1.0	569	FP32	89.9
$d1$	MobileNetV1-224 0.75	317	FP32	88.2
$d2$	MobileNetV1-224 0.5	150	FP32	84.9
$d3$	MobileNetV1-224 0.25	41	FP32	74.2
$d4$	MobileNetV1-224 1.0	569	Int8	88.9
$d5$	MobileNetV1-224 0.75	317	Int8	87.0
$d6$	MobileNetV1-224 0.5	150	Int8	83.2
$d7$	MobileNetV1-224 0.25	41	Int8	72.8

3.6.1 Performance

We demonstrate the proposed agent's performance in finding optimal orchestration decision-making. At design time, we determine the true optimal configuration for orchestrating a DL task under any given condition of workload and number of active users using a brute-force search. This is used for comparing the orchestration decision made by our agent to the true optimal decisions. Our proposed algorithm yields a 100% prediction accuracy in comparison with the true optimal decision. Table 4.2 presents the experimental scenarios A-D with different combinations of regular (R) and weak (W) networks. For each end-user node ($S1$-$S4$) within each experimental scenario (A-D), we present the orchestration decision made by the proposed agent. The decision consists of the choice of execution node and inference model ($d0$-$d7$). Table 3.4 shows the average response time (ART, in ms) and average accuracy (AA, in %), along with the constraint (Cnst) on minimum accuracy requirements. Our proposed agent explores the Pareto-optimal space of offloading choice and model selection to minimize the response time within the accuracy requirement. For instance, in the experimental scenario A: maintaining Max accuracy threshold results in

Table 3.4: Results of the framework for different accuracy constraints for different experiments (four users). Cnst, ART, and AA represent constraint, average response time, and average accuracy, respectively. For example, in Exp-D with 89% average accuracy constraint, our framework orchestrates $S1$, $S2$, $S4$ to execute DL inference using model $d4$ locally and $S3$ offloads execution using model $d0$ to the cloud.

		End-node Devices					
Exp	**Cnst**	**S1**	**S2**	**S3**	**S4**	**ART (ms)**	**AA (%)**
A	Min	$d7, L$	$d7, L$	$d7, L$	$d7, L$	72.08	72.80
	80%	$d6, L$	$d6, L$	$d7, L$	$d6, L$	99.15	80.60
	85%	$d6, L$	$d5, L$	$d6, L$	$d5, L$	137.22	85.10
	89%	$d4, L$	$d4, L$	$d8, L$	$d4, L$	276.01	89.15
	Max	$d0, L$	$d0, L$	$d0, E$	$d0, C$	410.35	89.90
B	Min	$d6, L$	$d7, L$	$d7, L$	$d7, L$	120.20	75.50
	80%	$d6, L$	$d6, L$	$d6, L$	$d7, L$	139.64	80.60
	85%	$d6, L$	$d6, L$	$d5, L$	$d5, L$	178.42	85.10
	89%	$d4, L$	$d4, L$	$d8, E$	$d4, L$	317.68	89.15
	Max	$d0, L$	$d0, C$	$d0, E$	$d0, L$	457.96	89.90
C	Min	$d7, L$	$d7, L$	$d7, L$	$d7, L$	128.85	72.80
	80%	$d6, L$	$d6, L$	$d7, L$	$d6, L$	158.18	80.60
	85%	$d6, L$	$d5, L$	$d5, L$	$d6, L$	197.06	85.10
	89%	$d4, L$	$d4, L$	$d4, L$	$d9, C$	339.16	89.15
	Max	$d0, C$	$d0, E$	$d0, L$	$d0, L$	480.70	89.90
D	Min	$d3, L$	$d7, L$	$d7, L$	$d7, L$	156.60	73.15
	80%	$d6, L$	$d6, L$	$d6, L$	$d7, L$	181.39	80.60
	85%	$d6, L$	$d5, L$	$d6, L$	$d5, L$	219.86	85.10
	89%	$d4, L$	$d4, L$	$d0, C$	$d4, L$	361.94	89.15
	Max	$d0, L$	$d0, C$	$d0, E$	$d0, L$	501.07	89.90

an average response time of 410.35ms, by: setting the models to $d0$ on device $S1$-$S4$ and offloading choices to L (local device), E (Edge device), and C (Cloud device). The agent can improve the response time by sacrificing the accuracy within a predetermined tolerable level. For example, by lowering the accuracy threshold to 85%, the average response time can be reduced by 62%.

3.6.2 Results

In this next section, we present the computational efficiency of the HDHL approach for RL-based inference orchestration and compare it to the state-of-the-art DQN algorithm as the comparative baseline [60]. Both Figure 4 and Table 5 shows the computational efficiency

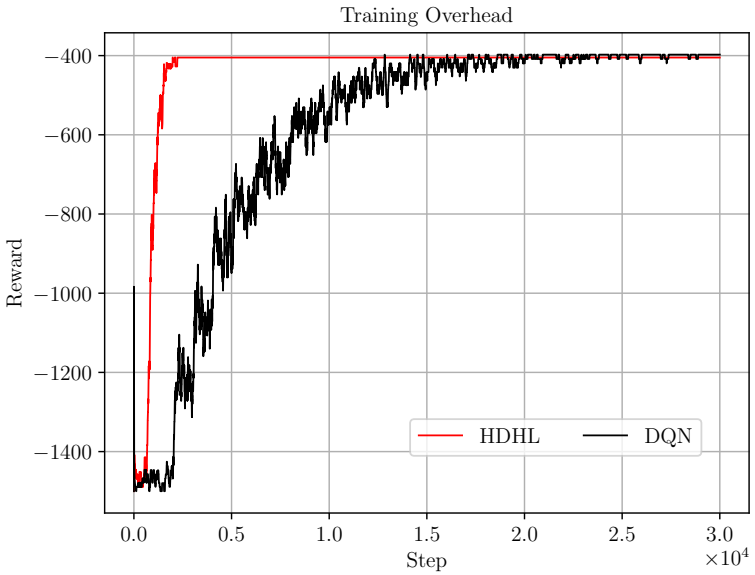


Figure 3.4: Learning Overhead for DQN and HDHL

between the baseline DQN and the two HDC-based RL models. Compared to DQN, the Hyperdimensional Q-Learning (QHD) algorithm realizes an overall 1.3× speed up to the total time, which includes the computational time to train the model and the environment interaction time, with the computational time accounting for the greatest reduction by 3.1×. This result is enhanced when deployed to a system with four end-devices with the total time seeing a speed up of 1.5×. Again, the greatest efficiency is due to the computation time being reduced by 4.9×. Meanwhile, the HDHL approach realizes an acceleration to the total time by 21.0× for 3 devices and by 9.5× for four devices. In addition, we compare the learning quality between the DQN and the HDHL models and demonstrate the latter's surpassing performance. These results are presented in Figure 5 where the reward i.e., the latency, is plotted against the number of direct interactions with the environment. It can be observed that HDHL converges at a significantly faster rate than the baseline DQN algorithm in 4.8× fewer interactions with the environment, attaining a near optimal reward, which is 4.6×10^3 milliseconds slower than the optimal configuration.

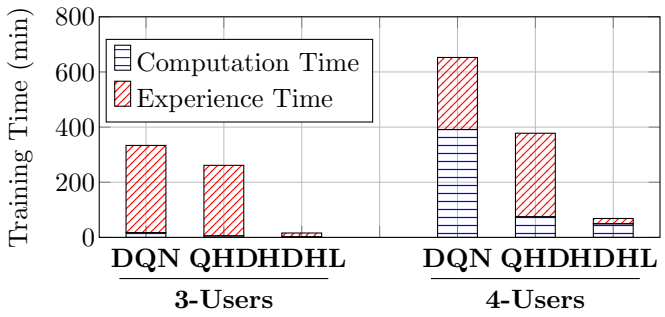


Figure 3.5: Comparison of training times between DQN [60], QHD, and HDHL.

3.7 Conclusion

In this paper we present HDHL, Hyperdimensional Hybrid Learning, the first HDC based hybrid learning algorithm, for orchestrating deep learning tasks in an end-edge-cloud architecture. We demonstrate its improved computational efficiency and learning quality compared to the neural network based Q-Learning algorithm. We show that HDHL can achieve as much as a 21.0× speedup in the total training time, as well as requiring 4.8× fewer direct interactions with the system environment to learn a near optimal configuration.

Chapter 4

CyberRL

4.1 Introduction

As we have experienced exponential growth in the technology industry, the prevalence of malicious network-targeted attacks have developed in parallel, making cybersecurity an increasingly critical challenge to address. Prior to the artificial intelligence resurgence, cybersecurity challenges were primarily delegated to domain experts, who were responsible for implementing defense protocols and directly responding to attacks. However, this system quickly grew impractical due to the increased software and infrastructure update frequencies heightening the likelihood of introducing new vulnerabilities.

A second driving force attributing to this transition is the acceleration in which cyberattacks are evolving. The advancement in their complexity is why intelligent and automated Intrusion Detection Systems (IDS) and other security defenses have been widely adopted. However, the industry standard is further progressing and moving away from human-in-the-loop to autonomous human-on-the-loop protocols. These solutions are highly sought out due to their ability to adapt to a consistently evolving environment, thus, enabling robust

solutions for handling future unforeseen threats [2, 80].

There have been various intelligent solutions for security tasks ranging from Naive Bayes [4] to neural networks [11], and SVMs [55]; however, these past approaches have required large amounts of high-quality data and careful fine-tuning to achieve adequate detection performance [23]. Furthermore, due to frequent infrastructure changes and the evolution of cyber-attacks, these supervised machine learning based IDS require maintenance for updating the training datasets to handle new attacks.

Differentiating itself from other machine learning methods, Reinforcement Learning (RL) does not require large, curated datasets at initialization. Instead, it learns from interacting with a simulated environment, which is comprised of a set of states, actions, and their respective rewards defined by environment feedback [6, 2]. One popular RL algorithm is Q-Learning, a value-based approach that learns a Q-function for approximating the Q-value (e.g., the expected reward of taking an action at a given state). The original Q-Learning algorithm is Tabular Q-Learning [89], which could only handle simple environments due to the *state explosion problem*[2]. As a result, the neural network based Q-Learning, Deep Q-Network (DQN) [62], was introduced to handle this scalability issue [27].

By implementing the Q-Learning algorithm using a deep neural network, it is able to handle applications with complex learning tasks. However, deep learning algorithms have the following computational challenges: demand unreasonable resources to train since neural networks rely on costly back-propagation and gradient-based methods, are extremely sensitive to noise in data, network, or underlying hardware, and lack human-like cognitive support for long-term memorization and transparency.

To address the aforementioned challenges, recent learning algorithms aim to model the human brain more closely, including HyperDimensional Computing (HDC). Brain-inspired HDC is gaining traction for its impressive learning ability, lightweight hardware implemen-

tation, and computational efficiency [28, 76, 96, 21, 57, 20]. The origin of this field stems from neuroscience research on how the human brain processes information in high dimension due to the brain's extensive circuitry. HDC abstractly extends this concept as a learning paradigm by modeling and operating on high-dimensional data representations [52].

HDC achieves this by encoding input data into high-dimensional space, outputting what is called *hypervectors*, and using well-defined HDC operations to train and execute inference tasks [28]. Each dimension of the hyperspace abstractly models a neuron's functionality in processing external stimuli [52]. HDC operations are simple arithmetic over these dimensions that can be supported on devices with limited resources and computational power [45]. In addition, this class of algorithms is equipped to accomplish both classification and regression machine learning tasks [28, 69, 66, 70], and has demonstrated comparable results to neural networks while offering higher learning quality (e.g., faster convergence, learning speed, and computational efficiency) [28, 95].

Given these distinctive advantages, HDC suggests itself as a promising potential solution in applications concerning low-powered Internet of Things (IoT) devices [16, 18, 33]. Most state-of-the-art machine learning approaches involve deep learning, which requires costly computational power to train, thus making them impractical for edge learning [61]. In contrast, HDC-based algorithms are an ideal candidate for their lightweight algorithmic design and energy efficiency [97].

We propose CyberRL, an HDC RL algorithm for developing cybersecurity attack and defense strategies in an abstract intrusion detection Markov game. Our contributions are the following:

- To the best of our knowledge, this paper presents the first HDC based RL algorithm to effectively learn cybersecurity strategies for network intrusion detection.
- We demonstrate CyberRL's ability to learn more effective cybersecurity defense and

attack strategies relative to previous Deep Q-Learning algorithms.

