UNIVERSITY OF CALIFORNIA,
IRVINE

Online Learning for Orchestrating Deep Learning Inference at Edge

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Sina Shahhosseini

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Amir M. Rahmani
Professor Fadi Kurdahi

2023

DEDICATION

To my parents and my sisters for all their support, kindness, and love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I am grateful to my advisor, professor Nikil Dutt for his patience, support, and guidance.

I would like to thank Prof. Amir Rahmani and Prof. Fadi Kurdahi for their participation in the thesis committee and providing helpful feedback.

In addition, I would like to thank ACM, IEEE, and ELSEVIER for permissions to include the following contents in this thesis:

- The text of Chapter 2 is a reprint of the material as it appears in [108], used with permission from ELSEVIER. The co-authors listed in this publication are A. Anzanpour, I. Azimi, S. Labbaf, D. Seo, S. Lim, P. Liljeberg, N. Dutt, and A. Rahmani.

- The text of Chapter 3 is a reprint of the material as it appears in [111], used with permission from ACM. The co-authors listed in this publication are D. Seo, A. Kanduri, T. Hu, S. Lim, B. Donyanavard, A. Rahmani, and N. Dutt.

- The text of Chapter 4 is a reprint of the material as it appears in [110], used with permission from IEEE. The co-authors listed in this publication are T. Hu, D. Seo, A. Kanduri, B. Donyanavard, A. Rahmani, and N. Dutt.

- The text of Chapter 5 is a reprint of the material as it appears in [40], used with permission from IEEE. The co-authors listed in this publication are M. Issa, Y. Ni, T. Hu, D. Abraham, A. Rahmani, N. Dutt, and M. Imani.

Finally, my work would not have been possible without funding from the Electrical Engineering and Computer Science department.

VITA

Sina Shahhosseini

EDUCATION

Doctor of Philosophy in Computer Engineering **2023**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Engineering **2018**
University of California, Irvine *Irvine, CA*

Master of Science in Electrical Engineering **2015**
Sharif University of Technology *Tehran*

Bachelor of Science in Electrical Engineering **2013**
University of Tehran *Tehran*

RESEARCH EXPERIENCE

Graduate Research Assistant **2019–2023**
University of California, Irvine *Irvine, CA*

TEACHING EXPERIENCE

Teaching Assistant **2019–2022**
University of California, Irvine *Irvine, CA*

REFEREED JOURNAL PUBLICATIONS

Online Learning for Orchestration of Inference in Multi-User End-Edge-Cloud Networks **2022**
ACM Transaction on Embedded Computing Systems

Exploring Energy Efficient Architectures for RLWE Lattice-Based Cryptography **2021**
Journal of Signal Processing Systems

Exploring Computation Offloading in IoT Systems **2020**
Journal of Information Systems

On the feasibility of SISO control-theoretic DVFS for power capping in CMPs **2019**
Microprocessors and Microsystems

Communication at the Speed of Light (CaSoL): A New Paradigm for Designing Global Wires **2019**
IEEE Transaction Electron Device

REFEREED CONFERENCE PUBLICATIONS

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

Hybrid Learning for Orchestrating Deep Learning Inference in Multi-user Edge-cloud Networks **2022**
International Symposium on Quality Electronic Design (ISQED)

Flexible and Personalized Learning for Wearable Health Applications using HyperDimensional Computing **2022**
Great Lakes Symposium on VLSI (GLSVLSI)

AMSER: Adaptive Multimodal Sensing for Energy Efficient and Resilient eHealth Systems **2022**
Design, Automation, and Test in Europe (DATE)

Exploring Energy Efficient Quantum-resistant Signal Processing Using Array Processors **2020**
International Conference on Acoustics, Speech, and Signal Processing (ICASSP)

Partition Pruning: Parallelization-Aware Pruning for Deep Neural Networks **2020**
Conference on Parallel, Distributed and Network-Based Processing (PDP)

An Edge-Assisted and Smart System for Real-Time Pain Monitoring **2019**
Connected Health: Applications, Systems and Engineering Technologies (CHASE)

Dynamic Computation Migration at the Edge: Is There an Optimal Choice? **2019**
Great Lakes Symposium on VLSI (GLSVLSI)

Dependability evaluation of SISO control-theoretic power managers for processor architectures **2017**
Nordic Circuits and Systems Conference (NORCAS)

ABSTRACT OF THE DISSERTATION

Online Learning for Orchestrating Deep Learning Inference at Edge

By

Sina Shahhosseini

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2023

Professor Nikil Dutt, Chair

Deep-learning-based intelligent services have become prevalent in cyber-physical applications including smart cities and health-care. Resource-constrained end-devices must be carefully managed in order to meet the latency and energy requirements of computationally-intensive deep learning services. Collaborative end-edge-cloud computing for deep learning provides a range of performance and efficiency that can address application requirements through computation offloading. The decision to offload computation is a communication-computation co-optimization problem that varies with both system parameters (e.g., network condition) and workload characteristics. On the other hand, deep learning model optimization provides another source of tradeoff between latency and model accuracy. An end-to-end decision-making solution that considers such computation-communication problem is required to synergistically find the optimal offloading policy and model for deep learning services. To this end, we propose a reinforcement-learning-based computation offloading solution that learns optimal offloading policy considering deep learning model selection techniques to minimize response time while providing sufficient accuracy. We demonstrate the efficacy of our strategies through experimental comparison with state-of-the-art RL-based inference orchestration. In addition, we investigate applying intelligent orchestration strategy in eHealth monitoring systems as a case study.

Chapter 1

Introduction

1.1 Overview

The Internet of Things (IoT) has become a promising solution to improve efficiency or provide novel solution in many industrial areas, such as automotive, health-care, and home automation. Demand for comprehensive solutions, however, requires more complex IoT systems [7]. For this reason, IoT devices need to interact with cloud servers and other smart IoT devices via the Internet and local networks to offer smart and connected solutions [28]. According to Cisco, 500 billion connected devices are expected by 2030, which represents a substantial increase in the scale and the complexity of existing computing and communication systems [25].

Machine learning (ML) is advancing real-time and interactive user services in IoT systems. ML applications are primarily deployed on cloud infrastructure to meet the compute intensity and storage requirements of ML algorithms, and address the resource constraints of user-end wearable sensory devices [11]. However, unpredictable network constraints including variable signal strength and availability of the network affect real-time delivery of cloud services [51].

The edge computing paradigm allows deployment of ML applications closer to the user-end devices, minimizing the latency of service delivery, reducing the total network load, and alleviating privacy concerns. This thesis addresses challenges of edge computing from three perspectives: systems, applications, and edge orchestration.

Systems Perspective Collaborative sensor-edge-cloud architecture presents multiple execution choices for workload partitioning and compute placement including execution on a single sensor, edge, cloud nodes, and any possible combinations of these devices. Considering the variable accuracy nature of ML algorithms, these execution choices expose a wide range of energy-performance-accuracy trade-off space. Choosing an optimal execution option under varying system dynamics, available energy budget of devices, network constraints, and error resilience of ML workloads is a complex run-time challenge.

Application Perspective Depending on the ML model employed, different applications feature varying compute, data, and communication intensities. At an application level, there is diversity in terms of sensitivities to latency, throughput, infrastructure availability, and accuracy. Further, different application execution choices result in different energy consumption patterns. Considering these application level variations, the choice of execution of an application is a subject of multiple factors varying at run-time.

Edge Orchestration Edge orchestration techniques handle workload partitioning, distribution, and scheduling of ML workloads, considering both applications' and systems' perspectives. Understanding the varying compute intensities of applications, latency and throughput requirements, user-interaction and responsiveness expectations, quality of interconnection network including signal strength, availability, load balancing, compute and storage capacities of underlying hardware elements put together makes application orchestration a stochastic process [107]. To maximize the efficiency of edge-enabled services, run-time solutions that holistically consider both requirements and opportunities vertically across the user, device, application, network, edge node, and platform layers are necessary [106].

1.2 Thesis Contributions

Determining the optimal orchestration policy for an unknown and dynamic system is of utmost importance, given that the environment's dynamicity, including changes in network conditions, workload arrivals at computing nodes, user traffic, and application characteristics, is subject to continual variation. Most current solutions are based on design time optimization, without considerations on varying system dynamics at runtime [139, 15, 73, 87, 17, 18, 10, 47, 133, 112, 63]. A complex system that runs a variety of applications in uncertain environmental conditions requires dynamic control to offer high-performance or low-power guarantees [111, 110, 109]. Considering the run-time variation of system dynamics, and making an optimal orchestration choice requires intelligent monitoring, analysis, and decision making. Existing heuristic and rule-based orchestration methods require an extensive design space exploration to make optimal compute placement decisions. Further, such a solution based on exhaustive search at run-time becomes practically infeasible for latency critical services. In this context, different offline and online machine learning models have been adopted for run-time resource management of distributed systems, to handle the complexity of orchestration choices. Among these models, reinforcement learning approach (RL) is effective in developing an understanding and interpreting varying system dynamics [78, 92]. By utilizing reinforcement learning, it becomes possible to identify intricate relationships between critical system parameters and make online decisions that optimize various objectives such as response time, energy usage, and quality of service [116].

The main contributions of this dissertation are outlined below:

- **Investigation on impact of various system parameters:** The majority of existing literature presents efficient solutions considering a limited number of parameters (e.g., computation capacity and network bandwidth) neglecting the effect of the application characteristics and dataflow configuration. In this work, we explore the impact of the

computation offloading on total application response time in end-edge-cloud networks considering more realistic parameters, e.g., application characteristics, system complexity, communication cost, and dataflow configuration. This work also highlights the impact of a new application characteristic parameter defined as Output-Input Data Generation (OIDG) ratio and dataflow configuration on the system behavior.

- **Runtime DL inference orchestration framework:** This work implements a runtime orchestration framework for DL inference services on multi-user end-edge-cloud networks. The majority of orchestration solutions that have been studied are evaluated through numerical methods, while some have been suggested and assessed using simulators. Moreover, there is a dearth of literature on actual hardware implementations for online learning frameworks. The proposed solution takes into account the entire service process, from when a request is initiated by the end-node device until the results are delivered back to the device on real test-bed.

- **Online learning for cross-layer optimization:** This work proposes a reinforcement-learning-based computation offloading solution that learns optimal offloading policy considering deep learning model selection techniques. The orchestrator uses reinforcement learning to optimize response time provided accuracy requirements. Online solutions have not previously coordinated offloading and model optimizations together. The related works rely on only computation offloading strategy. This work considers both computation offloading and application-level adjustment together in order to achieve required QoS.

- **Hybrid learning for orchestration:** Identifying optimal orchestration considering the cross-layer opportunities and requirements in the face of varying system dynamics is a challenging multi-dimensional problem. While Reinforcement Learning (RL) approaches have been proposed earlier, they suffer from a large number of trial-and-errors during the learning process resulting in excessive time and resource consumption. This

work proposes a Hybrid Learning orchestration framework that reduces the number of interactions with the system environment by combining model-based and model-free reinforcement learning.

- **Hyperdimensional computing for reinforcement learning:** Training deep neural networks, which is necessary for Deep Q-Learning algorithms, including the state-of-the-art DQL, is computationally expensive; Hyperdimensional Computing (HDC) is a computationally efficient learning paradigm that produces higher quality learning than its neural network counterparts. This work presents a new method, hyperdimensional Q-Learning and Hybrid Learning to efficiently learn the policy.

- **Intelligent sense-compute co-optimization:** ML-driven smart healthcare applications have different input data characteristics, computational requirements, and quality metrics. Continuous stream of input data, varying network conditions, and computational requirements of different ML models create dynamic workload scenarios. At an application-level, requirements include higher prediction accuracy of ML models, latency of inferencing results from ML models, resilience, and an overall higher quality of service. At a system-level, requirements include availability of compute nodes in edge and cloud layers, compute capabilities of edge nodes to meet performance requirements of ML models, network utilization, and overall energy efficiency. This work presents a joint sensing and sense-making co-optimization using an RL-agent as a general solution for eHealth services considering energy consumption while meeting accuracy requirements.

The rest of the thesis is organized as follows:

- **Chapter 2:** Provides an introduction to Computation Offloading problem in end-edge-cloud systems. It explores the impact of the computation offloading on total application response time in end-edge-cloud network considering variety of parameters,

e.g., application characteristic, system complexity, communication cost, and dataflow configuration.

- **Chapter 3:** Implements a runtime orchestration scheme for DL inference services on multi-user end-edge-cloud networks. It employes reinforcement learning to perform orchestration of DL inference in end-edge-cloud networks.

- **Chapter 4:** Proposes a hybrid learning approach to accelerate the RL learning process and reduce direct sampling during learning.

- **Chapter 5:** Proposes hyperdimensional reinforcement learning technique to efficiently orchestrate an end-edge-cloud network.

- **Chapter 6:** Proposes a reinforcement learning scheme for qualitative feature selection, weighted prioritization of input modalities, input-driven MMML model selection, and sensor configuration setting, based on run-time monitoring, and analysis of input modalities.

- **Chapter 7:** Concludes the work, and provides future research directions.

Chapter 2

Introduction to Computation Offloading

Modern IoT systems comprise three main layers including sensor layer, fog layer, and cloud layer. These layers may include different sensing devices, computational and energy resources, storage capacity, and connectivity infrastructure [7]. The sensor layer is often in charge of data collection, leveraging a local sensor network. The sensor layer is resource-constrained in terms of processing power and energy budget. Some works have addressed such limitations by offloading computations to the upper layers (i.e., fog and cloud) [99]. Offloading computation is based on the assumption that i) the response time is mostly determined by computation time and ii) shifting the computation toward upper layers can reduce the total response time. However, this assumption may not always hold, as the migration is often a communication-computation co-optimization problem [109]. More precisely, the response time of an application, in addition to being the function of execution time, is a function of communication cost as well. In other words, any significant increase in the transmission time leads to an increase in the response time [86]. In this chapter we only focus on exploring the impact of computation offloading on response time. However,

power consumption is another important co-design aspects of computation offloading for IoT systems which needs to be investigated in the future works.