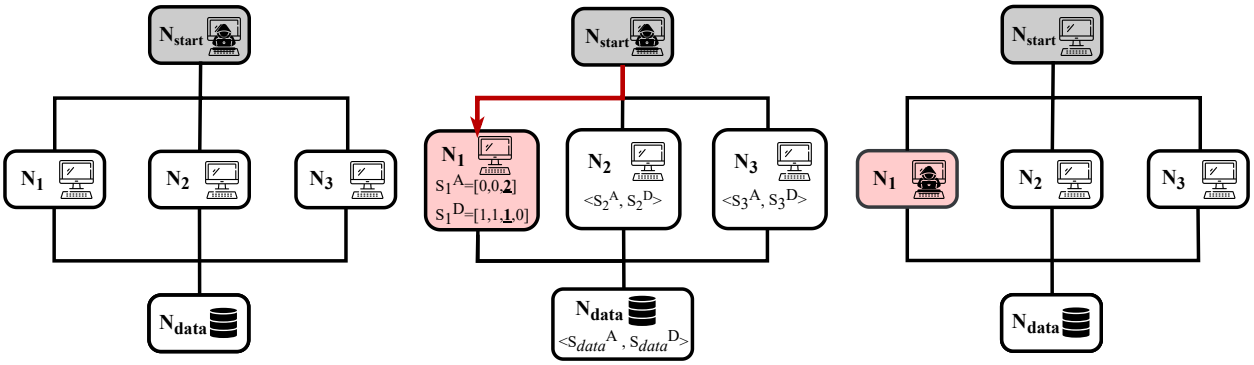
- CyberRL is computationally more efficient on multiple computing platforms, including on low-powered edge devices.
- Compared to traditional neural network based RL, CyberRL shows higher hardware efficiency. To further improve CyberRL's performance, we design an FPGA-based domain specific accelerator.

We evaluate the efficacy of our approach on multiple embedded platforms including the NVIDIA Jetson Orin I and the Xilinx Alveo U50 FPGA. Our evaluation shows that CyberRL is computationally more efficient than DQN with a speedup of 1.9× on CPU and 700× on FPGA. In addition, CyberRL is able to achieve 12.8× higher rewards, demonstrating an overall improved learning quality.

4.2 Related Works

The work from [31] introduces the developed Markov cybersecurity game environment, which demonstrates the efficacy of Deep Reinforcement Learning (DRL) algorithms for solving intrusion detection problems. Another paper from this group, which builds upon their previous work, includes [32], where they formulate the intrusion detection task into an optimal stopping problem. This subsequent work continues to utilize reinforcement learning to estimate optimal defense policies; however, the main contribution is primarily in reframing the cybersecurity environment to solve for optimal stopping.

Other recent work in the area include [26], which introduces a DRL framework for learning defense countermeasures to dynamically evolving environments. They tested various DRL algorithms, including Deep Q-Network (DQN), Advantage Actor Critic (A2C), Asynchronous



(a) Cybersecurity Environment (b) Attacking agent targeting N_1 (c) Attacking agent compromising N_1

Figure 4.1: (a) Shows an overview of the topology of a simple computer network to simulate the intrusion detection environment. The attacking agent initially resides at $\mathcal{N}_{start}$ at the beginning of each episode. $\mathcal{N}_{data}$ represents the database containing the network's critical data, which the attacking agent aims to compromise and the defending agent, the Intrusion Detection System (IDS), needs to secure. In this simulation, the network contains a single intermediate layer comprising of three computing devices: $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$ with the connectivity between devices represented by edges. (b) The partially observable state spaces for the agents are shown within each node in the network: $<S_i^A, S_i^D>$, where S_i^A is the attacking agent's cyber-attack repertoire and S_i^D is the IDS's defense to each attack with i indicating the network node. Note that the number of cyber attacks in this elementary simulation is limited to $m = 3$; thus, $|S^D| = 4$ since the defending agent has an additional attribute for the breach detection capacity of the respective node. In (c), the agent is able to successfully compromise $\mathcal{N}_1$ since $S_{1,2}^A = 2 > S_{1,2}^D = 1$. Upon compromising $\mathcal{N}_1$, the attacking agent is now able to target nodes adjacent to $\mathcal{N}_1$ (i.e., $\mathcal{N}_{data}$).

Actor-Critic, and Proximal Policy Optimization (PPO), to demonstrate the efficacy of DRL to proactively detect various cybersecurity threats. This work introduces a new reinforcement learning environment, which extends the OpenAI Gym library; however, their code has yet to be open-sourced.

In the space of HDC reinforcement learning algorithms, CyberRL is an optimized version of the QHD algorithm introduced in [66]. CyberRL is developed from the Double Q-Learning algorithm [34] and similar to [66], CyberRL implements the Q-function approximation with the HDC framework in lieu of a deep neural network. An issue limiting [66] is its inability to scale to solve RL tasks with large state and action spaces. In this prior work, its primary results utilize the OpenAI Gym game environments (e.g., Cartpole, Acrobot, and LunarLanding) [12] where the state and action spaces for each task is relatively small. The optimization of the former algorithm enables CyberRL to scale to learn more complex RL tasks, such as the one provided here [31].

4.3 Modeling Intrusion Prevention as a Game

This next section details Hammar and Stadler's abstract simulation of an intrusion detection cybersecurity environment [31]. The task is modeled as a multi-agent zero-sum Markov game, indicating that the positive reward of one agent is strictly at the detriment of the other(s). This particular game environment constitutes two agents: a cyber attacker and an Intrusion Detection System (IDS), the defending agent. The former's objective is to compromise a computer network, while the latter's adversarial task is to secure it; in addition to successfully detecting any attempted security breaches by the attacking agent. The game terminates when either agent completes their defined task.

The environment is implemented to simulate a computer network of interconnected comput-

ing devices. The node in the first layer, denoted $\mathcal{N}_{start}$, is where the attacking agent resides when the game is initialized. $\mathcal{N}_{data}$, which comprises the last layer, contains the network's sensitive data, which is the node that the defending agent aims to protect from breaches by the attacking agent. The intermediate layers can include any number of nodes and any defined depth. An illustration of such a structure is presented in Figure 4.1, where a simplified network is modeled using a single intermediate layer containing three nodes.

The attacker's objective is to reach $\mathcal{N}_{data}$ and successfully compromise it. In order to accomplish this, the attacker must explore the infrastructure through reconnaissance and compromise intermediate nodes until it reaches $\mathcal{N}_{data}$. However, the topology of the infrastructure remains unknown to the attacker with the only accessible information being the successfully compromised nodes and their adjacent neighboring devices. Conversely, the defending agent is aware of the entire network infrastructure, but lacks information pertaining to the status of the attacking agent. Since neither agent can observe the full state space, it formulates a Partially Observable Markov Decision Process (POMDP) [82].

The game is simulated on a round-by-round basis with the defender and attacker alternatively selecting actions to accomplish their respective adversarial objectives. The infrastructure of the game is represented as a graph $\mathcal{G} = <\mathcal{N}, \mathcal{E}>$, where $\mathcal{N}$ denotes the nodes in the network and $\mathcal{E}$ denotes the edges in the graph that represent the devices that are connected to one another. Each node in the network $\mathcal{N}_i \in \mathcal{N}$ has a node state $S_i = <S_i^A, S_i^D>$ to encompass each agent's observable subset of the state space.

Given a node $\mathcal{N}_k$, the attacking agent has a repertoire of cyber-attacks that it can leverage on the targeted node $S_k^A = [S_{k,0}^A, S_{k,1}^A, ..., S_{k,m-1}^A]$, where each element in the set abstractly represents the type of cyber-attack (e.g., Denial-of-Service (DOS) attack, cross-site scripting attack, etc.). Each attribute takes on a numerical value, which represents the strength of the attack and remains unknown to the defending agent.

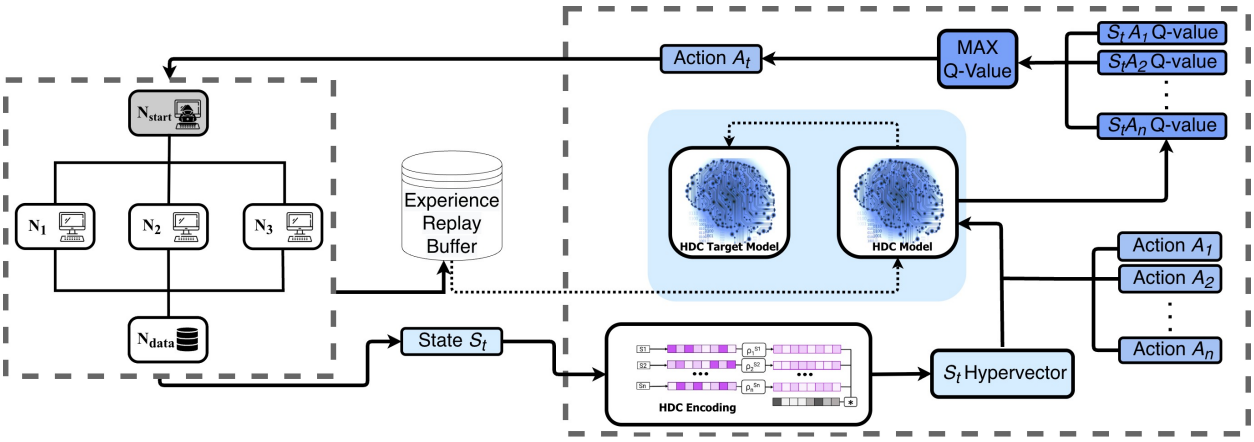


Figure 4.2: The above figure outlines the steps of the CyberRL algorithm in the given intrusion detection environment. (A) Given a state s_t, the state is then (B) encoded into a hypervector and (C) multiplied with the class hypervectors, which correspond to each action in the action space (e.g., $a_1, a_2, ..., a_n$). The action corresponding to the maximum Q-value is selected (D) as the next move in the environment. At the conclusion of each step, the given state s_t, the selected action a_t, and the resulting reward r_t, and next state s_{t+1} (E) are stored as a tuple (s_t, a_t, r_t, s_{t+1}) in the experience replay buffer to later be used for (G) updating the CyberRL model. (F) indicates the interaction with the environment, while (H) shows the periodic target model updates.

Similarly, the defending agent has a corresponding set of attributes for node $\mathcal{N}_k$ that map to the defense strength against each cyber-attack: $S_k^D = [S_{k,0}^D, S_{k,1}^D, ..., S_{k,m}^D]$. These represent cybersecurity defenses, such as firewalls or encryption functions. For instance, if attribute j represents a DOS attack, $S_{k,j}^A$ indicates the strength the attacking agent has in leveraging a DOS attack on node $\mathcal{N}_k$. Meanwhile, $S_{k,j}^D$ represents $\mathcal{N}_k$'s resilience from being corrupted by a DOS attack. The final defense attribute in the set for each node $S_{k,m}^D$ is the defending agent's strength in detecting the attacking agent at $\mathcal{N}_k$. The attributes for both the attacking and defending agents are limited to a finite range, specifically $S_{i,j}^A, S_{i,j}^D \in [0, w]$, where w is a natural number.

In terms of the action space, the attacking agent has two courses of action: the first would be to increase the strength of one of its cyber-attacks on any one of the accessible nodes (i.e., increment the value of $S_{k,j}^A$ for node $\mathcal{N}_k$ and attack j) or to attempt to compromise a visible node with an attack of its selection from $0...m-1$. Meanwhile, the defending agent may

either choose to strengthen its defense from a particular attack (i.e., increment the value of a defense attribute $S^D_{k,j}$ for node $\mathcal{N}_k$ and counter-attack j) or choose to strengthen the detection ability for any node in the infrastructure (i.e., increment $S^D_{k,m}$ attribute).

If the attacking agent chooses to wage a cyber-attack on node $\mathcal{N}_k$, an attack is simulated by exposing the defending agent's corresponding attribute S^D_k to the attacker. Given that the attacker chose to attack with the attribute j, the attacker would successfully compromise the node if $S^A_{k,j} > S^D_{k,j}$. If the attack is unsuccessful then the defender has an opportunity to identify the attack with a probability of $p = \frac{S^D_{k,m}}{w+1}$, where $S^D_{k,m}$ is the defending agent's detection capability.

For example, in Figure 4.1 (b-c) the attacking agent is able to successfully compromise $\mathcal{N}_1$ since it leveraged cyber attack $S^A_{1,2}$, which is stronger than $S^D_{1,2}$, the IDS security capability in defending from the respective cyber-attack. If in the scenario that $S^A_{1,2} <= S^D_{1,2}$, then with probability $\frac{S^D_{1,3}}{w+1}$, the IDS would detect the cyber attacker; hence, winning the episode.

The game progresses as each agent takes an action of its choice altering the state environment in each round of the game. The episode terminates when either the attacking agent successfully compromises $\mathcal{N}_{data}$ or when the IDS detects the attacker. The winning agent is awarded +1 utility, while the losing agent receives -1 utility in rewards.

4.4 CyberRL as an Intrusion Detection System

In this section, we detail the CyberRL algorithm employed to learn both attack and defense strategies.

As is known with learning new environments in RL tasks, the agent needs to learn undiscovered areas of the state space while also utilizing past experience to learn the best action

to take at any given state. This is known as the exploration vs. exploitation trade-off. Exclusively using one of these approaches results in one of the two extremes: either the model never learns the Q-function or the model converges to a local maxima, resulting in a sub-optimal configuration.

To mitigate this, the epsilon-greedy method is employed, which initializes the RL agent to prioritize exploration at the early stages of training. However, as more experience is accumulated, the agent increasingly relies on the learned Q-function until ϵ completely decays, enabling pure exploitation of its past experience. This is done by initializing $\epsilon = 1$ and applying a decay rate for epsilon to converge to zero. At each time step in training, if the generated number is less than ϵ, then a random action is selected. If not, the model selects the action with the highest Q-value, as shown below:

$$a_t = \begin{cases} \text{random action } a \in \mathcal{A}, & \text{with probability } \epsilon \\ argmax_{a \in \mathcal{A}} Q(s_t, a), & \text{with probability } 1 - \epsilon \end{cases}$$

given an action space $\mathcal{A}$ at time step t.

Similar to Deep Q-Learning, CyberRL also delegates the Q-function approximation to a machine learning model rather than recordkeeping a Q-table; however, instead of a neural network approximating these values, the agent utilizes an HDC model. This is achieved by mapping the states and actions into hyperdimensional space using the exponential kernel as the encoding function. The HDC model contains n model hypervectors $\{\vec{\mathcal{M}}_{a_1}, \vec{\mathcal{M}}_{a_2}, ...\vec{\mathcal{M}}_{a_n}\}$, where n is the size of the action space (i.e., each of these hypervectors corresponds to an encoded action).

With Deep Q-Learning, an action is selected by choosing the argmax of the highest Q-value computed by the trained neural network. In the HDC version, we do this by multiplying the encoded state hypervector $\vec{\mathcal{S}}_t$ with each model hypervector $\vec{\mathcal{M}}_{a_i}$ to output the corresponding

Q-value for each state-action pair. The agent selects the action with the highest Q-value and interacts with the environment, collecting feedback r_t, and observing the next state in this transition s_{t+1}. The algorithm advances to the next time step $t+1$, and iterates until the episode terminates. An overview of this process is detailed in Figure 4.2.

The class hypervectors of the HDC model are updated over episodes as more experience is accumulated. After each time step t, the state, the selected action, the reward, and the next state are stored in an experience replay buffer as a tuple (s_t, a_t, r_t, s_{t+1}). This experience replay strategy is used to collect additional data for training the agent in an online approach. After sufficient data samples are collected, a batch of these tuples are sampled from the experience replay buffer to update the HDC model $\mathcal{Q}$. Unlike supervised machine learning, which has labeled data at its disposal, the Q-Learning algorithm relies on the Bellman Equation or Dynamic Programming Equation [9], which is a recursive expression for the Q-value at time step t, to estimate the true Q-values. We use this to define the optimal Q-function

$$q_{t\_true} = r_t + \gamma max_a \mathcal{Q}'(s_{t+1}, a) \tag{4.1}$$

and to maximize the accumulated rewards in an episode.

In order to achieve this, the Bellman Equation states that each sub-task must be optimized to realize the highest rewards for the entire task, which means q_{true} is the sum of r_t and the maximum Q-value for the next time step. As noted in the equation above, we use $\mathcal{Q}'$, the target model, since our approach uses the Double Q-Learning method in updating the model [34]. Compared to $\mathcal{Q}$, which is updated at every time step, $\mathcal{Q}'$ is updated periodically and stabilizes the learning process to avoid overestimating the Q-value, a common consequence of maximizing the Bellman Equation. Additionally, the reward decay γ is included to adjust the weight on future or near-sighted rewards: a value of 1 puts a higher weight for the

long-term rewards and a value close to 0 prioritizes the more immediate rewards.

Returning to the update step: for a given tuple (s_t, a_t, r_t, s_{t+1}), we encode the state s_t into hyperdimensional space to give its corresponding hypervector $\vec{\mathcal{S}}_t$ and multiply it with action a_t's class hypervector, $\vec{\mathcal{M}}_{a_t}$, from the HDC model

$$q_{t\_pred} = \mathcal{Q}(s_t, a_t) = \vec{\mathcal{S}}_t \times \vec{\mathcal{M}}_{a_t} \tag{4.2}$$

We take the difference between q_{pred} and q_{true} and use it to update the HDC model

$$\vec{\mathcal{M}}_{a_t} = \vec{\mathcal{M}}_{a_t} + \alpha(q_{t\_true} - q_{t\_pred}) \times \vec{\mathcal{S}}_t \tag{4.3}$$

where α is the learning rate. Over the duration of learning, the Q-function gradually becomes more accurate in estimating the best actions across the state space.

Furthermore, this development of the HDC Double Q-Learning model includes optimizations in the action selection and model update steps using batch training. This allows for the HDC model to be vectorized in both the Q-value calculations and target model updates. As a consequence, the model experiences significant computational efficiency to enable it to scale to RL tasks with larger state and action spaces.

4.5 FPGA Architecture Design

Compared to CPU and GPU, HDC models generally have a higher execution performance and energy-efficiency on the FPGA platform [46, 67]. On one hand, FPGA accelerators achieve a balance between computation parallelism and energy efficiency through customized

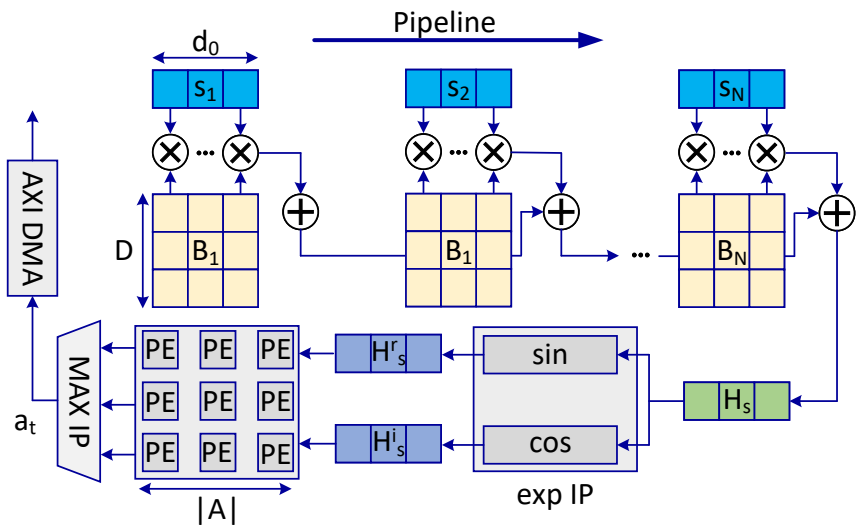


Figure 4.3: CyberRL model acceleration on the FPGA. Here s_i represents agent i's original state vector. $\vec{B}$ is the base hypervector matrix. H_s is the encoded state hypervector.

hardware design. On the other, HDC model computation does not require high-precision data computation due to the holographic nature of the HDC model. Essentially, this implies that we can achieve parallel hypervector computation without relying heavily on digital signal processing IP (DSP) and by utilizing lookup tables (LUT) instead.

In Figure 4.3, we present the FPGA acceleration of CyberRL. Here we assume our model simultaneously controls N agents with each agent's state vector dimensions being $1 \times d_0$. For each agent i, a corresponding base hypervector matrix $\mathbf{B_i}$ with dimensions $d \times D$ is stored in the on-chip storage (such as BRAM). During the state encoding process, we pipeline each agent's state encoding computation. The mathematical representation of this pipeline is:

$$\vec{H_s} = \sum_{i=1}^{N} \vec{d_i} \mathbf{B_i} \tag{4.4}$$

Here the dimension of the encoded state hypervector ($\vec{H_s}$) is $1 \times D$. Before conducting regression operations to select the optimal action, we also pass the encoded state hypervector through a kernel function, such as the exponential function used in CyberRL. To implement the kernel function on-chip, we divide the kernel function IP into two parts, the sine and

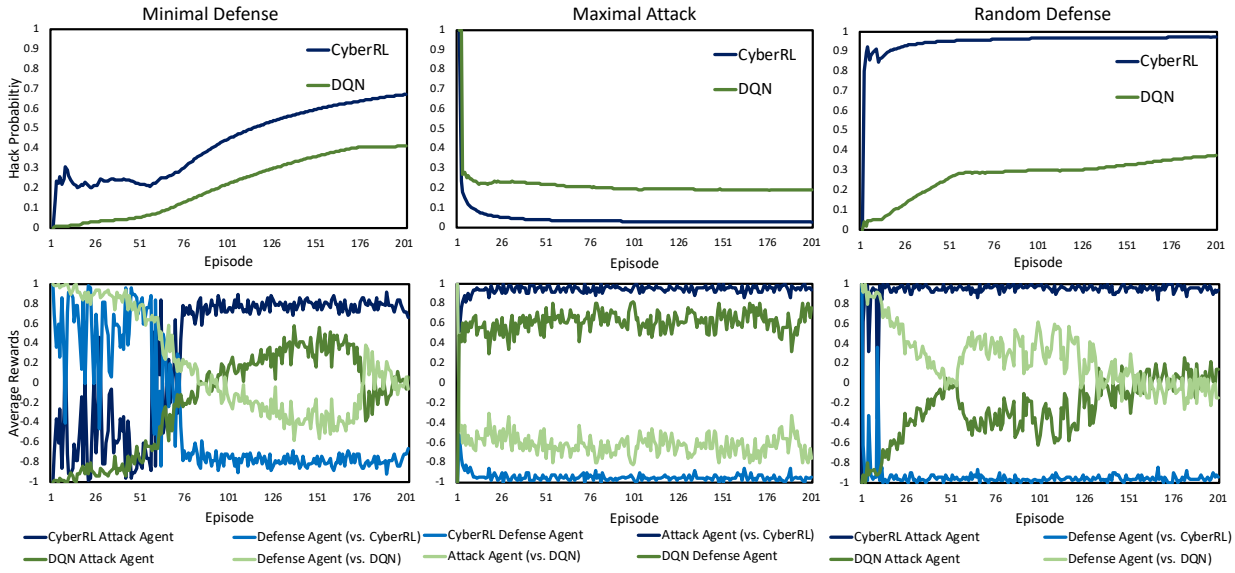
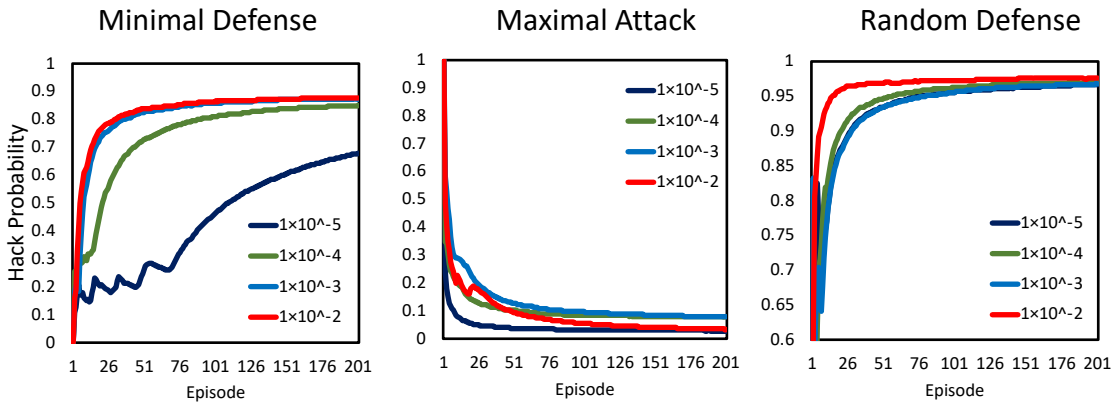


Figure 4.4: Comparison of our CyberRL to DQN [62] in the three different intrusion detection scenarios. The top row shows the Hack Probability of the network being compromised, while the bottom row shows the Average Rewards, computed from the last 100 episodes of self-play.