The response time is defined as the time difference between the moment when data is collected for decision making and the moment when the result is delivered to the consumer. Thus, different data flows from sensors to consumers may cause changes in the response time. Therefore, an end-to-end analysis of IoT systems that considers a service from data collection to actuation/ notification is essential. Furthermore, previous studies have neglected the impact of running applications with different characteristics in IoT systems. In other words, each application may need different input data size and generate different output data size after the execution leading to different communication cost in IoT systems. We define this application characteristic as OIDG ratio (γ) referring to the ratio of output data volume to input data volume generated in an application. Even though there have been recently some efforts to co-optimize communication and computation in a coupled manner, the literature lacks a comprehensive solution that considers system parameters (e.g., application characteristic (OIDG) and dataflow configuration) in a holistic manner. The majority of existing literature only focuses on communication-computation optimization with limited system parameters such as processing capacity and network bandwidth [113, 102, 58, 50, 19, 65, 64, 141].

In addition, finding optimal computation offloading policy for an unknown and dynamic system is critical since dynamicity of environment e.g., network condition, workload arrival at computing nodes, user traffic, and application characteristics changes over time. Most current solutions are based on design-time optimization suffering from the condition variation during the runtime [139, 15, 73, 87, 17, 18, 10, 47, 133, 112, 63]. A complex system that runs a variety of applications in uncertain environmental conditions requires dynamic control to offer high-performance or low-power guarantees [109, 106, 85].

In this chapter, we explore computation offloading through a computation- communication co-optimization lens and in an end-to-end fashion. We highlight the impact of aforementioned

system parameters (often neglected) on the response time. In addition, we examine a proof-of-concept dynamic computation offloading strategy in a case study to show IoT systems are required to be dynamically controlled at run-time. In short, we make the following key contributions:

- Explore the impact of the computation offloading on total application response time in three-layer IoT systems considering variety of parameters, e.g., application characteristic, system complexity, communication cost, and dataflow configuration.
- Investigate the impact of the new application characteristic parameter (i.e., OIDG) and the dataflow configuration on system behavior.
- Examine a proof-of-concept end-to-end dynamic computation offloading solution considering the aforementioned influential parameters. We compare the dynamic strategy with three static computing schemes to show the design time models are insufficient for IoT systems with dynamic behavior.

The rest of this chapter is organized as follows: Section 2.1, briefly introduces modern IoT architecture and highlights the advantage of the three-layer IoT architecture and computation offloading technique. In Section 2.2, we formulate the system behavior in two common Dataflow configurations. In Section 2.3, we explore an impact of computation offloading on different parameters. In Section 2.4, we examine the dynamic computation offloading system. Finally, we conclude the chapter in Section 2.5.

2.1 Background

In this section, we first briefly describe the three-layer IoT architecture. Then, we outline the background of computation offloading in IoT systems.

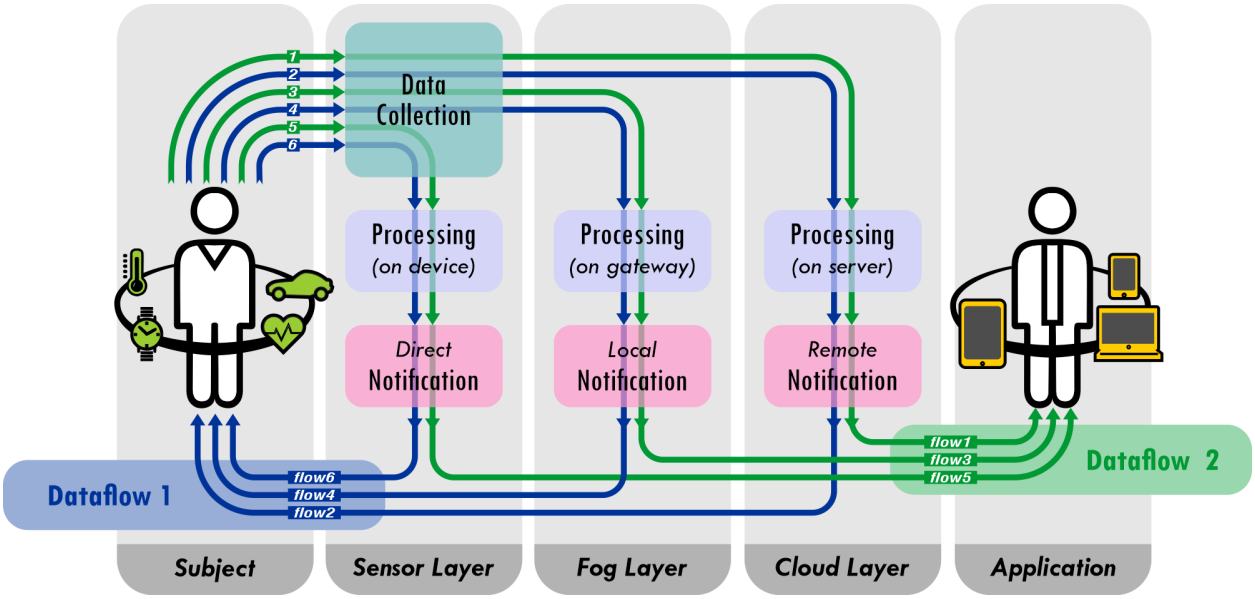


Figure 2.1: IoT System Architecture

2.1.1 Three-layer IoT Architecture

An IoT-based system conventionally includes a three-layer architecture. The devices and processing units are located in the three layers, requiring communication methods according to their functions and physical or virtual locations [7]. In general and from the functionality perspective, the lower layer –including sensing, actuating, and informing devices– is entitled as the sensor layer. The upper layer that contains computing and storage devices is the cloud layer, and the intermediate devices is the fog layer (see Figure 2.1).

The sensor layer consists of sensor nodes that collect data from the physical world. The devices may also provide actuation control or notification as a result of processing the collected data. The sensing devices are expected to be relatively small, connected to the rest of the system wirelessly. To communicate with the other layers, a sensor node requires a transmission module as well as a small and power-efficient processing unit. Due to the limited battery, the sensing devices are constrained by their energy sources, processing powers, and data transmission capacities.

The fog layer is defined as an intermediate layer to regulate the data transmission and processing loads, since constant direct communication with the server may not be feasible or reasonable for resource-constrained sensing devices. The fog layer consists of one or more gateway devices that are physically close to the sensing devices [99]. This short distance to the data sources significantly reduces the data transmission power consumption of the sensing devices and enables fast responses. The processing power of the gateway devices in the fog layer is higher than the sensing devices, enabling local data analysis before sending the data to the cloud server. This pre/local processing method, which is known as "fog computing," reduces data traffic, the load on the cloud server, and the response time. The gateway and sensing devices are connected to a local network. Therefore, the system is still able to provide the service in case of loss of Internet connection (i.e., connection to the cloud servers) [103, 120, 9].

At the cloud layer, the main processing unit of an IoT-based system is the cloud server. The cloud servers are considered as devices with more powerful processing capability [121]. Thanks to a centralized database of the incoming data, heavy data analytic approaches can be performed on a specific source of data, allowing a comparison with other data sources.

In this chapter, we focus on the computation offloading in the IoT-based systems, referring to how data is processed and transmitted over the IoT network: the sensor, fog, and cloud layers.

2.1.2 Computation Offloading

Resource allocation is classified into three main categories as resource placement, resource scheduling, and computation offloading. Resource placement is about where and how resources are placed in IoT systems. It aims to find optimal set resources in IoT systems to execute tasks or applications while satisfying QoS (Quality of Service) requirements by

optimizing specific objective function (minimizing latency, minimizing energy consumption, etc). Resource scheduling or scheduling in resource allocation is to determine when and how many resources to allocate in IoT systems. The resource scheduling determines optimal scheduling of tasks, services, or applications to be executed on resources in order to meet QoS requirements [61].

Computation offloading is to determine where and how many resources can be moved to execute tasks or applications. The technique in IoT context is the transfer of resource-intensive computational tasks to a separate external device in the network. The technique of offloading computation over a network can provide computing power and overcome the limitation of an IoT-based device such as computational power, storage, and energy. Optimizing the computation offloading problem can be addressed by finding the best solution for the following questions:

(i) where to offload: the scheduler should determine where the computation is offloaded, depending on the variety of parameters such as objectives, availability of resources, and required computation capacity for performing computations. Some studies have addressed this problem with optimally offloading workloads to more capable computing resources [133, 15].

(ii) when to offload: the scheduler should determine when the computation is offloaded to upper layers to achieve the required QoS due to many uncertainties in the system and environment such as network congestion, overloaded workload, and device battery. Some studies have addressed this problem with optimal time scheduling of offloading computations [129, 64].

(iii) what to offload: the computation scheduler should determine what portion of workloads is offloaded to upper layers. According to this fact, offloading solutions can be classified into two classes: (i) full offloading where the whole workloads are offloaded to external re-

Notation	Description
A_i	Application i to be executed on S_i which is defined as a tuple $\{D_{in}, D_{out}\}$
α	Decision offloading tuple
β	Packet loss ratio in connectivity between the sensor layer and fog layer
γ	OIDG ratio in application A
T_t^{sg}	Transmission time between the sensor layer to gateway layer
T_t^{gs}	Transmission time between the gateway layer to sensor layer
T_t^{cg}	Transmission time between the cloud layer to gateway layer
T_t^{gc}	Transmission time between the gateway layer to cloud layer
T_t^{ca}	Transmission time between the cloud layer to application layer
T_e^S	Execution time at the sensor node
T_e^G	Execution time for arrival workloads at the gateway device
T_e^C	Execution time for arrival workloads at the cloud device
N	Number of connected sensors
D_{in}	Input data size in A
D_{out}	Output data size in A
T_r^s	Total response time in the sensor computing scheme A
T_r^g	Total response time in the gateway computing scheme A
T_r^c	Total response time in the cloud computing scheme A

Table 2.1: Notations Descriptions

sources such as fog or cloud layers and (ii) partial offloading where workloads are partitioned into parts to be executed locally or externally. Many works have been addressed this optimization problem by the partial offloading method such as [125, 102, 63, 141]. Besides, some studies that have applied the full offloading method in their problems [113, 139].

2.2 System Model

In this section, we investigate response time behavior in an three-layer IoT system which uses the computation offloading technique. First, we briefly explain the system components and then we describe the computation offloading model which is applied to our system. Finally, the response time is modeled in two common dataflow configurations. The computing devices in three-layer IoT system are represented by (S,G,C) where S is the sensor layer with n sensor devices; G is a gateway device at the fog layer; C indicates a cloud device. Each sensor run

an application in a certain period of time. These applications are represented by a tuple $A_i = \{D_{in}, D_{out}\}$, where D_{in} is the amount of input data required for each application and D_{out} is the output data of each application. All the notations are described in Table 2.1.

2.2.1 Computation Offloading Model

As previously mentioned, sensor devices are supposed to execute application A_i in a certain period of time. Computation offloading model define whether the application should be uploaded to the upper computing resources or be performed at the local resource. The offload decision for the system is represented by a tuple $\alpha = \{\alpha^S, \alpha^G, \alpha^C\}$ where α^j represents offloading decision at layer j. For example, if the sensor layer executes the applications at layer $j \in \{S, G, C\}$ then, $\alpha^j = 1$, otherwise it equals to zero.

2.2.2 Response Time Model

In general, response time is the total time it takes to respond to a request for a service [86]. In IoT systems, the response time is defined as the time difference between the moment when enough data is collected for decision-making and the moment when the result is delivered to the consumer [86]. In general, the actuators can be located at different layers in IoT systems which leads to different dataflows from sensors to actuators. In this section, response time model is investigated for two common dataflows as follows:

Dataflow 1: The actuators (i.e., consumers) are located at the sensor layer, so the action response is issued at the processing unit and is performed at the sensor layer. Based on this scenario, the data analysis can be performed at the three layers: the sensor node processing unit, the gateway device processing unit, and the cloud processing unit. Assuming the sensor node is capable of performing local processing, the response time is obtained from the

following equation:

$$T_r^S = T_e^S \tag{2.1}$$

where T_e^S is the execution time of analyzing raw data at the sensor layer. In contrast, if the sensors offload the analyzing computation to the gateway layer, the total response time will include the data transmission time from the sensor layer to the gateway layer and the notification transmission time from the gateway to the actuators. Therefore, in the case of a two-layer sensor gateway, the total response time is as follows:

$$T_r^G = T_{tran}^{sg} + T_e^G + T_{tran}^{gs} \tag{2.2}$$

where T_{tran}^{sg} and T_{tran}^{gs} are the sensor-to-gateway and gateway-to-sensor transmission time, respectively. T_e^G is the execution time needed to analyze transmitted data on the gateway. If the computation needs to be performed with a more powerful processing unit, sensors will offload the computation to the cloud layer, making the total response time equal to the round trip transmission time from the sensor layer to the cloud layer. This time can be obtained as:

$$T_r^C = T_{tran}^{sg} + T_{tran}^{gc} + T_e^C + T_{tran}^{cg} + T_{tran}^{gs} \tag{2.3}$$

where T_{tran}^{sc} and T_{tran}^{cs} are the sensor-to-cloud and cloud-to-sensor transmission time, respectively, and T_e^C is the execution time at the cloud layer.