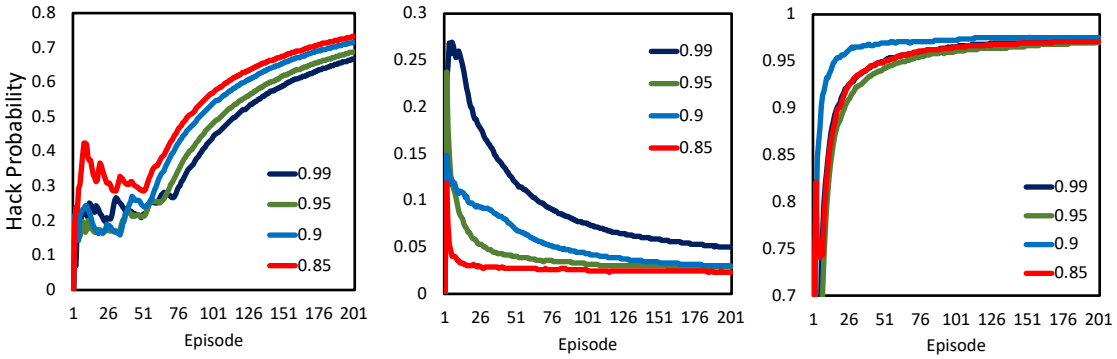
cosine function components since we have:

$$e^{jx} = cos(x) + jsinx(x) \tag{4.5}$$

To save on resources, we choose to pre-store sine and cosine values inside the on-chip storage (such as on-chip BRAM) instead of using Xilinx's existing hardware IP, such as CORDIC IP. For the regression part, we accelerate the encoded state and action hypervectors' matrix multiplication using a systolic array [19]. In the last stage, a max IP is used to choose the highest action index. This action index will also be passed back to host CPU via AXI DMA IP.



(a) Convergence of different values for α with $\varepsilon, \gamma, Q' update\ frequency$ set to their respective optimal values.



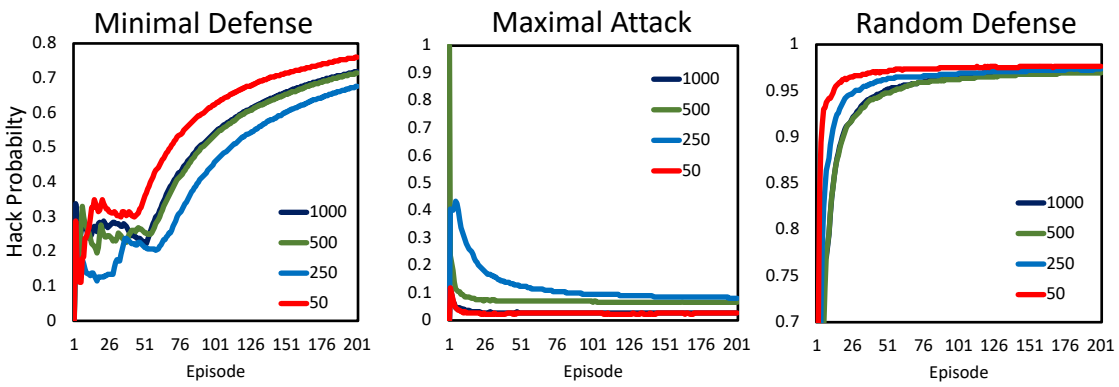
(b) Convergence of different values for ε with $\alpha, \gamma, Q' update\ frequency$ set to their respective optimal values.

Figure 4.5: Experiments for fine-tuning the values (a) α and (b) ϵ across the three cybersecurity scenarios.

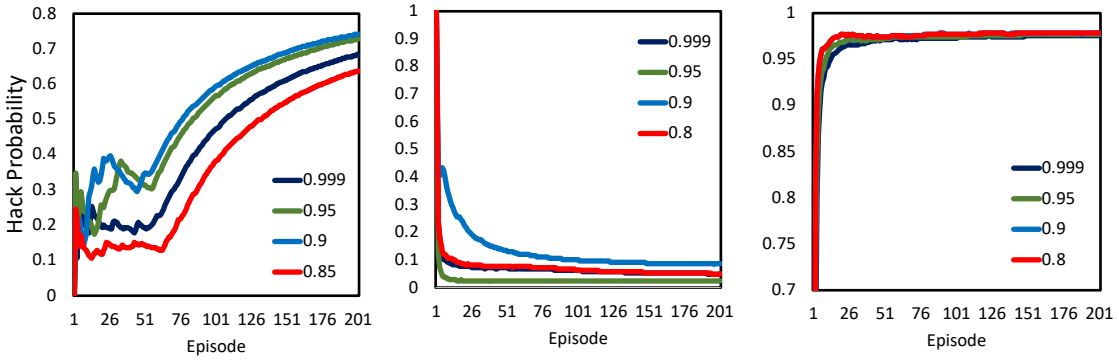
4.6 Experiments

4.6.1 Attack and Defense Scenarios

To measure the effectiveness of HDC in learning cybersecurity strategies, we applied our CyberRL algorithm to various intrusion detection scenarios given the simulated environment detailed in Section 4.3, which serves as the dataset [31]. This is a multi-agent RL environment with one agent attempting to compromise the given computer network, while the other acts as an Intrusion Detection System (IDS) (e.g., the attack and defense agents respectively). The given environment includes multiple scenarios to allow for training both of these agents. They are the following:



(a) Convergence of different values for $Q'update\ frequency$ with $\alpha, \varepsilon, \gamma$ set to their respective optimal values.



(b) Convergence of different values for γ with $\alpha, \varepsilon, Q'update\ frequency$ set to their respective optimal values.

Figure 4.6: Experiments for fine-tuning the values (a) $\mathcal{Q}'$ and (b) γ across the three security scenarios.

- *Minimal Defense Scenario*: where we train an attacking agent against the environment's IDS, which follows a policy that will always defend the attribute with the minimal value of its neighbors
- *Random Defense Scenario*: where we train an attacking agent against an IDS that uses a random baseline defense policy
- *Maximal Attack Scenario*: where we train the IDS to defend against an attacking agent, who will target the node which has the highest attack attribute

It's important to note that the adversarial agent in each of these scenarios will be simulated by the environment. For instance, in the *Maximal Attack* scenario, the adversarial agent

is the attacker, which is integrated into the environment, while we train the implemented CyberRL defense agent.

We run our CyberRL algorithm in each of these scenarios and compare it to the Deep Q-Network (DQN) from [62] in terms of computational efficiency and quality of the learned security strategies. The latter is measured by the trained agent's realized Average Rewards during training and Hack Probability.

The Hack Probability metric is the likelihood measure of the computer network environment being hacked. When training a defense agent, as is the case in the *Maximal Attack* scenario, a lower hack probability is desired since it is aligned with the trained model's objective; conversely, for the *Random* and *Minimal Defense* scenarios, a higher hack probability measures a better trained attack agent. The second metric, the Average Rewards, is calculated from the previous 100 game simulations or episodes, where rewards are the measured utility (e.g., -1, +1) of winning the simulated game.

The topology of the network consists of two intermediate layers, each with three nodes, for a total of 8 nodes. Additionally, the attack agent has ten different attack attributes that it can leverage against a node in the network. Hence, the state space of this environment consists of 168 features and the action spaces for the agents are 80 and 88 for the attacking and defending agents respectively. We use the implementation of the DQN from the original project [31] and compare it to our CyberRL model, which has been fine-tuned. Table 4.1 shows the values for the set hyperaparameters for the final model used to compute the results reported in Figure 4.4.

Table 4.1: CyberRL model hyperparameter configurations for different cybersecurity strategies.

Hyperparameter	**Minimal Defense**	**Maximal Attack**	**Random Defense**
HDC dimension	1000	1000	1000
Learning rate (α)	0.01	0.00001	0.01
Epsilon decay (ϵ)	0.85	0.85	0.90
Target model update	50	50	50
Discount factor (γ)	0.90	0.95	0.80
Batch size	32	32	32

Note: HDC = Hyperdimensional Computing. Values represent optimal configurations for each defense strategy.

Table 4.2: NVIDIA Jetson Orin I Platform. Here EMC means external memory controller. The value of EMC represents the external memory usage.

Power	**15W**	**30W**	**50W**	**Max**
Frequency	1.1GHz	1.7GHz	1.5GHz	2.8GHz
EMC	5%	6%	6%	2%
Memory	2.2GB	2.4GB	3GB	6.4GB
CPU	4 cores	8 cores	12 cores	12 cores

4.6.2 Experimental Setup

We run our algorithms on a 2.4GHz 8-core Intel Core i9 to report our results. In addition, we deploy our model on the NVIDIA Jetson Orin I with varying power settings. Table 4.2 reports the resource utilization of the NVIDIA Jetson Orin I on each of these power settings. As one can observe from Table 4.2, when the power consumption increases, the corresponding frequency, main memory, external memory controller (EMC), and computing cores consequently increase as well. Furthermore, for the FPGA acceleration, we implement the algorithm using C++ HLS and we select Xilinx Alveo U50 as the kernel FPGA board. The communication between host CPU and kernel FPGA is based on the Xilinx Vitis platform [53]. We also use Xilinx Vivado Power Estimator (XPE) to measure the kernel FPGA on-chip power consumption [8].

4.7 Evaluation

In this section, we demonstrate the efficacy of applying our CyberRL algorithm in finding security strategies in an intrusion detection RL environment. We compare it to DQN and evaluate the performance of the agents in multiple intrusion detection scenarios. We report the computational efficiency and the overhead incurred by the agents' learning process.

4.7.1 Hyperparameter Optimization and Training

We conducted a thorough exploration of the various hyperparameters for the CyberRL model, which is implemented using a Double Q-Learning framework. The hyperparameters that were fine-tuned include the following:

- α: the learning rate for the HDC model
- ϵ: the epsilon decay for the ϵ-greedy algorithm
- $\mathcal{Q}'$: the target model update frequency for stabilizing the learning curve
- γ: the discount factor for calculating the Temporal Difference Error for the Q-learning algorithm

Figures 4.5 and 4.6 display the full set of experiments. The columns correspond to the cybersecurity scenario, while the rows indicate the hyperparameter being optimized. The metric used to determine the optimal value is the *Hack Probability*. The first set of experi-

Table 4.4: Design Acceleration on Alveo U50. The FPGA kernel frequency is 200MHz and FPGA power consumption is 17W. Here $T_{attacker}$ and $T_{defender}$ are single time step latency.

Name	LUT	FF	DSP	BRAM	URAM
Total	398.5K	728.8K	7	713	316
Available	872K	1743K	5952	1344	640
Utilization	45.6%	41.8%	∼0%	53.1%	49.3%
$T_{attacker} = 8.94us$, $T_{defender} = 8.64us$					

ments demonstrated in Figure 4.5 include experiments for determining the values for α and ϵ in each scenario, while Figure 4.6 includes the experiments for $\mathcal{Q}'$ and γ.

4.7.2 Performance

We evaluate the performance between CyberRL and DQN across the three cybersecurity scenarios. Our primary results are presented in Figure 4.4, which compares the two models over the duration of training. The Hack Probability metrics are presented in the top row of Figure 4.4 and the Average Rewards for each agent and their adversarial agent in the lower row.

Across all three scenarios, the CyberRL model is able to learn a more effective strategy, as shown by achieving a higher Hack Probability when training an attack agent (the left and right columns of Figure 4.4), and a lower Hack Probability when training a defense agent in *Maximal Attack*. Furthermore, the effectiveness of the CyberRL implementation of the defense and attack agents are reflected in the Average Reward Plots.

The intuition for why CyberRL outperforms the deep learning equivalent stems from HDC's intrinsic memory-like capabilities [74] and pattern recognition from its training framework. By encoding the state and action spaces in hyperdimensional space, CyberRL is advantageous in learning the critical features of the task environment significantly faster than

the neural network architecture. These results are not anomalous since HDC-based reinforcement learning algorithms have shown to outperform their deep learning counterparts in various tasks [47, 66, 68, 65, 5].

Table 4.5: Comparison of computational efficiency between DQN [62] and CyberRL algorithms on NVIDIA Jetson Orin at various power settings. Measurements are for the Minimal Defense scenario (in seconds).

Algorithm	**15W**	**30W**	**50W**	**MAX**
DQN	1.9×10^3	1.5×10^3	1.6×10^3	1.0×10^3
CyberRL	1.4×10^3	1.0×10^3	1.3×10^3	0.8×10^3

Note: Lower values indicate better computational efficiency. All measurements in seconds.

4.7.3 Efficiency

In Table 4.4, we present the resource utilization of our algorithm's implementation on Xilinx Alveo U50 FPGA. For each time step, the attacker agent's execution latency is $8.94\mu s$ and the defender agent's latency is $8.64\mu s$. In addition, Figure 4.7 plots the CyberRL's improvements in latency and energy efficiency in each of the three scenarios on the CPU and FPGA platforms. The first plot of Figure 4.7 displays the latency improvements compared to DQN calculated from the training time, which is the duration of time for the model to converge. As anticipated, the CyberRL algorithm is significantly faster in all three scenarios: for the *Maximal Attack Scenario*, the CyberRL is 1.2× more efficient; 1.9× faster in the *Random Defense Scenario*; and 1.6× in the *Minimal Defense Scenario*. When ran on the Xilinx Alveo U50 FPGA, CyberRL achieves around 700× speedup and energy efficiency improvements compared to the CPU execution.

Furthermore, Table 4.5 compares CyberRL and DQN when executed on the NVIDIA Jetson Orin I platform on various low-powered settings (e.g., 15W, 30W, 50W, Maximum Power). We demonstrate CyberRL's efficiency advantage over DQN across all power settings, which

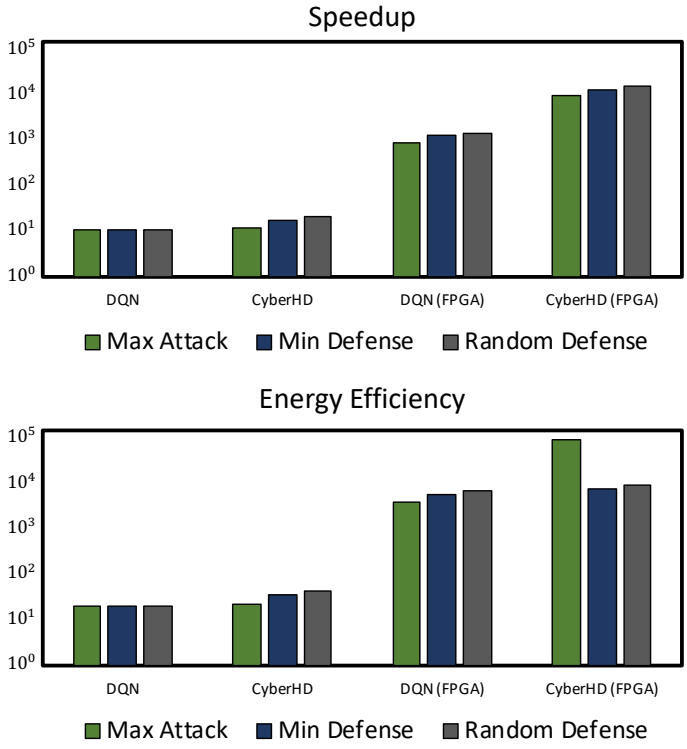


Figure 4.7: The above figure displays the time and energy efficiency improvements relative to DQN [62]. (a) shows the CyberRL speedup on both the CPU and FPGA platforms, while (b) shows the improved energy efficiency.

also realizes a reduced training time of up to 30%.

4.8 Conclusion

In this paper we present CyberRL, an efficient, brain-inspired algorithm for learning cybersecurity strategies in an abstract Markov game environment for solving intrusion detection type security problems. We demonstrate that CyberRL is advantageous in both computational efficiency and in producing stronger defense and attack strategies.

Chapter 5

HDCyberSAC: HDC Soft Actor Critic

5.1 Introduction

Modern cybersecurity faces unprecedented challenges from increasingly sophisticated threats that outpace traditional detection methods. This rapid evolution demands new approaches to threat monitoring and prevention. While machine learning (ML) has been employed to address these problems, ML itself introduces new vulnerabilities [25]. ML, and particularly deep learning, excels at learning from massive volumes of high-quality, labeled data, making it suitable for well-defined inference tasks. In cybersecurity, however, acquiring such large, curated datasets is notoriously difficult. Furthermore, the dynamic nature of cyber threats means systems must constantly grapple with novel, out-of-distribution data—a scenario where conventional ML models often struggle. These challenges significantly limit the practical application of ML in this domain.

A more promising approach to cybersecurity challenges is Reinforcement Learning (RL), a subfield of machine learning focused on sequential decision-making that exhibits stronger generalization capabilities when tackling novel inference tasks [79]. Deep Reinforcement

Learning (DRL) has proven its versatility in diverse fields such as robotics, gameplay, navigation, and industrial automation [62, 58, 84]. Its inherent ability to adapt and make decisions in complex, evolving environments holds strong potential for detecting new and unknown cyber attacks. Consequently, a significant body of work has emerged on applying DRL to cybersecurity problems, particularly in developing advanced Intrusion Detection Systems (IDS) [3, 40, 79].

However, a well-documented concern regarding deep learning is the massive computational cost these algorithms require. Specifically, they demand high compute capacity to handle backpropagation, and are sensitive to noisy data, networks, or hardware. Furthermore, deep learning lacks human-like learning qualities, including long-term memorization and transparency.

A recently pioneered technology is HyperDimensional Computing (HDC), introduced through Kanerva's cognitive model [52] to address "the binding problem" of connectionist models, such as neural networks [28]. This class of models has gained interest due to its computational efficiency, enhanced learning quality, and lightweight hardware implementation, particularly on Field Programmable Gate Arrays (FPGAs) [28, 76, 96, 21, 57]. This cognitive model is inspired by how the human brain processes sensory data in high dimensions, owing to its extensive neural circuitry. HDC abstracts this phenomenon by developing a learning paradigm that uses defined HDC operations to operate over high-dimensional vectors, called *hypervectors* [28]. These *hypervectors* are transformed data representations in hyperspace that encode the information they represent [52]. Since HDC operations are simple arithmetic operations, they are well supported on devices with limited resources and computational power [45].

Extending this alternative computing paradigm to RL is a compelling direction for addressing the computational limitations of DRL. Cybersecurity deployments often operate under resource constraints, where the high compute capacity demanded by backpropagation and

gradient-based training in DRL presents a significant bottleneck. This is especially relevant in penetration testing, where agents must be trained and evaluated across complex, large-scale network environments. HDC-based RL offers a more scalable and efficient alternative, enabling capable security agents to be developed and deployed with reduced computational overhead.

Thus, we propose HDCyberSAC, a novel HDC-based RL algorithm for solving partially observable network security problems. By extending the state-of-the-art Soft Actor-Critic (SAC) algorithm [30] to partially observable systems, HDCyberSAC is designed to develop robust security strategies in offensive penetration-testing scenarios that reflect realistic constraints, while offering meaningful improvements in latency and computational efficiency over its deep learning counterparts.

5.2 Network Attack Simulator and Emulator Environment

To develop robust security strategies, researchers employ penetration testing, which simulates cyber attacks to identify and exploit vulnerabilities in a computer network, providing analysts with critical insights for improving defenses and preventing future breaches. To facilitate this research, a number of RL environments have been developed to simulate computer networks. Within these simulations, an RL agent assumes the role of either a hacker attempting to compromise the system or an Intrusion Detection System (IDS) working to secure it. By modeling these adversarial interactions, researchers can gain a deeper understanding of where critical security vulnerabilities lie.

While numerous pentesting RL environments exist (see Section 5.6), many are limited to pure simulation, constraining their ability to model realistic network dynamics. The Net-

work Attack Simulator and Emulator (NASimEmu) environment [49] addresses this gap. Building on the Network Attack Simulator (NASim) [78]—a flexible framework for training RL agents in offensive pentesting—NASimEmu extends the environment to full emulation using industry-standard tools like Vagrant, VirtualBox, and Metasploit. This approach fosters the development of RL agents capable of generalizing across novel network configurations, thereby significantly enhancing the realism and applicability of cybersecurity strategies.

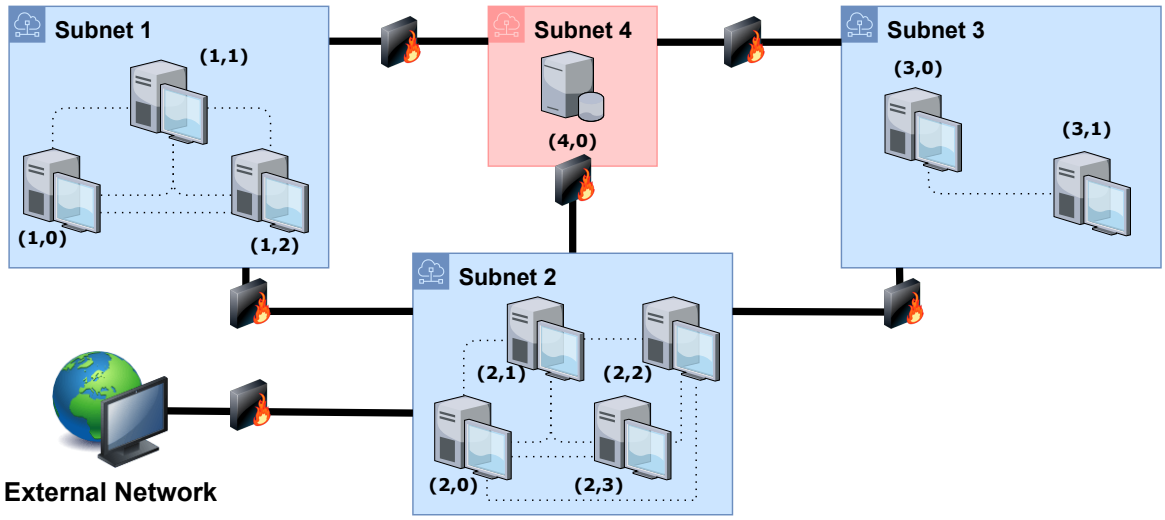


Figure 5.1: Network structure in the NASim(Emu) environment, featuring four subnets with connections between subnets controlled by firewall-settings. Device identifiers $(subnet_{ID}, machine_{ID})$ simplify the 32-bit IP addressing. The sensitive target host, (4,0), is highlighted by the color scheme.

The underlying simulation model, NASim, structures a computer network as multiple *subnets*, each containing any number of devices (Fig. 5.1). Each device is connected to all others within its subnet; however, connections between subnets are defined by a topological map and firewall settings that control inter-subnet and external communication via permissible services—in lieu of ports for simplification.

Devices within the network run either Windows or Linux operating systems and host common computer protocols (e.g., *ssh*, *ftp*, *http*, *samba*, *smtp*) and processes (e.g., *tomcat*, *daclsvc*, *schtask*). The complexity of the environment can be tuned by limiting the available OSes,

services, and processes. Given this setup, the state space size is calculated as:

$$state\_space = N_{hosts} \times 2^{(N_{os}+N_{services}+3)\cdot 3} \tag{5.1}$$

where N_{hosts} is the number of devices, N_{os} is the number of operating systems, and N_{services} is the number of services. The base of 2 represents a boolean value for the host's presence. The first 3 in the exponent enumerates the possible states of a device: *compromised*, *reachable*, and *discovered*. The second 3 defines the number of access levels on a host (e.g., *user*, *root*).

NASim provides a suite of exploits, each targeting a specific service and operating system with a defined probability of success and a cost to the agent. The agent can also use privilege escalation actions that target running processes. The agent's goal is to compromise *sensitive hosts*—devices containing critical data. To gather information, the agent can execute various scans (subnet, process, OS, service) to reveal network details.