Dataflow 2: the actuators can be physically distant from the sensor nodes. A network of sensors can collect data to be analyzed through the layers and notify the end-user on the application layer. Tele-monitoring IoT systems usually follow this type of dataflow. In this regard, the result (e.g., notification) is transmitted from the cloud layer to an end user, assuming the end user is connected to the cloud layer. According to where the processing of

collected data is performed, the total response time can be obtained from one of the following equations. Assuming the sensor node performs the processing at its local processing unit, the total response time is the following:

$$T_r^S = T_e^S + T_{tran}^{sg} + T_{tran}^{gc} + T_{tran}^{ca} \tag{2.4}$$

where T_{tran}^{ca} is the cloud-to-end user transmission time. In contrast, if either the gateway or cloud performs the computations, the total response times can be obtained from the following equations, respectively:

$$T_r^G = T_{tran}^{sg} + T_e^G + T_{tran}^{gc} + T_{tran}^{ca} \tag{2.5}$$

and,

$$T_r^C = T_{tran}^{sg} + T_{tran}^{gc} + T_e^C + T_{tran}^{ca} \tag{2.6}$$

Therefore, the response time in both dataflows for request from the sensor layer with $\alpha = \{\alpha^S, \alpha^G, \alpha^C\}$ as offload decision tuple can be summarized into the following equation:

$$T_r = \alpha^S.T_r^S + \alpha^G.T_r^G + \alpha^C.T_r^C \tag{2.7}$$

where only one α^j equals to one while the others are zero. The lowest response time can be obtained by choosing the optimal computing scheme for IoT systems in different system conditions. The offload decision tuple determines the computing scheme for the system to obtain the lowest response time in the design time or dynamic solution.

2.3 Exploring Computation Offloading Space

A complex Internet of Things system needs to be systematically designed in the initial stages of the design to operate optimally between available design alternatives. This requires an exploration of all influential parameters on system behavior. System behavior depends on several factors that many studies have tended to neglect. In the previous section, we investigated the response time in three-layer IoT systems, and modeled the response time in different dataflow configurations. This section presents a design space exploration for computation offloading in IoT systems, showing impact the following parameters on the response time:

- **Communication:** offloading computations to upper layers in IoT systems requires transmitting data through the layers. Therefore, communication has a significant impact on the total response time. We consider the packet loss ratio (β) as a parameter which impacts communication cost between the devices [16].

- **System Complexity:** execution time might vary depending on what amount of workloads arrives at the computing resource or the number of requested services. Therefore, investigating the response time must include the effect of system complexity at design time or run-time.

- **Dataflow:** the locations of end-users can change the communication cost. Therefore, any change in dataflow from where the data is collected, processed, and delivered can directly impact system behavior. In this chapter, we investigate two common dataflows where in Dataflow 1, the actuators (i.e., consumers) are located at the sensor layer, and in Dataflow 2, actuators are physically distant from sensor nodes and located at the application layer.

- **Application Characteristic:** previous studies have neglected the impact of appli-

Network	Download Speed(Mbps)	Upload Speed(Mbps)
3G	2.0275	1.1
4G	13.76	5.85
Wi-Fi	54.97	18.88

Table 2.2: Average bandwidth for download and upload in different networks

cation characteristics in design space explorations when a proper computation layer needs to be determined. Response time not only depends on system characteristics (e.g., data flow) but also on application characteristics. The ratio of generated output data to input data can vary, which can affect the response time. One application characteristic is the OIDG ratio (γ). To show the impact of this application characteristic on system behavior, this study examines the following four manifestations of various OIDG regions: **(i)** an application from the region of extremely low OIDG ratios, such as machine-learning classifiers. A large volume of input data can be classified into a few classes ($\gamma << 1$). **(ii)** an application from the region of low OIDG ratios, such as compression or down-sampling ($\gamma < 1$). **(iii)** an application from the region of OIDG ratios almost equal to 1, such as noise filtering where the sample per second is unchanged ($\gamma \simeq 1$). **(iv)** an application from the region of extremely high OIDG ratios, such as encryption or video decoding ($\gamma > 1$).

In the following subsections, we conduct an experiment to investigate the dependency of IoT systems on the mentioned parameters.

2.3.1 Experimental Setup

We exploit Raspberry Pi Zero as the sensor device and NVIDIA Jetson TX2 as a gateway device. In addition, AWS c5.4xlarge node is used as a cloud layer. The gateway is equipped with integrated 256-core NVIDIA Pascal GPU and hex-core ARMv8 CPU, which

Application #	Application Name	OIDG Ratio	Input Size(Byte)	Output Size(Byte)	Description
1	YOLO	0.001	164000	161	Image object detector
2	Canny Edge Detector	0.043	14000000	615000	Video edge detector
3	Video-denoising	1	8200000	8200000	Non-local denoising application
4	Video-decoder	95.141	145536	13846510	Image object detector

Table 2.3: Applications' Specifications

	Raspberry Pi Zero	NVIDIA Jetson TX2	AWS c5.4xLarge
Processor	Single Core	Quad-core Cortex-A57	16 Core Intel Xeon 8000
Architecture	ARM	ARM	X86
Speed	1 GHz	2.0 GHz	3.2 GHz
GPU	—	256-core Pascal	—
RAM	512 MB	8GB	32GB
External Storage	16GB eMMC	32GB eMMC	512GB SSD

Table 2.4: Platforms specifications

is substantially more powerful than the sensor node. However, the cloud layer has more powerful computing units, such as a 16-Core Intel Xeon Platinum 8000 processor. Table 2.4 contains the specifications of all devices. The sensor and gateway use MQTT protocol to communicate with each other [49]. The cloud node, in contrast, runs Gunicorn with Flask, which is a micro web framework written in Python. The gateway is also connected to the AWS instance via cellular network or Wi-Fi. Table 2.2 lists the average bandwidth for each connectivity protocol [1].

2.3.2 Evaluation

We conduct experiments to indicate the system with applications selected in different OIDG regions. The described system is investigated through a variety of environmental conditions, application characteristics, dataflow configurations, and system complexity as follows. Four applications with different characteristics are chosen as explained in Table 2.3. The appli-

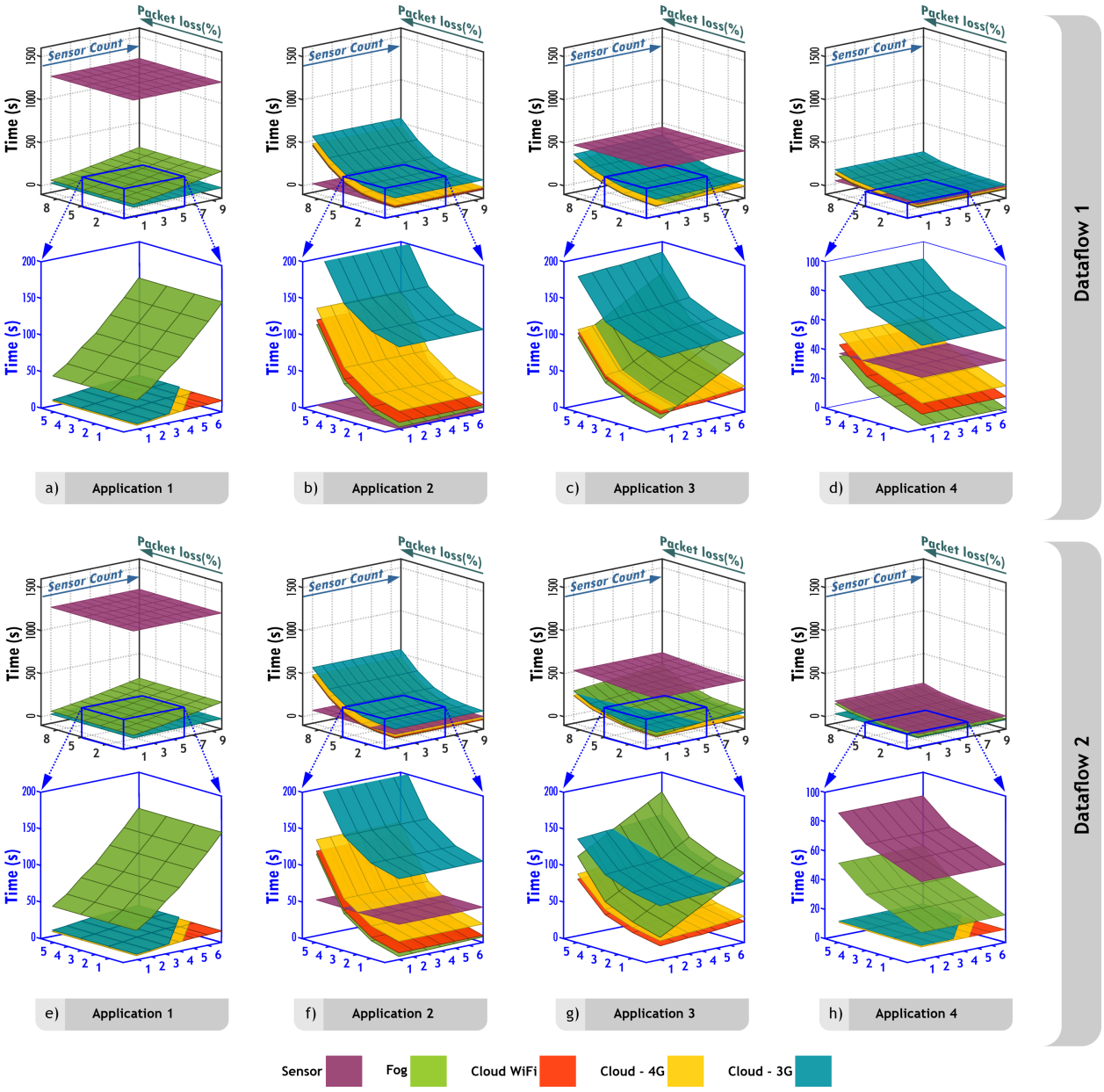


Figure 2.2: The Computation Offloading Exploration for different environmental parameters and system complexity for chosen applications in two different dataflows.

cation's executable file is located at all the computing computing nodes. Therefore, once the offloading decision determines the computing scheme, the corresponding node executes the application from its local memory. System complexity varies from a single node to ten nodes in the same network. Dataflows are selected from the configurations, as explained in the System Model section. Communication over a Wi-Fi network is affected by packet loss

ratio, which varies within a certain range of 0% to 14% in our investigation. The packet loss ratio is targeted within the range as any ratio more 14% would dramatically increase the communication delay and response time consequently. In addition, communication between the fog layer and the cloud layer is based on 3G, 4G, or Wi-Fi connectivity.

Figure 2.2 shows the result of design space exploration considering the mentioned parameters. In the figure, each plot shows an exploration for a specific application type and dataflow configuration. Besides, each plane represents a specific computation offloading policy. For example, the green planes represent the response time (Z-axis) when all sensors offload workloads to the fog node while the yellow planes represent the cloud computing with 4G connectivity to the fog layer. In each plot, X-axis and Y-axis represents the packet loss ratio and number of connected sensors respectively.

The total response time is significantly affected by transmission time across the layers. The system running Application 2 on Dataflow 2 (see Figure 2.f) is remarkably affected by variation in packet loss ratio. In the low packet loss ratio, fog computing is the optimal solution among other schemes as the green plane is below the others. However, in high packet loss ratio sensor layer computing is the optimal solution. In this case, an increase in packet loss ratio leads to a significant increase in response time showing the advantage of sensor layer computing in some conditions. On the other hand, the packet loss ratio does not affect the optimal solution in the system running Application 1 with both dataflow configurations (see Figure 2.a and 2.e).

Furthermore, simultaneous requests to a computation unit can severely interrupt request processing, which affects response time. Therefore, in complex systems, as the number of connected sensors increases, there is a possibility of significant change in system behavior. As our experiments show in Figure 2.2, the system running Application 3 on Dataflow 1 (see Figure 2.c) is remarkably affected by increasing the number of connected sensors. The fog computing scheme is the optimal solution when number of connected sensor equals to

1. However, it becomes the cloud computing scheme for the system with more connected sensors. In the case of simultaneous requests, the fog layer is more sensitive than the cloud layer, which leads to changing the system behavior in a more complex system.

In addition, the consumer's location affects the response time in our experiment. The system running Application 4 has different behavior in both dataflows. In this case, cloud-3G computing is the worst solution for Dataflow 1 (see Figure 2.d), while it can be the optimal solution with Dataflow 2 (see Figure 2.h), implying that, for the same environmental parameters and system complexity, a different flow of data can change the optimal solution. In another case, the system running Application 2 with Dataflow 1 (see Figure 2.e) performs computation optimally in the sensor layer. However, that might not be always true for Dataflow 2.

The OIDG ratio (γ) as the application characteristic remarkably affects system behavior. In other words, a system with a higher ratio has more communication costs delivering the outputs to the consumer. For example, running Application 2 with Dataflow 1 (see Figure 2.b) over the sensor layer is more suitable while running Application 4 with Dataflow 1 (see Figure 2.d) on the upper layers is faster. In addition, the system behavior is more sensitive to packet loss when OIDG is high. Consequently, various parameters affect IoT systems, and they need to be considered to design the model.