Building upon NASim's simulated environment, NASimEmu integrates an emulator, enabling simulation-trained agents to be deployed and tested in a more realistic setting. NASimEmu also introduces several key extensions: dynamic scenario configurations with randomness, support for parallel training and testing of multiple agents, and random perturbations of node and segment IDs at each episode's initialization to prevent agent memorization. An overview of the SAC algorithm's interaction with this environment is depicted in Fig. 5.2.

5.3 HDCyberSAC Design

HDCyberSAC adapts the Soft Actor-Critic (SAC) framework by replacing its deep neural network components with HDC-based function approximators. The network state is represented as a variable-length set of N device nodes, each described by a d-dimensional binary

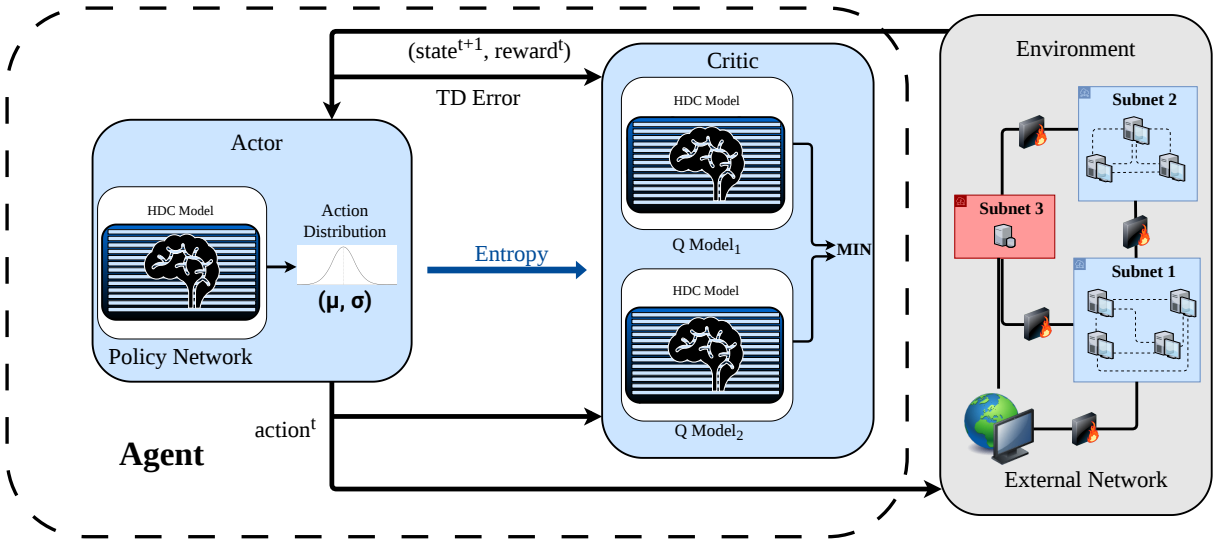


Figure 5.2: An overview of HDCyberSAC Framework, consisting of the HDC-based Actor and Critic models, interacting with the NASimEmu environment.

feature vector encoding the device's operating system, running services, and active processes. The architecture consists of three components: an HDC state encoder, an HDC actor (policy), and an HDC critic (Q-function) model, all operating over complex-valued hypervectors of dimension D.

5.3.1 HDC State Encoder

The state encoder maps each device node into a D-dimensional complex hypervector that captures both its local attributes within its respective subnet and its relational context within the broader network state.

The encoder is parameterized by two randomly sampled projection matrices: a feature matrix $\mathbf{F} \in \mathbb{R}^{2\times d\times D}$, where d is the device node dimension in the state encoding. The feature matrix $\mathbf{F}$ stores separate basis vectors for each possible binary feature value and a positional projection matrix $\mathbf{P} \in \mathbb{R}^{p\times D}$ with a random phase bias $\mathbf{b} \in \mathbb{R}^{D}$, where p is a positional encoding dimension to preserve the structure of the state space (i.e., features in computing

network). Both matrices are sampled from a standard normal distribution at initialization and remain fixed throughout training.

For each node i with feature vector $\mathbf{x}_i \in \{0,1\}^d$, a feature hypervector is constructed by selecting the corresponding row of $\mathbf{F}$ for each binary feature value, applying a complex exponential, and averaging across all d features:

$$\mathbf{h}_i^{\text{feat}} = \frac{1}{d} \sum_{k=1}^{d} \exp(j \cdot \mathbf{F}[x_{i,k},\, k]) \tag{5.2}$$

In parallel, a positional hypervector is derived from sinusoidal positional encodings $\mathbf{e}_i \in \mathbb{R}^p$ of node i's index within the state:

$$\mathbf{h}_i^{\text{pos}} = \exp(j \cdot (\mathbf{e}_i \mathbf{P} + \mathbf{b})) \tag{5.3}$$

The feature and positional components are then bound together:

$$\mathbf{h}_i = \mathbf{h}_i^{\text{feat}} \odot \mathbf{h}_i^{\text{pos}} \tag{5.4}$$

To incorporate global network context into each node's representation, the node hypervectors are bundled into a single global state hypervector by summing and normalizing across all N_s devices:

$$\mathbf{g}_s = \frac{1}{N} \sum_{i=1}^{N} \mathbf{h}_i \tag{5.5}$$

This global hypervector is permuted by one position to yield $\tilde{\mathbf{g}}_s = \rho(\mathbf{g}_s)$, ensuring quasi-orthogonality with the individual node hypervectors. Each node hypervector is then binded to this global context:

$$\tilde{\mathbf{h}}_i = \mathbf{h}_i \odot \tilde{\mathbf{g}}_s \tag{5.6}$$

The resulting set $\{\tilde{\mathbf{h}}_i\}_{i=1}^{N}$ constitutes the encoded state, where each hypervector simultaneously represents the local device attributes and their relationship to the global network topology.

5.3.2 Actor — HDC-Based Policy Model

The HDC-based actor implements a factorized policy over the joint device-action space. At each step, the agent selects a device d_i, where $i \in \{1, \ldots, N_{\text{Hosts}}\}$ and an action $a \in \{1, \ldots, A\}$, yielding an effective action space of size $N_{\text{Hosts}} \times A$. The joint log-probability is decomposed as:

$$\log \pi(d_i, a \mid s) = \log P(a \mid d_i) + \log P(d_i \mid s) \tag{5.7}$$

where $P(d_i \mid s)$ models device selection given state s and $P(a \mid d_i)$ models action selection a conditioned on the chosen device.

The actor maintains two complex-valued weight matrices initialized to zero: an action matrix $\mathbf{W}_a \in \mathbb{C}^{A \times D}$ and a device matrix $\mathbf{W}_d \in \mathbb{C}^{1 \times D}$. Logits for both components are computed as scaled real-valued inner products with the complex conjugate of the encoded node hypervectors:

$$\mathbf{a}_i = \frac{1}{D} \operatorname{real}\left(\mathbf{W}_a \, \overline{\tilde{\mathbf{h}}_i}\right), \qquad d_i = \frac{1}{D} \operatorname{real}\left(\mathbf{W}_d \, \overline{\tilde{\mathbf{h}}_i}\right) \tag{5.8}$$

Device log-probabilities are computed via scatter log-softmax over the variable-length device set, and action log-probabilities via standard log-softmax over the fixed action space.

The per-node log-probabilities are restructured from $bm \times A$ into a padded $b \times (M_{\max} \cdot A)$ matrix via the reshape operation, with invalid device-action pairs masked using a filler value of -10^8. During training, actions are sampled from a Categorical distribution over the joint

Table 5.1: Experimental Configuration

Parameter	**SAC**	**HDCyberSAC**
# of Episodes (for convergence)	1.5×10^5	1.5×10^5
Max # of Steps (per episode)	2×10^5	2×10^5
α	4×10^{-1}	4×10^{-1}
α Learning Rate	3×10^{-4}	3×10^{-4}
π Learning Rate	3×10^{-4}	3×10^{-4}
Critic Learning Rate	3×10^{-4}	3×10^{-4}
τ	5×10^{-3}	5×10^{-3}
γ	0.99	0.99
Hidden Size	64	–
HDC Dimension	–	4096
Positional Encoding Dimension	8	8
Batch Size	64	64
Replay Buffer Size	10^6	10^6
NASimEmu Environment	medium	medium

space, while evaluation selects the greedy action.

5.3.3 Critic — HDC-Based Value-Based Model

To mitigate Q-value overestimation, the critic employs a clipped double Q-learning architecture consisting of two independent Q-models, Q_1 and Q_2. Each model maintains a complex-valued weight matrix, where the Q-values are computed as scaled real-valued inner products. The target Q-function returns the element-wise minimum of both models: $Q_{\text{target}} = \min(Q_1, Q_2)$ to form a conservative Q-value estimate. Meanwhile, a Polyak-averaged copy of the Q-function parameters is maintained and periodically updated for stability.

5.4 Evaluation

5.4.1 Experimental Setup

To test HDCyberSAC's efficacy in cybersecurity, we employ the Network Attack Simulator and Emulator environment and conduct our experiments on an AMD Ryzen Threadripper PRO 5965W, which features 24 cores and 48 threads with a 3.8 GHz base clock frequency. For all our experimental results, we run our algorithm over multiple seeds (3-5). In Table 5.1, we include the parameters for training our HDCyberSAC algorithm. The environmental configuration for the *medium* NASimEmu network can be found in the NASimEmu repository [49].

5.5 Results

5.5.1 Experiments

Figure 5.3 and Figure 5.4 demonstrate the learning progression of the HDCyberSAC algorithm, with all results reported across three independent random seeds. Figure 5.3 presents the agent's entropy and actor loss over 125,000 training steps, while Figure 5.4 presents the average episodic rewards over approximately 3,500 training episodes.

As shown in Figure 5.3, the HDCyberSAC agent exhibits the desired learning behavior across all three seeds. Entropy begins at a peak value of approximately 1.25, reflecting broad exploration of the state space during the early stages of training, and gradually decreases toward zero by approximately 75,000–100,000 steps as the policy becomes increasingly deterministic and confident. This progressive reduction in entropy indicates that the agent successfully transitions from exploration to exploitation, ultimately converging on a stable policy. Cor-

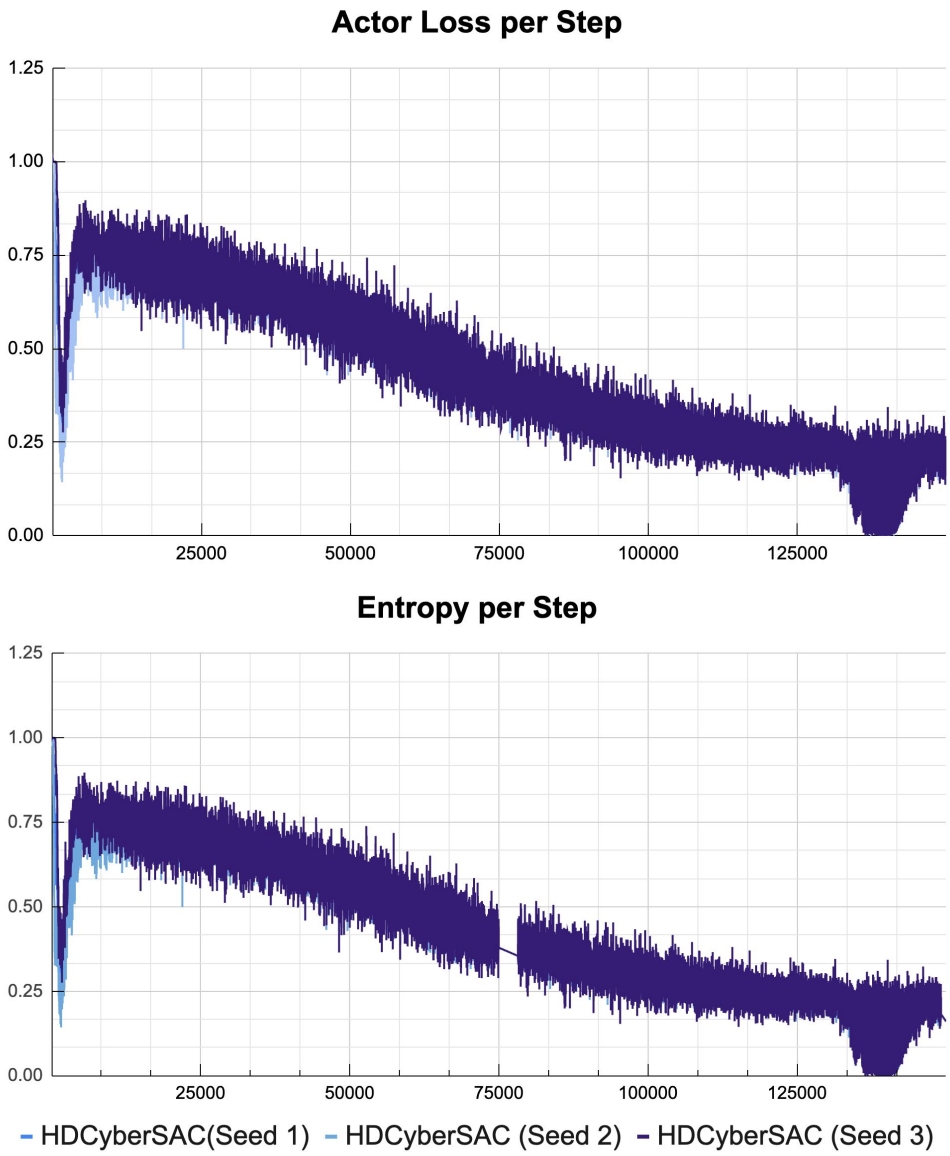


Figure 5.3: (Top) Entropy of HDCyberSAC algorithm and (bottom) Actor Loss across three random seeds.