2.4 A Proof-of-Concept Dynamic Computation Offloading Technique

In the previous section, we examined the response times of IoT systems, showing how communication and computational limitations could influence different applications. Our experiment demonstrated the behavior of IoT systems leveraging different computational resources

Algorithm 1 The dynamic computation offloading decision-making in the *Decide* component

Initialization in design time:
$\vec{\alpha} \leftarrow \{1, 0, 0\}$
▷ $\vec{\alpha}$ represents the decision offloading tuple for all sensors:
{1,0,0} (in sensor), {0,1,0} (in gateway) and {0,0,1} (in cloud)
while system is on **do**
  $Mode \leftarrow$ Dataflow mode ▷ i.e., 1 or 2 selected by the user
  if $Mode = 1$ **then**
    From the *Observe*:
      Collect T_e^S, T_e^G, T_e^C, T_{tran}^{sg}, T_{tran}^{gc}, T_{tran}^{cg}, and T_{tran}^{gs}
  else if $Mode = 2$ **then**
    From the *Observe*:
      Collect T_e^S, T_e^G, T_e^C, T_{tran}^{sg}, T_{tran}^{gc}, T_{tran}^{ca}
  end if
  Calculate T_r^S, T_r^G, and T_r^C
  Update $\vec{\alpha}$ based on $\{\min(T_r^S, T_r^G, T_r^C)$ - $\Delta\}$
  To the *Act*:
    Send $\vec{\alpha}$ for the actuation
end while

in three-layer architecture. Such an investigation is necessary for system evaluation in design time and the provision of optimal computation schemes for stationary conditions; however, it is insufficient in the dynamic runtime environment.

The behavior of IoT-based systems dramatically changes during runtime due to several variations in status and context. For example, degraded Internet connection and uncertainty in network latency have an impact on transmission latency. Moreover, the computation time of an application varies considerably due to varying computation loads on different system nodes (e.g., gateway devices).

2.4.1 System Design

In this sub-section, we present and examine a simple proof-of-concept end-to-end dynamic computation offloading technique tailored for IoT-based systems. The proposed approach selects the near-minimum computation offloading scheme according to the system's condi-

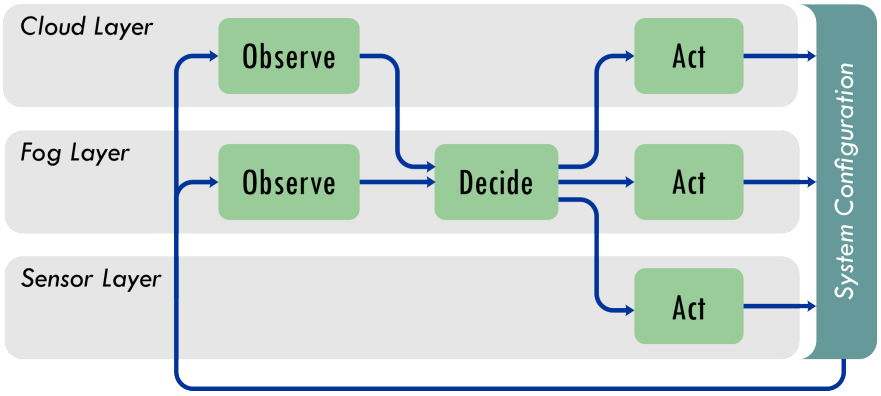


Figure 2.3: The ODA control loop in the three-layer IoT system [8].

tion. As mentioned earlier, there are two dataflow modes in these systems, since the end-user can be considered in the vicinity of the sensor layer or in the cloud layer (see Figure 2.1). Therefore, in each iteration, the proposed method is performed to minimize the latency of mode 1 or mode 2. The proposed dynamic computation migration approach employs a closed-loop control strategy to minimize response latency. We exploit the Observe, Decide, and Act (ODA) control loop in this regard to leverage the system's conditions into the computation migration decision-making [8]. A conventional ODA paradigm includes three major components. The *Observe* component captures the system status, enabling self- and context-awareness in the system. The *Decide* component performs the decision-making according to the measurements obtained in the Observe component. Finally, the *Act* component fulfills the decision, making the (required) changes in the system accordingly [35].

The ODA paradigm, as the backbone of the proposed computation migration approach, is positioned in the three-layer IoT systems, as shown in Figure 2.3. The *Observe* component is distributed into the fog and cloud layers, as the transmission and computation loads are dynamic and should be collected continuously. Note that computation time at all nodes is measured in the design time to be used for the run-time decision making. The *Decide* component is fully positioned at the fog layer, enabling local computation migration decision-making. The actuation of this approach is carried out in all three layers, since the computation can be performed in the sensor network, gateway devices, or cloud server.

Therefore, the *Act* component is partitioned into the three layers.

The computation offloading decision-making, performed in the *Decide* component, is indicated in Algorithm 1, which indicates how the computation is dynamically switched between a sensor node, a gateway device, and a cloud server in our system. In design time, $\vec{\alpha}$ is initialized as $\{1, 0, 0\}$ for all sensors, and the computation time of the sensor, T_e^S, is measured. When the system is on (i.e., runtime), the algorithm first checks the dataflow mode selected by the user. As mentioned earlier, the response time of the system is different in the two modes. According to the selected mode, the required values are collected from the *Observe* component. Then, the response time of the three layers, T_r^S, T_r^G, and T_r^C, are calculated via Equations (1)/(4), (2)/(5), and (3)/(6). By obtaining the minimum response time, $\vec{\alpha}$ is updated and sent to the *Act* component for actuation in the next iteration. The possible permutations for $\vec{\alpha}$ are limited to $\{1, 0, 0\}, \{0, 1, 0\}, \{0, 0, 1\}$. Therefore, the computational complexity for the decision making is $O(1)$. It is worth mentioning that the system might oscillate between two layers if the response time values are proximal. To avoid such unnecessary oscillations, we consider a simple threshold strategy by adding a soft margin (i.e., Δ) in the computation of the minimum layer's response time. The soft margin is defined based on the application sensitivity and the quality of the communication. In this work, we selected 10% of the current response time as the soft margin. Therefore, α is changed if the response time in the layer with computation subtracted by Δ is greater than the other layers' response times (line 15 in Algorithm 1). The runtime dynamic computation offloading system leverages the ODA loop to minimize the response time during the runtime. The system dynamically assigns the computation to the layers by observing environmental parameters, system complexity, etc.

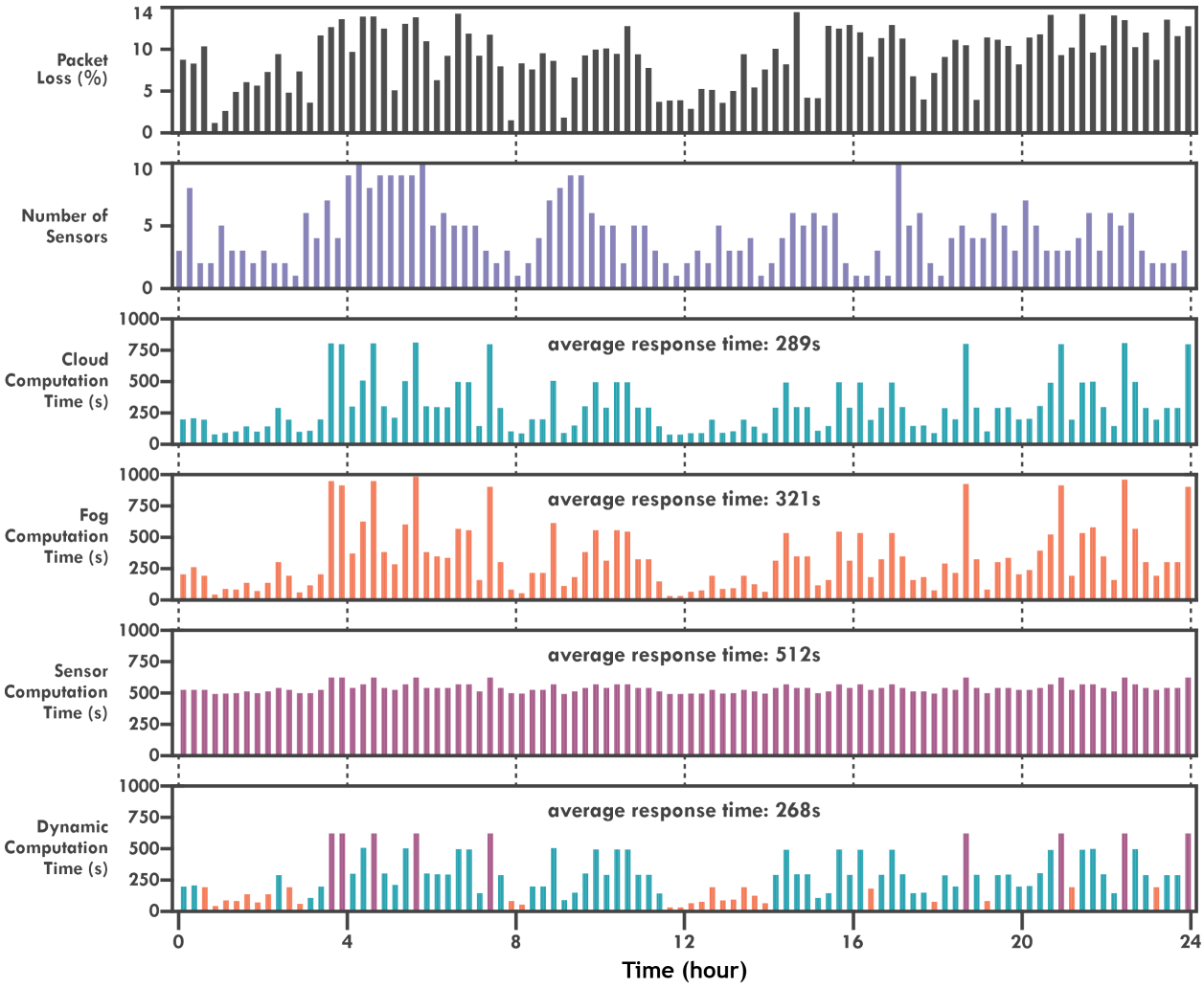


Figure 2.4: Dynamic system behavior over a scenario

2.4.2 Evaluation

This subsection evaluates the dynamic system under various uncertain complexity and environmental conditions. In the envisaged scenario that executes over 24 hours, packet loss varies between 0% and 14% and the number of sensors connected to our network changes between 1 and 10. According to the Exploring Computation Offloading Space section, the system which executes Application 3 in Dataflow 2 behaves more sensitive to the mentioned influential parameters (see Figure 2.g). Therefore, in our evaluation, the system is supposed to execute Application 3 during the experiment within Dataflow 2. The dynamic system

assigns the computation to one of the layers using Algorithm 1, as shown in Figure 2.4. On the other hand, other computing schemes are static over the run-time where variations in the system parameters such as packet-loss or systems complexity do not change the computing scheme. The response time is evaluated with both the static and dynamic approaches. In the scenario, the dynamic solution begins with performing computation at the cloud layer. However, once the packet loss and the number of connected sensors changes, the dynamic system assigns the computation to the fog layer. On the other hand, the static approaches do not consider the condition variations where it leads to run the IoT system inefficient.

As a result, the system executes the application at the sensor, fog, and cloud layers with 289s, 321s, and 512s average response time respectively and through dynamic solution with 268s average response time, which demonstrates that better performance can be achieved through the dynamic solution. Figure 4 shows that the sensor layer computing is sometimes better than the others, due to the packet loss and system complexity. On the other hand, cloud layer computing is the optimal computing scheme where communication cost is low. These findings verify that static computing scheme would perform inefficient in compare with dynamic solution. Therefore, many parameters affect the IoT system, and they need to be considered. The proof-of-concept 24 hours test highlights the importance of considering the mentioned parameters for dynamic systems. It shows the design time models are insufficient for IoT based systems with dynamic behavior as the context and condition change. Self-aware optimization methods are required at run time, considering such dynamic and unpredictable conditions of IoT systems.

2.5 Related Work

This section illustrates related works in the computation offloading technique in IoT systems. Computation offloading is a technique to transfer compute-intensive tasks or applications

from the resource constrained IoT devices to more computing capable nodes to enhance QoS specially for IoT latency-sensitive applications. In this section we explain efforts to address the problem. Even though there have been recently some efforts to co-optimize communication and computation in a coupled manner, the literature lacks a comprehensive solution that highlight the impact of various system parameters (e.g., application characteristic [OIDG] and dataflow configuration).

Zhang et al. [139] design an energy-efficient computation offloading algorithm. The work formulates an optimization problem to minimize energy consumption of the offloading system by considering computation task offloading and its data transmission in the offloading decision. The optimization problem is reduced with a simplification method to polynomial complexity and solved in numerical method. Cao et al. [15] investigate a joint computation and communication user cooperation for Mobile Edge Computing (MEC) where a nearby helper node is enabled to share its resource to improve QoS. The work proposes a energy-efficient framework to minimize the total energy consumption at both the user and the helper using convex optimization. You et al. [133] propose a design for mobile cooperative computing that enables a user to exploit non-causal CPU-state information shared by offloading computation to a nearby helper. It helps to fully utilize random computation resources at the helper with minimum energy consumption.

Sheng et al. [112] present a joint optimization problem of computation and communication in the MEC system to optimally partition, offload and execute tasks between sensor nodes and edge nodes to minimize energy consumption. Kao et al. [46] formulate a task assignment in multiple resource constrained mobile devices and provide a fully polynomial time approximation algorithm to find the optimal offloading scheme that balances latency and energy consumption of mobile devices. An efficient one-dimensional search algorithm is proposed [63] to find optimal task scheduling policy for MEC by formulating a power-constrained delay minimization problem.

Skarlat et al. [113] model the service placement problem for IoT applications over fog resources as an optimization problem and propose a genetic algorithm to solve the problem. Latency-minimization problem in a multi-user time-division multiple access for mobile-edge computation offloading is investigated by [102]. The work formulates a convex optimization problem for partial compression problem and develops an optimal joint communication and computation resource allocation algorithm to solve it. Kouloumpris et al. [57, 56] uses a mathematical programming based framework to optimally allocate tasks while satisfying operational constraints.

Kwak et al. [58] jointly optimize dynamic cloud offloading policy and CPU scaling in dynamic mobile network environment. The work uses the Lyapunov optimization technique to minimize energy consumption for given delay constraints. Partial computation offloading modeling in MEC is presented to minimize energy consumption and latency by jointly optimizing the computational speed (DVFS) of mobile devices and communication power consumption and offloading ratio [125]. The work formulates the problem and offers the globally optimal solutions in closed-form which shows that total offloading could not be optimal when the smart mobile device has the capability of computation scaling. A message-passing framework is proposed to reduce complexity of partial computation offloading by formulating the problem as a mixed integer program [50]. An online algorithm for partial computation offloading is proposed based on Lyapunov optimization to minimize energy and latency. The agent decides the CPU-cycle frequencies for local execution and the bandwidth allocation and transmit power for computation offloading [73].

Chen et al. [19] aim to find the overall optimal decision on partial computation offloading in a general mobile cloud computing system to minimize a weighted total cost of energy, computation, and delay of all users. The work formulates the problem as a non-convex quadratically constrained quadratic program and proposes an efficient algorithm to solve the problem. Nan et al. [87] present an adaptive online decision-making algorithm which uses

the Lyapunov optimization technique to distribute the incoming data to the corresponding tiers to minimize response time and application loss number. Liu et al. [65] formulate a multi objective optimization problem to minimize the energy consumption and delay performance in mobile devices. The work optimizes the offloading probability and transmission power by using an Interior Point Method based algorithm.

Zhao et al. [141] provide an offloading algorithm to minimize energy consumption with consideration of latency tolerance and the transmission power of mobile devices. The work solves the optimization problem using Non-linear fractional programming dynamically and makes a decision to offload computing by adjusting transmission power. Chamola et al. [17] propose an optimal task assignment in a network of connected cloudlets which provide services to mobile devices to minimize the latency. Chang et al. [18] investigate the problem of energy optimization for a fog computing system. The work proposes a scheme that optimizes both offloading probability and transmit power to minimize the energy consumption by using nonlinear optimization and ADMM-based method.

The majority of existing solutions only focus on communication-computation optimization considering limited system parameters such as processing capacity and network bandwidth. In contrast, we highlight the impact of variety of parameters (e.g., application characteristics, system complexity, communication cost, and dataflow configuration) on IoT-based systems (see Figure 2). Furthermore, finding optimal computation offloading policy for dynamic systems is critical since dynamicity of environment changes over time. In this chapter, we presented a proof-of-concept dynamic computation offloading strategy to show IoT-based systems are required to be dynamically controlled at run-time.

2.6 Summary

This chapter investigated various co-design aspects of computation offloading for IoT systems with typical three-layer architecture. We explored the impact of dataflows, communication, system complexity, and application characteristics on the response time in IoT systems. Then, we examined an IoT-based system enabled by an end-to-end dynamic computation offloading. The system minimized the response time of the application by 1) observing the system condition, 2) performing computation migration decision-making, and 3) carrying out the computation in the sensor, fog, or cloud layer. Experiments showed the optimal solution varies based on the network latency, dataflow configuration, application characteristic, and the number of connected sensors to the fog layer. We showed it in a scenario with run-time variations in system complexity and packet loss ratio. The dynamic computation offloading system reduced the average response time with $8\%, 17\%, 48\%$ in comparison with only the cloud or fog or sensor computing scheme respectively.

Chapter 3

Online Learning for Deep Learning Orchestration

3.1 Introduction

Deep-learning (DL) is advancing real-time and interactive user services in domains such as autonomous vehicles, natural language processing, healthcare, and smart cities [104]. Due to user device resource constraints, deep learning kernels are often deployed on cloud infrastructure to meet computational demands [11]. However, unpredictable network constraints including signal strength and delays affect real-time cloud services [51]. Edge computing has emerged to complement cloud services, bringing compute capacity closer to the user-end devices [134]. A collaborative end-edge-cloud architecture is essential to provide deep-learning-based services with acceptable latency to user-end devices [79]. The edge paradigm increases offloading opportunities for resource-constrained user-end devices. Offloading DL services in a 3-tier end-edge-cloud architecture is a complex optimization problem considering: (i) diversity in system parameters including heterogeneous computing resources, network

constraints, and application characteristics, and (ii) dynamicity of DL service environment including workload arrival rate, user traffic, and multi-dimensional performance requirements (e.g., application accuracy, response time) [27, 109, 107].

Existing offloading strategies for DL tasks are based on the assumptions that (i) all DL tasks have similar compute intensity and require similar communication bandwidth, (ii) offloading improves performance, and (iii) latency is guaranteed with offloaded tasks. However, these assumptions do not hold in practice due to dynamically varying application and network characteristics, where the computation-communication and accuracy-performance tradeoffs are inconsistent and nontrivial [27, 109, 119]. Under varying system dynamics, such offloading strategies limit the gains from using the edge and cloud resources. Further, model optimization techniques such as quantization and pruning can reduce the computation complexity of DL tasks by sacrificing the model accuracy [24, 117]. Considering model optimization techniques in conjunction with offloading provides opportunities to influence the computation-communication trade-off [118]. This exposes an alternative to offloading in resource constrained devices executing DL inference. Finding the optimal choice between offloading the DL tasks to edge and cloud layers and using optimized models for inference at local devices results in a high-dimensional optimization problem.

Understanding the underlying system dynamics and intricacies among computation, communication, accuracy, and latency is necessary to orchestrate the DL services on multi-level edge architectures. Reinforcement learning is an effective approach to develop such an understanding and interpret the varying dynamics of such systems [78, 92]. Reinforcement learning allows a system to identify complex dynamics between influential system parameters and make a decision online to optimize objectives such as response time [116]. We propose to employ online reinforcement learning to orchestrate DL services for multi-users over the end-edge-cloud system. Our contributions are:

- Runtime orchestration scheme for DL inference services on multi-user end-edge-cloud networks. The orchestrator uses reinforcement learning to perform platform offloading and DL model selection at runtime to optimize response time provided accuracy requirements.
- Implementation of our online learning solution on a real end-edge-cloud test-bed and demonstration of its effectiveness in comparison with state-of-the-art [105] edge orchestration strategies.

3.2 Background

In this section, we present the relevant background and significance of orchestrating DL workloads on end-edge-cloud architecture.

3.2.1 Offloading DL Workloads in End-Edge-Cloud Architecture

Computation offloading techniques offload an application (or a task within an application) to an external device such as cloud servers [72]. Offloading is typically done in order to improve performance or efficiency of devices [11]. DL workloads on end-devices are conventionally offloaded to cloud servers, but delay-sensitive services for distributed systems rely on performing inference at the edge as an alternative [134]. Inference at the edge can provide cloud-like compute capability closer to the user devices, reducing data transmission and network traffic load. Edge offloading can provide relatively predictable and reliable performance compared to cloud offloading, as there is less workload and network variance [51] [45]. In the context of the end-edge-cloud paradigm, computation offloading techniques partition workloads and distribute tasks among multiple layers (local device, edge device, cloud servers) such that the performance and efficiency objectives are met.

The collaborative end-edge-cloud architecture provides execution choices such that each workload can be executed on the device, on the edge, on the cloud, or a combination of these layers. Each execution choice effects the performance and energy consumption of the user end device, based on the system parameters such as hardware capabilities, network conditions, and workload characteristics. A distributed end-edge-cloud system consists of the following layers:

- **application layer**: provides user level access to a set of services to be delivered by computing nodes
- **platform layer**: provides a set of capabilities to connect, monitor and control end/edge/cloud nodes in a network
- **network layer**: provides connectivity for data and control transfer between different physical devices across multiple
- **hardware layer**: provides hardware capabilities for computing nodes in the system

Each layer presents a diverse set of requirements, constraints, and opportunities to tradeoff performance and efficiency that vary over time. For example, the application layer focuses on the user's perception of algorithmic correctness of services, while the platform layer focuses on improving system parameters such as energy drain and data volume migrated across nodes. Both application and platform layers have different measurable metrics and controllable parameters to expose different opportunities that can be exploited for meeting overall objectives. In the case of DL inference, different DL model structures present opportunities in the application layer, and different computation offloading decisions in a collaborative end-edge-cloud system present opportunities in the platform layer, both for optimizing the execution while meeting required model accuracy.

3.2.2 Intelligence for Orchestration

Runtime system dynamics affect orchestration strategies significantly in addition to requirements and opportunities. Sources of runtime variation across the system stack include workload of a specific computing node, connectivity and signal strength of the network, mobility and interaction of a given user, etc. Considering cross-layer requirements, opportunities, and runtime variations provide necessary feedback to make appropriate choices on system configurations such offloading policies. Identifying optimal orchestration considering the cross-layer opportunities and requirements in the face of varying system dynamics is a challenging problem. Making the optimal orchestration choice considering these varying dynamics is an NP-hard problem, while brute force search of a large configuration space is impractical for real-time applications. Understanding the requirements at each level of the system stack and translating them into measurable metrics enables appropriate orchestration decision making. Heuristic, rule-based, and closed-loop feedback control solutions are not efficient until reaching convergence, which requires long periods of time [116]. To address these limitations, reinforcement learning approaches have been adapted for the computation offloading problem [105]. Reinforcement learning builds specific models based on data collected over initial epochs, and dramatically improves the prediction accuracy [116].

3.3 Motivation

This section presents a comprehensive investigation of DL inference for multi-users in end-edge-cloud systems. We examine the scenario using a real setup including five AWS a1.medium instances with single ARM-core as end-node devices connected to an AWS a1.large instance as edge device and an AWS a1.xlarge instance as cloud node. We conduct experiments for DL inferences with the MobileNetV1 model while varying (i) network connection, (ii) number of active users, and (iii) accuracy requirement. We consider three possible execution

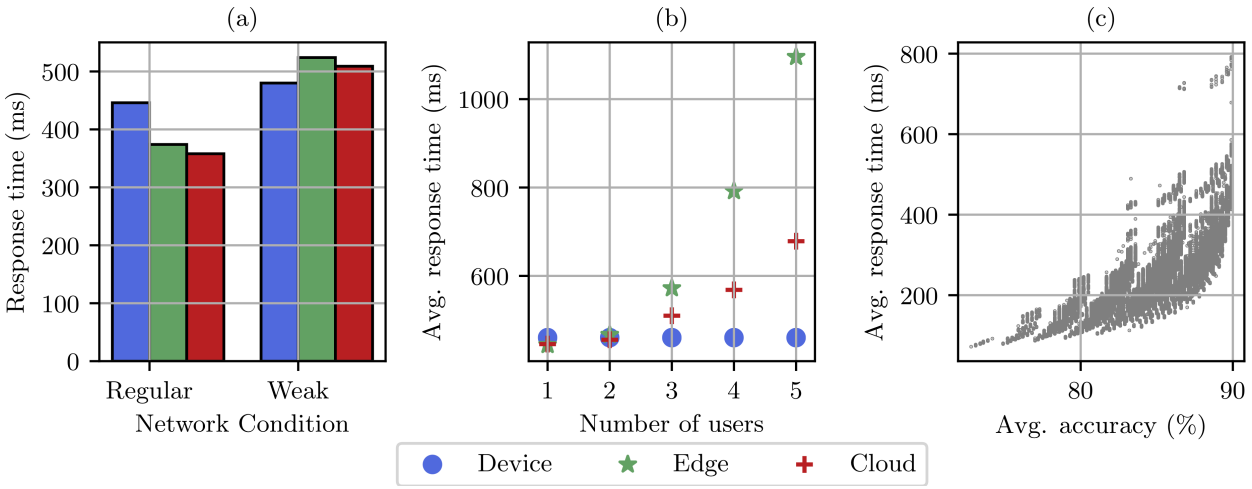


Figure 3.1: Impact of varying system and application dynamics on performance for MobileNet application. (a) Response time on user-end device, edge and cloud layers with regular and weak network conditions. (b) Average response time with varying number of active users for different computing schemes. (c) Average response time achieved with varying levels of average accuracy.

choices: (i) on device, (ii) on edge, and (iii) on cloud. The device, edge, and cloud execution choices represent executing the inference completely on the local device, on the edge, and on the cloud respectively. The detailed specifications for the end-edge-cloud setup appear in Section 3.5.

3.3.1 Impact of System Dynamics on Inference Performance

Network We consider two possible levels of network connections: (i) a low-latency (regular) network that has the signal strength for better connectivity, and (ii) a high-latency (weak) network that has a weaker signal with poor connectivity. Figure 3.1 (a) shows the response time of MobileNet application on user device, edge, and cloud layers with regular and weak networks. With a regular network, the response time is highest for executing the application on the user end device. The response time decreases as the computation is offloaded to edge and cloud layers, with the higher computational resources. With a weak network, the response time of the edge and cloud layers is higher, as the poor signal strength

adds delay. The response time of the edge node in this case is higher than the cloud layer, given the lower compute capacity of the edge node. Performance of the user end device is independent of the network connection, resulting in lowest response time. This demonstrates the spectrum of response times achievable with compute nodes at different layers, under varying network constraints. For example, the best execution choice with a regular network is the cloud layer, whereas it is the local execution with a weak network.

Users We examine user variability by considering multiple simultaneously active users ranging from 1 to 5. Figure 3.1 (b) shows the average response time with varying number of users. The average response time remains constant when running the application on a user end device, i.e., each user executes the application on their local device. When offloaded to the edge layer, the average response time increases significantly as the number of users increase. This is attributed to the increased network load with multiple simultaneously active users as well as limited resources at the edge layer to handle several user requests concurrently. The average response time also increases when offloaded to the cloud layer as the number of simultaneous users increases. However, the response time is lower when compared to the edge layer, since the cloud layer has a larger volume of resources to handle multiple simultaneous user requests.