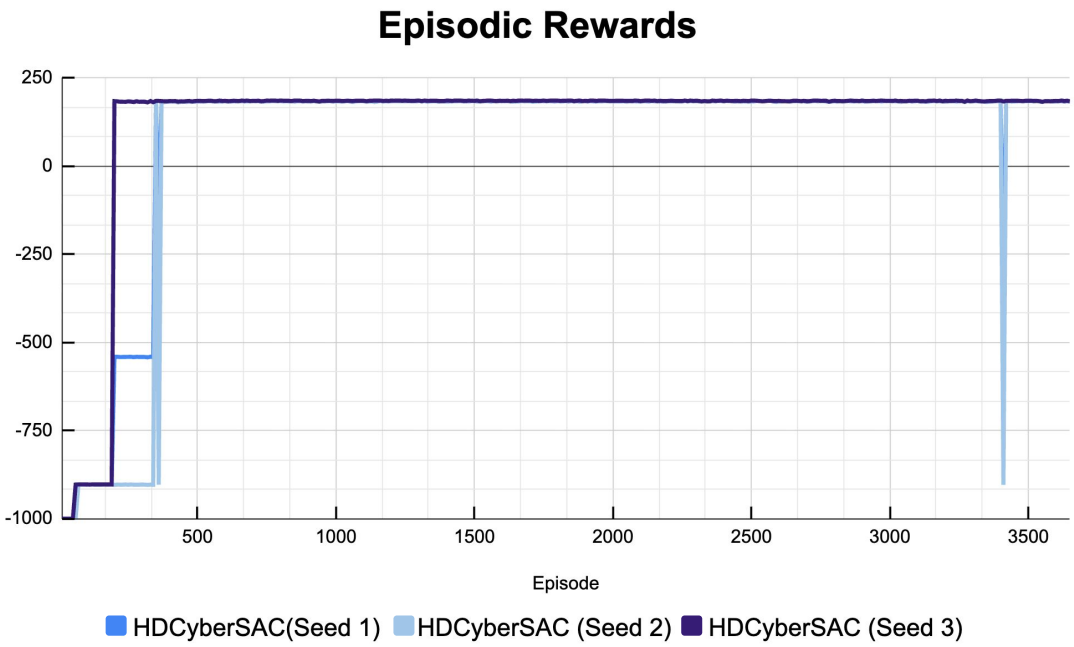


Figure 5.4: Episodic rewards for the HDCyberSAC algorithm across three random seeds.

respondingly, the actor loss begins near 1.25 and steadily decreases toward zero over the course of training, further demonstrating the agent's effective policy optimization. All three seeds exhibit consistent entropy and actor loss trajectories with low variance, demonstrating the stability and reproducibility of the HDCyberSAC training procedure.

Figure 5.4 further confirms successful learning, with episodic rewards converging to the target value of approximately 183 rewards per episode by roughly episode 500 and remaining stable through the remainder of training. The consistent convergence across all three seeds reinforces the robustness of the HDCyberSAC algorithm in learning the target task.

5.6 Related Work

The application of RL to solve cybersecurity problems, including intrusion detection, is a well-explored research area. In parallel, brain-inspired HDC has emerged as a promising

paradigm in recent years for its energy efficiency, enhanced learning quality, and robustness in the machine learning realm. However, the fusion of these two approaches into HDC-based RL, let alone its application in cybersecurity, remains a nascent and sparsely explored field. We synthesize the relevant work in both domains to clearly delineate the unique contribution of our paper.

HDC in Cybersecurity. Initial efforts that demonstrate HDC's potential application in cybersecurity tasks include [87]. This work presents an HDC-based classification model, CyberHD, for detecting cybersecurity intrusion breaches. CyberHD is competitive in terms of accuracy, while maintaining efficient training time, inference latency, and robustness to hardware error, underscoring HDC's hallmark advantages. However, CyberHD is a purely supervised classification approach and does not explore RL-based approaches.

RL in Cybersecurity. A significant body of research has explored the application of RL to cybersecurity, particularly for intrusion detection systems: [26, 27, 6, 79, 40, 2, 24, 3]. These works leverage a number of deep RL algorithms, including Deep Q-Networks and other policy-gradient methods. Notably, the SAC algorithm has also been adopted in security tasks, including in novel detection models [59], optimized gravitational search algorithms [51], and alert prioritization [15, 14]. While these works validate SAC's utility in cybersecurity, they operate within the conventional deep learning paradigm and do not investigate alternative computing paradigms for energy efficiency optimization.

HDC-Based RL for Cybersecurity. To the best of our knowledge, CyberRL [48] is the first work to explore the utilization of HDC-based RL in intrusion detection and the wider scope of cybersecurity. CyberRL adapts a value-based Double Q-Learning algorithm for the HDC framework. Aside from the RL approach, CyberRL differs quite significantly from the work presented in this paper. While CyberRL was evaluated in a simulated network environment [31], we benchmarked our method using the NASimEmu network attack simulator [49]. This environment provides network emulation, offering a higher degree of realism through

components including actual operating systems and services, which is critical for validating the practical applicability of cybersecurity solutions.

5.7 Conclusion

In this paper, we presented HDCyberSAC, a hyperdimensional computing algorithm that leverages the soft-actor algorithm and is designed to effectively solve partially observable penetration testing environments in cybersecurity. We demonstrated its enhanced learning capabilities. Future work involves scaling our approach to larger, more complex networks and extending it to a wider range of pentesting scenarios.

Chapter 6

Summary and Future Work

6.1 Summary

This dissertation explores the extension of HDC to a number of RL algorithms and establishes HDC as a powerful and efficient paradigm for enabling advanced RL on resource-constrained devices. We demonstrated that HDC's advantageous properties of efficiency, robustness, and rapid learning directly address the critical limitations of deep neural networks—namely, high computational cost, sensitivity to noise, and lack of transparency—making it an ideal candidate for the next generation of edge intelligence.

Through a number of contributions, we developed and validated a comprehensive suite of HDC-based RL algorithms. We first established the foundation with a novel positional encoding scheme for value-based RL, which we then extended to a number of frameworks. This included HyperDimensional Hybrid Learning (HDHL) for dynamic resource orchestration in End-Edge-Cloud systems, CyberRL for developing efficient cybersecurity strategies, and the HDCyberSAC algorithm, which applies HDC to actor-critic algorithms for advanced, partially observable cybersecurity environments.

Our contributions across the domains of resource management and cybersecurity demonstrate how HDC-based RL achieves competitive learning performance while also delivering improvements in energy efficiency compared to its deep learning counterparts. These contributions show that HDC is not only a competitive alternative but may also be a *preferred* computing paradigm for transparent and sustainable edge learning.

6.2 Future Work

Naturally, this work presents certain limitations. The application of HDC to more complex RL environments requires advances in the theoretical realm of HDC. Further development in HDC encoding schemes would improve its scalability and enable HDC in other critical domains, where efficiency and robustness are paramount. The inherent transparency of HDC operations also presents a unique opportunity to develop formal explainability (XAI) frameworks that provide human-understandable insights into the agent's learned policy. Finally, there is a plethora of other advanced RL algorithms that could harness HDC to fully explore its versatility.

In conclusion, this dissertation provides compelling evidence that HDC can fundamentally reshape how we design reinforcement learning systems for the edge. By moving away from computationally expensive deep learning models towards efficient, robust, and brain-inspired computation, we pave the way for a future where intelligent decision-making is truly ubiquitous and sustainable.

Bibliography

[1] F. Ahamed and F. Farid. Applying internet of things and machine-learning for personalized healthcare: Issues and challenges. In *2018 International Conference on Machine Learning and Data Engineering (iCMLDE)*, pages 19–21, 2018.

[2] H. Alavizadeh, H. Alavizadeh, and J. Jang-Jaccard. Deep q-learning based reinforcement learning approach for network intrusion detection. *Computers*, 11(3):41, 2022.

[3] H. Alavizadeh, H. Alavizadeh, and J. Jang-Jaccard. Deep q-learning based reinforcement learning approach for network intrusion detection. *Computers*, 11(3), 2022.

[4] N. B. Amor, S. Benferhat, and Z. Elouedi. Naive bayes vs decision trees in intrusion detection systems. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 420–424, 2004.

[5] H. Amrouch, P. R. Genssler, M. Imani, M. Issa, X. Jiao, W. Mohammad, G. Sepanta, and R. Wang. Beyond von neumann era: Brain-inspired hyperdimensional computing to the rescue. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, ASPDAC '23, page 553–560, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Y. Badr. Enabling intrusion detection systems with dueling double deep q-learning. *Digital Transformation and Society*, (ahead-of-print), 2022.

[7] H. Barkam, S. Yun, H. Chen, P. Gensler, A. Mema, A. Ding, G. Michelogiannakis, H. Amrouch, and M. Imani. Reliable hyperdimensional reasoning on unreliable emerging technologies. pages 1–9, 10 2023.

[8] J. Becker, M. Huebner, and M. Ullmann. Power estimation and power measurement of xilinx virtex fpgas: trade-offs and limitations. In *16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.*, pages 283–288. IEEE, 2003.

[9] R. Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[10] M. Botvinick, S. Ritter, J. X. Wang, Z. Kurth-Nelson, C. Blundell, and D. Hassabis. Reinforcement learning, fast and slow. *Trends in cognitive sciences*, 23(5):408–422, 2019.

[11] Y. Bouzida and F. Cuppens. Neural networks vs. decision trees for intrusion detection. In *IEEE/IST workshop on monitoring, attack detection and mitigation (MonAM)*, volume 28, page 29. Citeseer, 2006.

[12] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[13] Z. Chang, S. Liu, X. Xiong, Z. Cai, and G. Tu. A survey of recent advances in edge-computing-powered artificial intelligence of things. *IEEE Internet of Things Journal*, 8(18):13849–13875, 2021.

[14] L. Chavali, T. Gupta, and P. Saxena. Sac-ap: Soft actor critic based deep reinforcement learning for alert prioritization. In *2022 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2022.

[15] L. Chavali, A. Krishnan, P. Saxena, B. Mitra, and A. Sreevallabh Chivukula. Off-policy actor-critic deep reinforcement learning methods for alert prioritization in intrusion detection systems. *Computers Security*, 142:103854, 2024.

[16] H. Chen and M. Imani. Density-aware parallel hyperdimensional genome sequence matching. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–4. IEEE, 2022.

[17] H. Chen, M. Issa, Y. Ni, and M. Imani. Darl: Distributed reconfigurable accelerator for hyperdimensional reinforcement learning. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery.

[18] H. Chen, Y. Kim, E. Sadredini, S. Gupta, H. Latapie, and M. Imani. Sparsity controllable hyperdimensional computing for genome sequence matching acceleration. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2023.

[19] H. Chen, M. H. Najafi, E. Sadredini, and M. Imani. Full stack parallel online hyperdimensional regression on fpga. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 517–524. IEEE, 2022.