Accuracy We demonstrate the impact of varying DL models on performance under different system dynamics. We select between eight models with Top-5 accuracy between 72.8% and 89.9%, while also considering all three layers for execution, and between 1 and 5 simultaneously active users. Figure 3.1 (c) shows the average response time achieved with varying levels of average accuracy over a multi-dimensional space of different execution choice and different number of users. Each point in Figure 3.1 (c) represents a unique case of an execution choice (among device, edge, and cloud), number of active users (among 1 to 5), and accuracy level. We present the average response time achieved with different levels of

Table 3.1: Reinforcement Learning Based Works. *CO* represents the computation offloading technique. *HW* and *APP* represents knobs belong to the hardware and application layer, respectively.

Related Works	**Real System Evaluation**	**Multi-User**	**End-to-End**	**Actions**
[97, 23, 21, 76, 129]	✗	✗	✗	CO
[53]	✓	✗	✗	CO,HW
[48, 59, 70, 126, 2, 22, 105]	✗	✓	✗	CO
Ours	✓	✓	✓	CO,APP

accuracy. As expected, the response time increases with increase in model accuracy. However, we observe tradeoffs among different response times between accuracy and number of active users. For instance, it is possible to support multiple users within the response time of servicing a single user, by lowering the model accuracy.

Considering the three major sources of variations in number of users, network conditions, and model accuracy, finding an optimal choice of execution for end-edge-cloud architectures at runtime is challenging. As such architectures scale in the number of users and edge nodes, the accuracy-performance Pareto-space becomes increasingly cumbersome for finding an optimal configuration among the fine-grained choices. Brute force and smart search algorithms do not offer practically feasible solutions to orchestrate applications in real-time. While machine learning algorithms can identify near-optimal configuration choices, they require exhaustive training, considering continuously varying system dynamics. We propose to employ online reinforcement learning to understand the volatility of system dynamics and make near-optimal orchestration decisions in real-time to improve the response time of DL inferencing on end-edge-cloud architectures.

3.3.2 Related Work

We categorize research related to optimally deploying DL services at the edge in two ways: (i) work related to deploying DL inference tasks over the end-edge-cloud collaborative architecture, and (ii) work related to adopting reinforcement learning methods to optimally offload tasks.

DL Inference in End-edge-cloud Networks Prior works propose frameworks to decompose DL inference into tasks and perform distributed computations. In these works, a DL model can be partitioned vertically or horizontally along the end-edge-cloud architecture. Generally, DL models are partitioned according to the compute cost of model layers and required bandwidth for each layer to be distributed among the end-edge-cloud [100, 130, 41, 45]. These works find the optimal partition points based on traditional optimization techniques and offer design-time optimal solutions. Some efforts try to reduce the computation overhead of DL tasks through various model optimization methods such as quantization. These methods transform or re-design models to fit them into resource-constrained edge devices with little loss in accuracy [32, 26, 75]. AdaDeep [66] proposes a Deep Reinforcement Learning method to optimally select from a pool of compressed models according to available resources. However, AdaDeep relies only on the model selection technique while our work combines computation offloading and model selection techniques to achieve the optimal response time.

Learning-based Offloading Prior works address the offloading problem to optimize different objectives including latency and energy consumption. Most of the works formulate the offloading problem with limited number of influential parameters and adopt online learning techniques with numerical evaluation [2, 129, 22, 97, 126, 48, 23, 21, 76, 59]. Lu et. al. [70] propose a Deep Recurrent Q-Learning algorithm based on Long Short Term Memory

network to minimize the latency for multi-service nodes in large-scale heterogeneous MEC and multi-dependence in mobile tasks. The algorithm is evaluated in iFogSim simulator with Google Cluster Trace. [105] proposes a Q-Learning based algorithm to minimize energy by considering various parameters in task characteristics and resource availability. Young Geun et al. [53] propose a reinforcement learning based offloading technique for energy efficient deep learning inference in the edge-cloud architecture. The work focuses on the learning for heterogeneous systems and lacks a comprehensive solution for multi-users end-edge-cloud systems. Haung et al. [37] uses a supervised learning algorithm for complicated radio situations and communication analysis and prediction to make an optimal actions to obtain a high quality of service (QoS). However, our proposed work employs model-free reinforcement learning algorithm to find the optimal orchestration scheme. There have been other efforts which apply game theory algorithms to address the orchestration problem in the network. Apostolopoulos et al. [4] propose a decentralized risk-aware data offloading framework using non-cooperative game formulation. The work uses a model-based game theory algorithm to find the optimal offloading decision which makes difference with our proposed model-free reinforcement learning approach. Model-based algorithm use a predictive internal model of the system to seek outcomes while avoiding the consequence of trial-and-error in real-time. The approach is sensitive to model bias and suffers from model errors between the predicted and actual behavior leading to sub-optimal orchestration decisions. Our Model-free RL technique operates with no assumptions about the system's dynamic or consequences of actions required to learn a policy.

Some works have been applied traditional optimization techniques to optimize the computation offloading problem [20]. Yuan et al. [136] model a profit-maximized collaborative computation offloading and resource allocation algorithm to maximize the profit of systems and meet the required response time. In another work, Bi et al. [14] propose a partial computation offloading method to minimize the total energy consumed by mobile devices. The work formulates the problem and optimizes using a novel hybrid meta-heuristic algorithm.

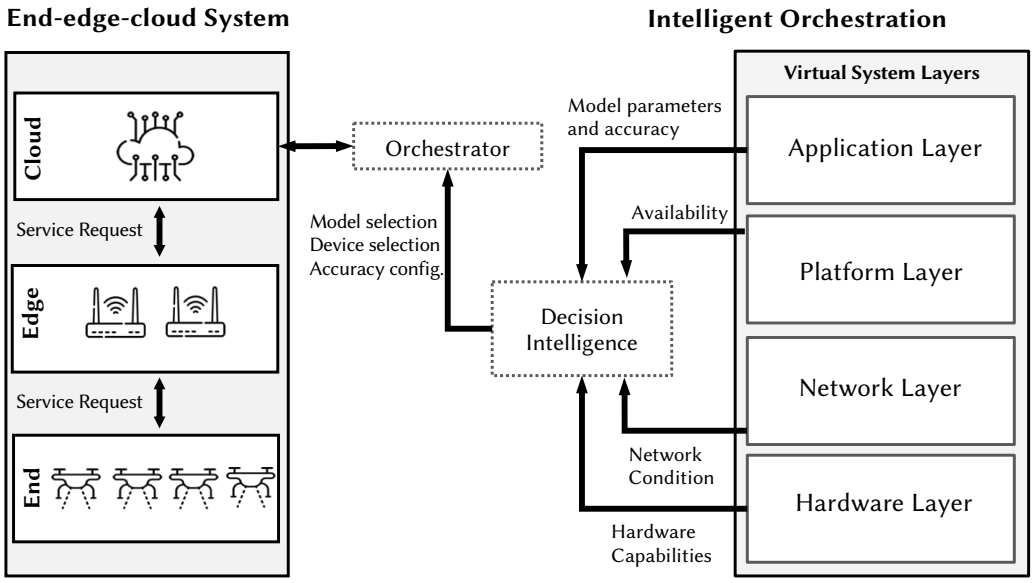


Figure 3.2: Intelligent orchestration of DL inference in end-edge-cloud architectures.

Considering systems are unknown with dynamic behavior, the traditional optimization techniques are not applicable for runtime decision-making. Table 3.1 positions our work with respect to state-of-the-art solutions. Our solution uses RL to optimally orchestrate DL inference in multi-user networks considering offloading and DL model selection techniques combined together.

3.3.3 Contributions

The ideal DL inference deployment provides maximum inference accuracy and minimum response time. Figure 3.2 shows an abstract overview of our target multi-layered architecture for online computation offloading of DL services. We consider three layers viz., user-end device, edge and cloud. Further, we classify this architecture into virtual system layers that include application, platform, network and hardware layers. Each of the virtual system layers provide sensory inputs for monitoring system and application dynamics such as DL model parameters, accuracy requirements, availability of devices for execution, network characteristics, and hardware capabilities. The *Decision Intelligence* component in Figure

3.2 periodically monitors resource availability from all virtual system layers to determine appropriate execution choice and DL models to achieve the required QoS (e.g, accuracy, response time). *Decision Intelligence* analyzes the system parameters to make orchestration decisions in terms of model selection, accuracy configuration, and offloading choices. The orchestrator is a software component that is hosted at the cloud layer and enforces the orchestration decisions upon receiving a service request from the user-end devices.

Finding an optimal computation policy including offloading and model selection to optimize objectives (e.g., accuracy, response time) is considered an NP-hard problem. The problem generally can be solved using traditional optimization techniques such as heuristic-based methods, meta-heuristic methods, or exact solutions. Considering systems are unknown with dynamic behavior, the traditional optimization techniques are not applicable for runtime decision-making to optimize objectives. Modeling an unexplored high-dimensional system is feasible using model-free reinforcement learning techniques [116]. Model-free RL operates with no assumptions about the system's dynamic or consequences of actions required to learn a policy. Model-free RL builds the policy model based on data collected through trial-and-error learning over epochs [116]. In this work, we use model-free reinforcement learning to deploy DL inference at the edge by considering offloading and model selection. Some works have been proposed to address the computation offloading problem using online techniques [2, 129, 22, 97, 126, 48, 23, 21, 76, 53]. However, there is no relevant work to investigate the integration of online learning with DL inference deployment. Therefore, the literature suffers from some shortcomings that are summarized as follows:

- ***Cross-layer Optimization*:** online solutions have not previously coordinated offloading and model optimizations together. As Table 3.1 shows, all related work relies on only computation offloading (CO). To the best of our knowledge, for the first time, this work considers both computation offloading and application-level adjustment (APP) together in order to achieve required QoS.

Table 3.2: Notation descriptions

Notation	Description
S	end-node device
E	edge device
C	cloud device
P	processor utilization
M	memory utilization
B	network condition
o	offloading decision
o_i^j	offloading decision for end-node i to resource j
N	number of end-node devices
d_k	DL model k
l	number of available DL models
T_{res}^j	response time for offloading DL task to resource j
α	learning rate
γ	discount factor

- ***Real System Evaluation*:** most RL-based solutions in the literature are numerically evaluated. Some works have been proposed and evaluated with simulators. As Table 3.1 shows, the literature lacks a real hardware implementation for online learning framework. This work implements the online system on real hardware devices which leads to realistic evaluation of online agent's overhead.

- ***End-to-End Solution*:** end-to-end solution considers a service from the moment a request is issued from the end-node device to delivering results to itself. Table 3.1 illustrates that the literature lacks an end-to-end solution.

3.4 Online Learning Framework

Our goal is to make offloading decisions and inference model selections in order to minimize inference latency while achieving acceptable accuracy. To do so, we first define the optimization problem, then we propose a reinforcement learning agent to solve the problem. Table 3.2 defines the notation used for the problem definition.

3.4.1 System Model and Problem Formulation

All computing devices in the end-edge-cloud system are represented by (S,E,C) where $S = \{S_1, S_2, .., S_n\}$ represents a set of end-node devices whose number is N; E represents the edge layer (in our case, a single device); C represents the cloud layer. Each end-node device requires a DL inference periodically. The inference model is selected from a pool of optimized models where each model has different characteristics including computational complexity and model accuracy. All device resources are represented in a tuple $\{P_i, M_i, B_i\}$ where P_i represents processor utilization of device i; M_i represents available memory for device i; B_i represents network's connection condition between the device i and upper layer's node.

The computation offloading decision determines whether each end-node device should offload an inference to higher-layer computing resources, or perform computation locally. The offload decision for each end-node device is represented by a tuple $o_i = \{o_i^S, o_i^E, o_i^C\}$ where o_i^j represents offloading decision to layer j. If end-node device i executes at layer $j \in \{S, E, C\}$, then $o_i^j = 1$; otherwise it must be zero. For a given end-node device i, the sum of all offloading decisions $\sum_j^{\{S,E,C\}} o_i^j$ must equal 1. $o = \{o_1, o_2, ..., o_n\}$ represents the offloading decision vector for all end-node devices. The inference model selection determines the implementation of the model deployed for each inference on each end-node device. Each end-node device S_i can perform inference with one of l DL models $\{d_1, d_2, d_3, ..., d_l\}$.

In general, response time is the total time between making a request to a service and receiving the result [86]. In our case, response time is the sum of the round trip transmission time from an end-node device to the node that performs the computation, plus the computation time. Response time T_{res} for a request from end-node device i with offload decision tuple $o_i = \{o_i^S, o_i^E, o_i^C\}$ can be summarized as follows:

$$T_{res_i} = o_i^S.T_{res}^S + o_i^E.T_{res}^E + o_i^C.T_{res}^C \tag{3.1}$$

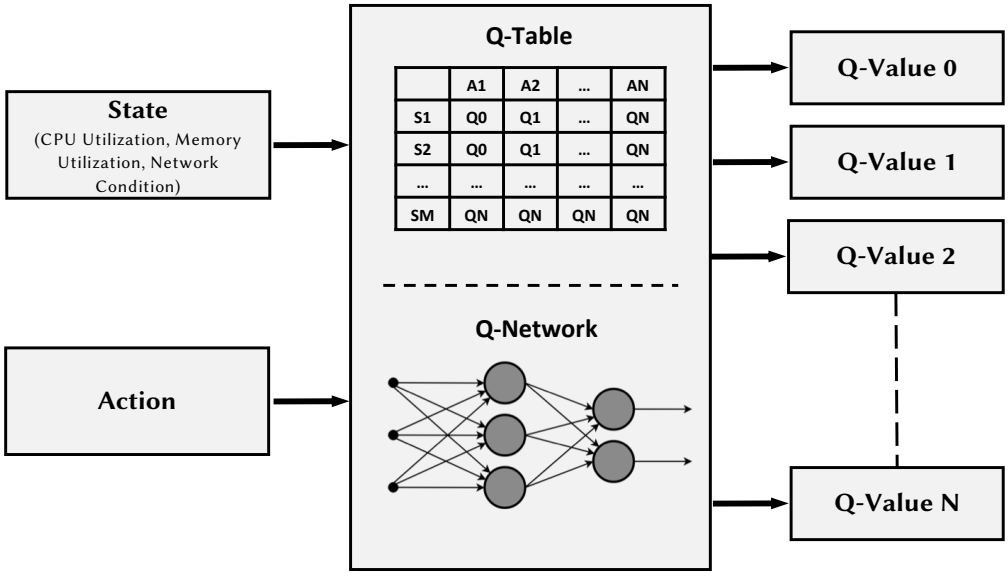


Figure 3.3: Proposed reinforcement learning agent with Q-Learning and Deep Q-Learning algorithms. Q-Learning uses a Q-Table to store $Q(S, A)$ values, Deep Q-Learning estimates Q-Values with a neural network architecture.

Our objective is to minimize the average response time while satisfying the average accuracy constraint. The problem is formulated in the following formula:

$$\begin{aligned} \textbf{P1:} \min \quad & \frac{1}{N}\sum_{i=1}^{N} T_{res_i}(o_i, d_k) \\ \text{s.t.} \quad & \overline{accuracy} > threshold \end{aligned} \tag{3.2}$$

where $\overline{accuracy}$ is the spatial average accuracy for simultaneous DL inferences.

3.4.2 Reinforcement Learning Agent

Reinforcement learning (RL) is widely used to automate intelligent decision making based on experience. Information collected over time is processed to formulate a policy which is based on a set of rules. Each rule consists three major components viz., (a) state, (b) action, and (c) reward. Among the various RL algorithms [116], Q-learning has low execution overhead, which makes it a good candidate for runtime invocation. However, it is ineffective for large

space problems. There are two main problems with Q-learning for large space problems [77]: (a) required memory to save and update the Q-Values increases as the number of actions and state increases. (b) required time to populate the table with accurate estimates is impractical for the large Q-table. In our case, increasing number of users will increase the problem's space dimension. The reason is more number of users leads to more number of rows and columns in the Q-table. Therefore, it takes more time to explore every state and update the Q-values. Due to the curse of dimensionality, function approximation is more appealing [77]. The Deep Q-Learning (DQL) algorithm combines the Q-Learning algorithm with deep neural networks. DQL uses Neural Network architecture to estimate the Q-function by replacing the need for a table to store the Q-values. In this work, we build an RL agent using two reinforcement learning algorithms: (a) an epsilon-greedy **Q-Learning** and (b) a **Deep Q-Learning** algorithms. We evaluate the RL agent with the mentioned algorithms considering different problem complexities. Figure 3.3 depicts high-level black diagram for our agent. The RL agent is invoked at runtime for intelligent orchestration decisions. In general, the agent is composed as follows:

State Space: Our state vector is composed of CPU utilization, available memory, and bandwidth per each computing resource. Table 3.3 shows the discrete values for each component of the state. The state vector at time step τ is defined as follows:

$$S_\tau = \{P^E, M^E, B^E, P^C, M^C, B^C, P^{S_1}, M^{S_1}, B^{S_1}, ..., P^{S_n}, M^{S_n}, B^{S_n}\} \tag{3.3}$$

Action Space: The action vector consists of which inference model to deploy, and which layer to assign the inference. We limit the edge and cloud devices to always use the high accuracy inference model, and the end-node devices have a choice of l different models. Therefore, the action space is defined as $a_\tau = \{o^i, d_j\}$ where $i \in \{S, E, C\}$ and $d_j \in \{d_1, d_2, ..., d_l\}$.

Reward Function: The reward function is defined as the negative average response time

Table 3.3: State Discrete Values

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

of DL inference requests. In our case, the agent seeks to minimize the average response time. To ensure the agent minimizes the average response time while satisfying the accuracy constraint, the reward R is calculated as follows:

$$\begin{aligned} &\text{if } \overline{accuracy} > \text{threshold:} \\ &\qquad R_\tau \leftarrow -Average\ Response\ Time \\ &\text{else:} \\ &\qquad R_\tau \leftarrow -Maximum\ Response\ Time \end{aligned} \tag{3.4}$$

To apply the accuracy constraint, the minimum possible reward is assigned when the accuracy threshold is violated. On the other hand, when the selected action satisfies the average accuracy constraint, the reward is negative average response time.

Q-Learning Algorithm

Q-Learning algorithm is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. The algorithm does not require a model of the environment where it can handle problems with stochastic transitions and rewards without requiring adaptations. The Q-Learning algorithm stores data in a Q-table. The structure of a Q-

Learning agent is a table with the states as rows and the actions as the columns. Each cell of the Q-table stores a Q-value, which estimates the cumulative immediate and future reward of the associated state-action pair. Epsilon-greedy is a common enhancement to Q-Learning that helps avoid getting stuck at local optima [116]. Algorithm 5 defines our agent's logic with the epsilon-greedy Q-Learning:

Line Description

3: First the agent determines the current system state from the resource monitors.

4-8: Next, either the state-action pair (S_τ, A_τ) with the highest Q-value is identified to choose the next action to take, or a random action is selected with probability ϵ.

9-10: The selected action is applied and normal execution resumes. After all inferences are completed, the reward R_τ for the execution period is calculated based on measured response time.

11-12: Based on the resource monitors, the new state $A_{\tau+1}$ is identified, along with the state-action pair with highest Q-value.

13: The Q-value of the previous state-action pair is updated.

14: The current state is updated, and the loop continues.

Deep Q-Learning Algorithm

Q-Learning has been applied to solve many real-world problems. However, it is unable to solve high-dimensional problems with many inputs and outputs [77] as it is impractical to represent the Q-function as a Q-table for large pair of S and A. In addition, it is unable to transverse $Q(S, A)$ pairs. Therefore, a neural network is used to estimate the Q-values. The

Algorithm 2 Q-Learning Algorithm

1: **Initialization in design time:**
   τ represents time step
   S_τ represents state at τ
   A_τ represents action at τ
2: **while** system is on **do**
3: **From *Resource Monitoring*:**
     $S_\tau \leftarrow$ State at step τ
4: **if** $RAND < \epsilon$ **then**
5: Choose random action A_τ
6: **else**
7: Choose action A_τ with largest $Q(S_\tau, A_\tau)$
8: **end if**
9: Monitor the response time for each devices
10: Calculate reward R_τ
11: **From *Resource Monitoring*:**
     $S_{\tau+1} \leftarrow$ State at step $\tau + 1$
12: Choose action $A_{\tau+1}$ with the largest $Q(S_{\tau+1}, A_{\tau+1})$
13: **To *Updating Qtable*:**
     $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[R_\tau + \gamma.Q(S_{\tau+1}, A_{\tau+1}) - Q(S_\tau, A_\tau)]$
14: $S_\tau \leftarrow S_{\tau+1}$
15: **end while**

Deep Q-learning Network (DQN) inputs include current state and possible action, and outputs the corresponding Q-value of the given action input. The neural network approximation is capable of handling high dimensional space problems [128]. One of the main problems with Deep Q-Learning is stability [77]. In order to reduce the instability caused by training on correlated sequential data, we improve the DQL algorithm with *replay buffer* technique [62]. During the training, we calculate the loss and its gradient using a mini-batch from the buffer. Every time the agent takes a *step* (moves to the next state after choosing an action), we push a *record* into the buffer. Algorithm 2 defines Deep Q-Learning algorithm which is described below:

Line Description

4: First the agent determines the current system state from the resource monitors.

4:9 Next, either the state-action pair S_τ, A_τ with the highest Q-value estimated by neural

network (θ) is identified to choose the next action to take, or a random action is selected with probability ϵ.

10:11 The selected action is applied and normal execution resumes. After all inferences are completed, the reward R_τ for the execution period is calculated based on measured response time.

12: At each time step, each record (S_τ,A_τ,R_τ,$S_{\tau+1}$) is added to a circular buffer D called the *replay buffer*.

13: We randomly sample *Batch Size records* from the buffer and then feed it to the network as mini-batch.

14: We calculate the temporal difference loss on the mini-batch and perform a gradient descent calculation to update the network. The *temporal difference loss* function calculates the *mean-square error* of the predicted and target Q-values as the loss of the mini-batch.

15: The current state is updated, and the loop continues.

There is few study about the theoretical analysis of the computational complexity of reinforcement learning because of the problem itself that reinforcement learning solves is hard to be explicitly modeled. Reinforcement learning is in the nature of trail-and-error and exploration-and-exploitation, which involves randomness and makes it difficult to be theoretically analyzed.

The complexity of problem for Bruteforce strategy is discussed in the following: Brute-force strategy searches the entire State×Action space of the problem and sort corresponding Q-values in order to find out the optimal action. Therefore, we can define the state space

complexity as follows:

$$(L_{CPU} \times L_{Network} \times L_{Memory})^N (L'_{CPU} \times L'_{Network} \times L'_{Memory})^2 \tag{3.5}$$

where, N stands for the number of end-devices. L_{CPU}, L_{Memory}, and $L_{Network}$ represent the number of CPU, Memory, and Network condition levels for end devices, respectively. In addition, L'_{CPU}, L'_{Memory}, and $L'_{Network}$ represent the number of CPU, Memory, and Network condition levels for edge and cloud devices, respectively. Besides, the action space is defined as $(Number\ of\ Actions)^N$. Therefore, the complexity is defined as follows:

$$(L_{CPU} \times L_{Network} \times L_{Memory})^N \times (L'_{CPU} \times L'_{Network} \times L'_{Memory})^2 \times (Number\ of\ Actions)^N \tag{3.6}$$

Algorithm 3 Deep Q-Learning Algorithm with Experience Replay

Initialization in design time:
τ represents time step
S_τ represents state at τ
A_τ represents action at τ
Initialize replay buffer D to capacity N
Initialize action-value function Q with random weight θ
for epoch = 1, Epochs **do**
  for episode = 1, Episodes **do**
    From *Resource Monitoring*:
    $S_\tau \leftarrow$ State at step τ
    if $RAND < \epsilon$ **then**
      Choose random action A_τ
    else
      Choose action A_τ with largest $Q_\theta(S_\tau, A_\tau)$
    end if
    Monitor the response time for each devices
    Calculate reward R_τ
    Store the record (S_τ,A_τ,R_τ,$S_{\tau+1}$) into buffer D
    Sample random mini-batch of records from buffer D
    To *Updating Q-Network*:
    Compute temporal difference loss with respect to the network parameter θ
    $S_\tau \leftarrow S_{\tau+1}$
  end for
end for

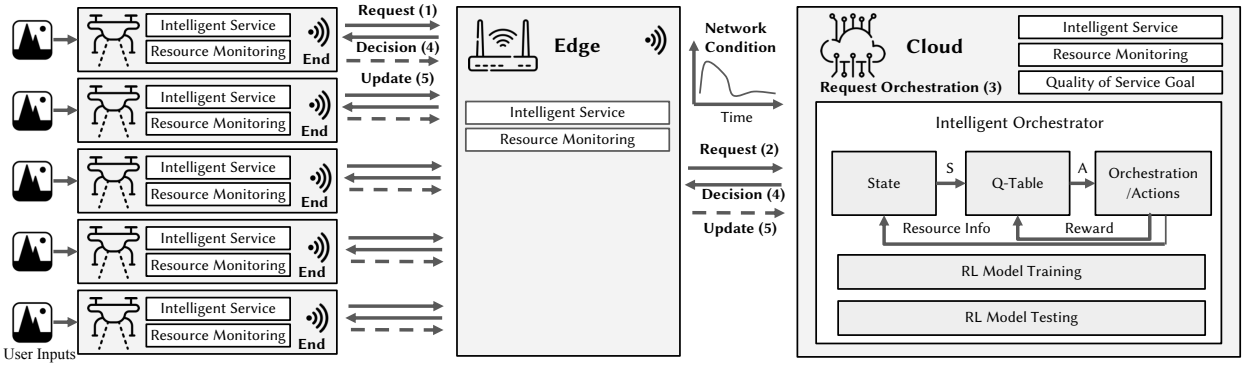


Figure 3.4: Orchestration framework with online learning for orchestrating DL inference.

The reinforcement learning agent requires distinct state-action pairs for training the Deep Q-network. To generate distinct state-action pair vectors, our proposed framework supports execution requests that are submitted by all the end-devices synchronously. With synchronous requests, we eliminate the discrepancy of different optimal actions for the same state vector.

3.5 Framework Setup

In this section we describe our proposed framework for dynamic computation offloading based on online learning, targeted at multi-layered end-edge-cloud architecture.

3.5.1 Framework Architecture

Figure 3.4 shows our proposed framework for end-edge-cloud architecture, integrating service requests, resource monitoring, and intelligent orchestration. The *Intelligent Orchestrator* (IO) acts as an RL-agent for making computation offloading and model selection decisions. The end-device layer consists of multiple user-end devices. Each end-device has two software components: (i) *Intelligent Service* - an image classification kernel with DL models of varying compute intensity and prediction accuracy; (ii) *Resource Monitoring* - a periodic service that

collects devices' system parameters including CPU and memory utilization, and network condition, and broadcasts the information to the edge and cloud layers. Both the edge and cloud layers also have the *Intelligent Service* and *Resource Monitoring* components. The *Intelligent Orchestrator* acts a centralized RL-agent that is hosted at the cloud layer for inference orchestration. The agent collects resource information (e.g., processor utilization, available memory, available bandwidth) from Resource Monitoring components throughout the network. The agent also gathers the reward information (i.e., response time) from the environment in order to learn an optimal policy. The agent builds the Q-function based on the RL algorithm. It builds a Q-Table for Q-Learning algorithm and a Q-Network for Deep Q-Learning algorithm based on cumulative reward obtained from the environment over time. *Quality of Service Goal* provides the required QoS for the system (i.e., the accuracy constraint).