[20] H. Chen, Y. Ni, W. Huang, and M. Imani. Scalable and interpretable brain-inspired hyper-dimensional computing intelligence with hardware-software co-design. In *2024 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE, 2024.

[21] H. Chen, A. Zakeri, F. Wen, H. E. Barkam, and M. Imani. Hypergraf: Hyperdimensional graph-based reasoning acceleration on fpga. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 34–41. IEEE, 2023.

[22] W. Y. Chung, H. Errahmouni Barkam, T. Das, and M. Imani. Robust reasoning and learning with brain-inspired representations under hardware-induced nonlinearities. In *Proceedings of the Great Lakes Symposium on VLSI 2025*, GLSVLSI '25, page 968–975, New York, NY, USA, 2025. Association for Computing Machinery.

[23] K. A. Da Costa, J. P. Papa, C. O. Lisboa, R. Munoz, and V. H. C. de Albuquerque. Internet of things: A survey on machine learning-based intrusion detection approaches. *Computer Networks*, 151:147–157, 2019.

[24] Q.-V. Dang and T.-H. Vo. Reinforcement learning for the problem of detecting intrusion in a computer system. pages 755–762. Springer Singapore, 2022.

[25] D. Dasgupta, Z. Akhtar, and S. Sen. Machine learning in cybersecurity: a comprehensive survey. *The Journal of Defense Modeling and Simulation*, 19(1):57–106, 2022.

[26] A. Dutta, S. Chatterjee, A. Bhattacharya, and M. Halappanavar. Deep reinforcement learning for cyber system defense under dynamic adversarial uncertainties, 2023.

[27] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[28] L. Ge and K. K. Parhi. Classification using hyperdimensional computing: A review. *IEEE Circuits and Systems Magazine*, 20(2):30–47, 2020.

[29] L. Ge and K. K. Parhi. Classification using hyperdimensional computing: A review. *IEEE Circuits and Systems Magazine*, 20(2):30–47, 2020.

[30] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[31] K. Hammar and R. Stadler. Finding effective security strategies through reinforcement learning and Self-Play. In *International Conference on Network and Service Management (CNSM 2020) (CNSM 2020)*, Izmir, Turkey, Nov. 2020.

[32] K. Hammar and R. Stadler. Learning near-optimal intrusion responses against dynamic attackers, 2023.

[33] E. Hassan, Z. Zou, H. Chen, M. Imani, Y. Zweiri, H. Saleh, and B. Mohammad. Efficient event-based robotic grasping perception using hyperdimensional computing. *Internet of Things*, 26:101207, 2024.

[34] H. Hasselt. Double q-learning. *Advances in neural information processing systems*, 23:2613–2621, 2010.

[35] Y. He et al. Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach. *IEEE Communications Magazine*, 55(12):31–37, 2017.

[36] P. Henderson et al. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, 2018.

[37] A. Hernandez-Cane, N. Matsumoto, E. Ping, and M. Imani. Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 56–61. IEEE, 2021.

[38] T. Hester et al. Deep q-learning from demonstrations. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[39] A. G. Howard et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. 2017.

[40] Y.-F. Hsu and M. Matsuoka. A deep reinforcement learning approach for anomaly network intrusion detection system. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–6, 2020.

[41] M. Imani et al. Voicehd: Hyperdimensional computing for efficient speech recognition. In *ICRC*, pages 1–8. IEEE, 2017.

[42] M. Imani et al. Control of gene regulatory networks using bayesian inverse reinforcement learning. *IEEE/ACM transactions on computational biology and bioinformatics*, 16(4):1250–1261, 2018.

[43] M. Imani et al. Partially-observed discrete dynamical systems. In *2021 American Control Conference (ACC)*, pages 310–315. IEEE, 2021.

[44] M. Imani et al. Scalable inverse reinforcement learning through multifidelity bayesian optimization. *IEEE transactions on neural networks and learning systems*, 2021.

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

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

[47] M. Issa, S. Shahhosseini, Y. Ni, T. Hu, D. Abraham, A. M. Rahmani, N. Dutt, and M. Imani. Hyperdimensional hybrid learning on end-edge-cloud networks. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 652–655, 2022.

[48] M. A. Issa, H. Chen, J. Wang, and M. Imani. Cyberrl: Brain-inspired reinforcement learning for efficient network intrusion detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(1):241–250, 2025.

[49] J. Janisch, T. Pevný, and V. Lisý. Nasimemu: Network attack simulator emulator for training agents generalizing to novel scenarios, 2023.

[50] N. Jay et al. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*. PMLR, 2019.

[51] L. Jin, R. Fan, X. Han, and X. Cui. Igsa-sac: a novel approach for intrusion detection using improved gravitational search algorithm and soft actor-critic. *Frontiers in Computer Science*, 7:1574211, 2025.

[52] P. Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1(2):139–159, 2009.

[53] V. Kathail. Xilinx vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 173–174, 2020.

[54] H. Khelifi, S. Luo, B. Nour, A. Sellami, H. Moungla, S. H. Ahmed, and M. Guizani. Bringing deep learning at the edge of information-centric internet of things. *IEEE Communications Letters*, 23(1):52–55, 2019.

[55] D. S. Kim and J. S. Park. Network-based intrusion detection with support vector machines. In *International conference on information networking*, pages 747–756. Springer, 2003.

[56] Y. G. Kim and C.-J. Wu. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1082–1096, 2020.

[57] H. Lee, J. Kim, H. Chen, A. Zeira, N. Srinivasa, M. Imani, and Y. Kim. Comprehensive integration of hyperdimensional computing with deep learning towards neuro-symbolic ai. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[58] H. Li, Q. Zhang, and D. Zhao. Deep reinforcement learning-based automatic exploration for navigation in unknown environment. *IEEE Transactions on Neural Networks and Learning Systems*, 31(6):2064–2076, 2020.

[59] Z. Li, C. Huang, S. Deng, W. Qiu, and X. Gao. A soft actor-critic reinforcement learning algorithm for network intrusion detection. *Computers Security*, 135:103502, 2023.

[60] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. pages 389–400, 06 2018.

[61] Y. Lu and L. Da Xu. Internet of things (iot) cybersecurity research: A review of current research topics. *IEEE Internet of Things Journal*, 6(2):2103–2115, 2018.

[62] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[63] A. Moin et al. A wearable biosensing system with in-sensor adaptive machine learning for hand gesture recognition. *Nature Electronics*, 2021.

[64] B. A. Mudassar, J. H. Ko, and S. Mukhopadhyay. Edge-cloud collaborative processing for intelligent internet of things: A case study on smart surveillance. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[65] Y. Ni, D. Abraham, M. Issa, A. Hernández-Cano, M. Imani, P. Mercati, and M. Imani. Dynamic mac protocol for wireless spectrum sharing via hyperdimensional self-learning. *IEEE Access*, 12:138519–138534, 2024.

[66] Y. Ni, D. Abraham, M. Issa, Y. Kim, P. Mercati, and M. Imani. Efficient off-policy reinforcement learning via brain-inspired computing. In *Proceedings of the Great Lakes Symposium on VLSI 2023*, pages 449–453, 2023.

[67] Y. Ni, H. Chen, P. Poduval, Z. Zou, P. Mercati, and M. Imani. Brain-inspired trustworthy hyperdimensional computing with efficient uncertainty quantification. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 01–09. IEEE, 2023.

[68] Y. Ni, M. Issa, D. Abraham, M. Imani, X. Yin, and M. Imani. Hdpg: Hyperdimensional policy-based reinforcement learning for continuous control. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, page 1141–1146, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Y. Ni, Y. Kim, T. Rosing, and M. Imani. Algorithm-hardware co-design for efficient brain-inspired hyperdimensional learning on edge. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 292–297. IEEE, 2022.

[70] Y. Ni, N. Lesica, F.-G. Zeng, and M. Imani. Neurally-inspired hyperdimensional classification for efficient and robust biosignal processing. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[71] Y. Ni, Y. Yang, H. Chen, X. Wang, N. Lesica, F.-g. Zeng, and M. Imani. Hyperdimensional brain-inspired learning for phoneme recognition with large-scale inferior colliculus neural activities. *IEEE Transactions on Biomedical Engineering*, 71(11):3098–3110, 2024.

[72] B. Peng, X. Li, J. Gao, J. Liu, and K. Wong. Integrating planning for task-completion dialogue policy learning. *CoRR*, abs/1801.06176, 2018.

[73] P. Poduval et al. Graphd: Graph-based hyperdimensional memorization for brain-like cognitive learning. *Frontiers in Neuroscience*, page 5, 2022.

[74] P. Poduval, M. Issa, F. Imani, C. Zhuo, X. Yin, H. Najafi, and M. Imani. Robust in-memory computing with hyperdimensional stochastic representation. In *2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 1–6, 2021.

[75] A. Rahimi et al. Hyperdimensional computing for blind and one-shot classification of eeg error-related potentials. *Mobile Networks and Applications*, 25(5):1958–1969, 2020.

[76] 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*, ISLPED '16, page 64–69, New York, NY, USA, 2016. Association for Computing Machinery.

[77] A.-R. Sadeghi, C. Wachsmann, and M. Waidner. Security and privacy challenges in industrial internet of things. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[78] J. Schwartz and H. Kurniawati. Autonomous penetration testing using reinforcement learning. *CoRR*, abs/1905.05965, 2019.

[79] K. Sethi, Y. V. Madhav, R. Kumar, and P. Bera. Attention based multi-agent intrusion detection systems using reinforcement learning. *Journal of Information Security and Applications*, 61:102923, 2021.

[80] M. Sewak, S. K. Sahay, and H. Rathore. Deep reinforcement learning for cybersecurity threat detection and protection: A review. In *Secure Knowledge Management In The Artificial Intelligence Era*, pages 51–72. Springer International Publishing, 2022.

[81] D. Silver et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[82] M. T. J. Spaan. *Partially Observable Markov Decision Processes*, pages 387–414. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[83] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In B. Porter and R. Mooney, editors, *Machine Learning Proceedings 1990*, pages 216–224. Morgan Kaufmann, San Francisco (CA), 1990.

[84] L. Tai and M. Liu. Towards cognitive exploration through deep reinforcement learning for mobile robots. *CoRR*, abs/1610.01733, 2016.

[85] N. Thompson, K. Greenewald, K. Lee, and G. Manso. The computational limits of deep learning, 07 2020.

[86] W. Uther. *Markov Decision Processes*, pages 642–646. Springer US, Boston, MA, 2010.

[87] J. Wang, H. Chen, M. Issa, S. Huang, and M. Imani. Late breaking results: Scalable and efficient hyperdimensional computing for network intrusion detection. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–2, 2023.

[88] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(2):869–904, 2020.

[89] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[90] W. Xu, V. Swaminathan, S. Pinge, S. Fuhrman, and T. Rosing. Hypermetric: Robust hyperdimensional computing on error-prone memories using metric learning. pages 243–246, 11 2023.

[91] Y. Xu, H. G. Lee, Y. Tan, Y. Wu, X. Chen, L. Liang, L. Qiao, and D. Liu. Tumbler: Energy efficient task scheduling for dual-channel solar-powered sensor nodes. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[92] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[93] S. Yue, D. Zhu, Y. Wang, and M. Pedram. Reinforcement learning based dynamic power management with a hybrid power supply. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 81–86, 2012.

[94] S. Yun, H. E. Barkam, P. R. Genssler, H. Latapie, H. Amrouch, and M. Imani. Hyperdimensional computing for robust and efficient unsupervised learning. In *2023 57th Asilomar Conference on Signals, Systems, and Computers*, pages 281–288, 2023.

[95] A. Zakeri, Z. Zou, H. Chen, H. Latapie, and M. Imani. Conjunctive block coding for hyperdimensional graph representation. *Intelligent Systems with Applications*, 22:200353, 2024.

[96] Z. Zou, H. Chen, P. Poduval, Y. Kim, M. Imani, E. Sadredini, R. Cammarota, and M. Imani. Biohd: an efficient genome sequence search platform using hyperdimensional memorization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 656–669, 2022.

[97] Z. Zou, Y. Kim, F. Imani, H. Alimohamadi, R. Cammarota, and M. Imani. Scalable edge-based hyperdimensional learning system with brain-like neural adaptation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.