Figure 3.4 illustrates the procedure step-wise of the inference service in our framework. The end-device layer consists of resource-constrained devices that periodically make requests to a DL inference service (step 1). The requests are passed through the edge layer (step 2) to the cloud device to be processed by *Intelligent Orchestrator* (step 3). The agent determines where the computation should be executed, and delivers the *Decision* to the network (step 4). Each device updates the agent after it performs an inference with the response time information of the requested service (step 5). In addition, all devices in the framework send the available resource information including the processor utilization, available memory, and network condition to the cloud device (step 5).

3.5.2 Benchmarks and Scenarios

MobileNets are small, low-latency deep learning models trained for efficient execution of image classification on resource-constrained devices [36]. For DL workloads, we consider

MobileNetV1 image classification application as the benchmark [36]. We deploy the MobileNetV1 service for end-node classification. We consider eight different MobileNet models ($d0$ through $d7$) with varying levels of accuracy and performance. Each model among $d0$ through $d7$ has varying number of Multiply-Accumulate units (MACs), MAC width and data format (e.g., FP32 and Int8), exposing models with different accuracy-performance trade-offs. Table 3.4 summarizes the MobileNet models we consider, detailing the number of Multiply-Accumulates (MACs), MAC width and data formats (e.g., FP32 and Int8). The multiplier width is used to reduce a network's size uniformly at each layer. For a given layer and multiplier width, the number of input channels and the number of output channels is decreased and increased, respectively, by a factor of the width multiplier. During the orchestration phase, we select an appropriate model from $d0$-$d7$ to achieve the target level of classification accuracy while maximizing the performance.

Our framework supports multiple end-devices, networked with edge and cloud layers. For evaluation purposes, we set the maximum number of simultaneously active user devices to five. Each user-end device is connected to a single edge device, and can request a DL inference service to the cloud layer. The cloud layer hosts the IO that contains the RL agent, which handles the inference service requests. Upon on each service request, the RL agent is invoked to determine: (i) where the request should be processed and (ii) what DL

Table 3.4: MobileNet Models [36]

#	Model	Million MACs	Type	Top-1 Accuracy (%)	Top-5 Accuracy (%)
$d0$	1.0 MobileNetV1-224	569	FP32	70.9	89.9
$d1$	0.75 MobileNetV1-224	317	FP32	68.4	88.2
$d2$	0.5 MobileNetV1-224	150	FP32	63.3	84.9
$d3$	0.25 MobileNetV1-224	41	FP32	49.8	74.2
$d4$	1.0 MobileNetV1-224	569	Int8	70.1	88.9
$d5$	0.75 MobileNetV1-224	317	Int8	66.8	87.0
$d6$	0.5 MobileNetV1-224	150	Int8	60.7	83.2
$d7$	0.25 MobileNetV1-224	41	Int8	48.0	72.8

Table 3.5: Experiment Environment Setup. *R* and *W* represent *Regular* and *Weak* network condition, respectively.

Experiment	S1	S2	S3	S4	S5	E
EXP-A	R	R	R	R	R	R
EXP-B	R	W	R	W	R	W
EXP-C	W	W	W	R	R	R
EXP-D	W	W	W	W	W	W

model should be executed for the corresponding request. The RL agent's goal is to minimize average response time for all end-node devices while satisfying the accuracy constraint. This enforces quality control by imposing a strict threshold on the average DL model accuracy. In this work, we conduct experiments under four unique scenarios with varying network conditions. Each scenario represents a combination of regular (R) and weak (W) network signal strength over five user-end devices (S1-S5) and 1 edge device (E). The experimental scenarios are summarized in Table 3.5. The regular network has no transmission delay, while we add 20ms delay to all outgoing packets to emulate the weak connection behavior. Each experimental scenario in Table 3.5 shows the network condition of the specific device. Putting together the five different user devices and one edge device forms a unique combination of varying network conditions per each experimental scenario.

3.5.3 Experimental Setup

The platform consists of five AWS a1.medium instances with single ARM-core as end-devices connected to an AWS a1.large instance as edge device and an AWS a1.xlarge instance as cloud node. Table 3.6 summarizes device specifications in details. DL model inferences are executed on processor cores on all nodes using ARM-NN SDK [69]. The inference engine is a set of open-source Linux software tools that enables machine learning workloads on ARM-core-based devices. The framework's message passing protocol is implemented using web services deployed at each node. Section 3.6.2 provides our analysis on framework's setup

overhead.

3.5.4 Hyper-parameters and RL Training

An RL agent has a number of hyper-parameters that impact its effectiveness (e.g., learning rate, epsilon, discount factor, and decay rate). The ideal values of parameters depend on the problem complexity, which in our case scales with the number of users (i.e., active end-node devices). In order to determine the learning rate and discount factor, we evaluated values between 0 and 1 for each hyper-parameters. We observed that a higher learning rate converges faster to the optimal, meaning the more the reward is reflected to the Q-values, better the agent works. We also observed that a lower discount factor is better. This means that the consecutive actions have a weak relationship, so that giving less weight to the rewards in the near future improves the convergence time. Table 3.7 shows the different problem configurations we used to determine the hyper-parameters. We train the agent with two different learning algorithms (See Section 3.4.2). Our Q-Learning agent initializes a Q-table with Q-values of zero, and chooses actions using an $\epsilon - greedy$ policy where ϵ is the exploration rate. We initially set $\epsilon = 1$, meaning the agent selects a random action with probability 1, otherwise it selects an action that gives the maximum future reward (i.e., Q-value) in the given state. Although we perform probabilistic exploration continuously, we decay the exploration by epsilon decay parameter (See Table 3.7) per agent invocation. The Deep Q-Learning agent uses different neural network structure for different number of users as the problem complexity changes. We train DNN models with two fully connected

Table 3.6: Device Specification

Node Type	**vCPUs**	**Memory (GiB)**	**Frequency (GHz)**	**Architecture**
End	1	2	2.3	aarch64
Edge	2	4	2.3	aarch64
Cloud	4	8	2.3	aarch64

Table 3.7: Hyper-parameter values

	Q-Learning		Deep Q-Learning	
Number of Users	**Learning Rate (α)**	**Epsilon Decay**	**Learning Rate (α)**	**Epsilon Decay**
1	0.9	1e-1	–	–
2	0.9	1e-2	–	–
3	0.9	1e-2	1e-3	0.4
4	0.9	1e-3	1e-3	0.7
5	0.9	1e-4	1e-3	0.9

layers where the hidden layers have 48, 64, 128 neurons for three, four, and five devices, respectively. We implement the experience replay as a FIFO buffer with size equal to 1000. In order to update the network, at each step, we randomly sample 64 records from the buffer and then use them as a mini-batch. We use $\epsilon - greedy$ policy to train the Deep Q-network, where we initially set the ϵ equal to 1.

3.6 Evaluation Results and Analysis

In this section, we demonstrate the effectiveness of our online learning based inference orchestration. We evaluate our approach on the multi-layered end-edge-cloud framework, described in Section 3.3.4. Our approach features online reinforcement learning for intelligent orchestration, DL inference services and end-edge-cloud architectures, targeting DL inference performance. [105] presents state-of-the-art machine learning based orchestration for end-edge-cloud architecture baseline. For a fair comparison, we evaluate our approach against the strategy proposed in [105], which integrates the aforementioned features of our approach.

3.6.1 Performance Analysis

We evaluate our agent's ability to identify the optimal orchestration decision at each invocation. Through reinforcement learning, the agent predicts orchestration decisions including

offloading policy and DL model configuration to maximize performance and meet the accuracy threshold. At design time, we determine the true optimal configuration in any given conditions of workloads, network, and number of active users using a brute force search. First, we compare our reinforcement learning based Intelligent Orchestrator's (IO) prediction accuracy against this true optimal configuration. Our proposed approach with both ***Q-Learning*** and ***Deep Q-Learning*** algorithm has yielded a 100% prediction accuracy in comparison with the true optimal configuration. Thus, our reinforcement learning based orchestration decisions always converge with the optimal solution. Next, we evaluate our agent's efficacy by comparing it with a representative state-of-the-art [105] baseline in terms of performance and accuracy. To implement the baseline policy into our framework, we limit the agent to actions that specify offloading decisions $a_\tau = \{o^i\}$, using the most accurate DL model. We additionally compare fixed orchestrations for points of reference. The fixed solution is limited to configurations where all end-devices either (a) perform the most accurate DL inference execution locally, (b) offload to the edge, or (c) offload to the cloud. In the following subsections, we demonstrate the efficacy of our proposed agent to find the optimal configuration in presence of different number of users (up to five). Then, we investigate its ability to adapt to network variations and evaluate its overhead. We explain the impact of varying DL models on the performance under different system dynamics and elaborate how the proposed agent follows the defined constraints.

User Variability

To evaluate the user variability, we consider up to five simultaneously active user-end devices, keeping the network constraints constant. We consider five different levels of accuracy thresholds viz., *Min*, *80%*, *85%*, *89%*, and *Max*. *Min* refers to the accuracy threshold for computing where no constraint is applied to the learning algorithms (See Equation 4) and *Max* represents the accuracy threshold for computing where the average accuracy constraint

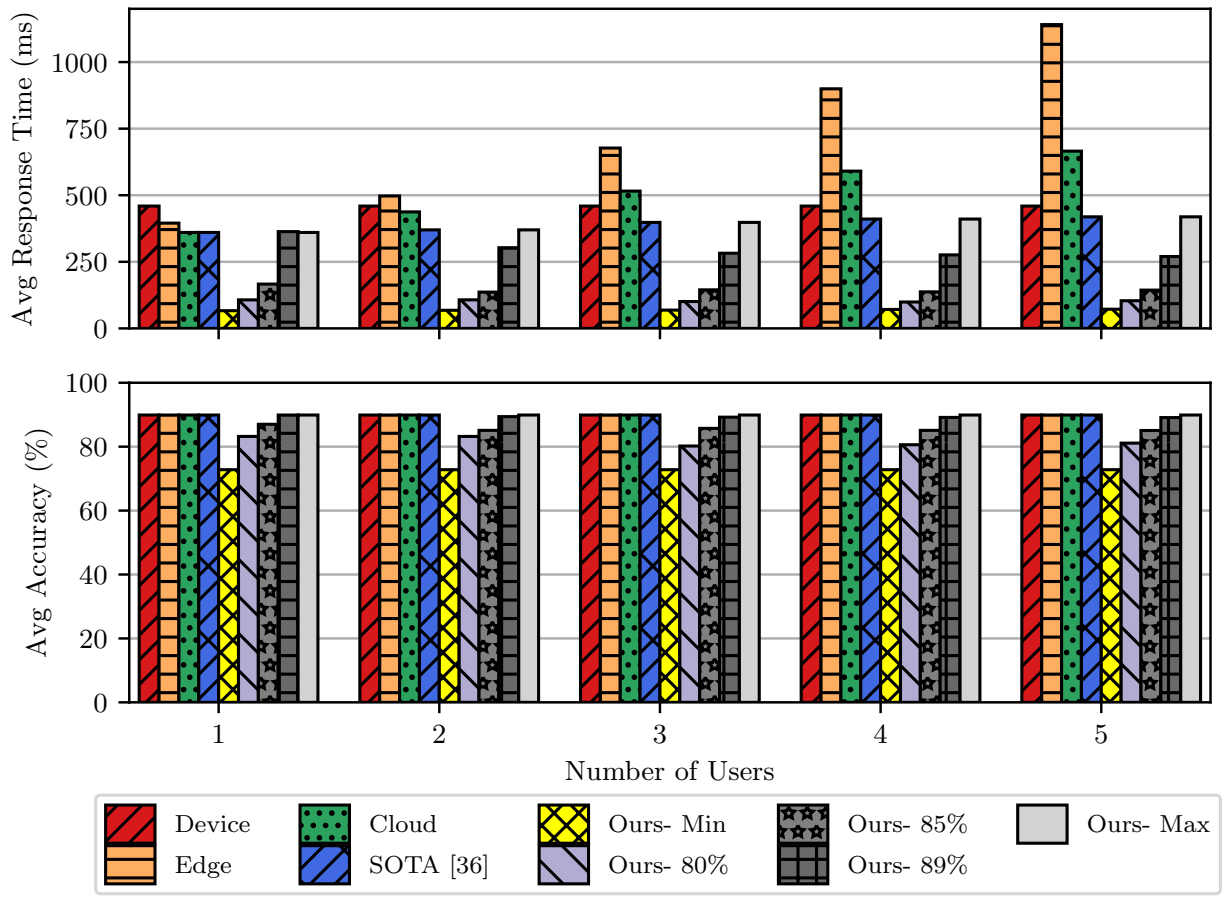


Figure 3.5: Results of the framework within Exp-A for different number of active users.

is set to 89.9%. We present the average response time and average accuracy for each of these thresholds using our proposed approach. For evaluation, we also present the average response time and accuracy metrics achieved with the state-of-the-art baseline approach [105], and three fixed orchestration decisions viz., device only, edge only and cloud only.

Fixed Strategies Figure 3.5 shows the average response time and accuracy for different numbers of active users for regular network conditions (represented by scenario Exp-A in Table 3.5), using different orchestration strategies. The x-axis represents the number of active users. Each bar represents a different orchestration decision made by using the corresponding orchestration strategy. With the device only strategy, each user-end device executes the inference service on the local device. Thus, varying number of users has no effect on the average response time in this case. With the edge and cloud only strategies, simultaneous

requests contend for edge and cloud resources. This increases the average response time significantly, as the number of users increase. For instance, the fixed edge only strategy with five active users leads to an average response time of 1140ms, while it is $665ms$ with cloud only strategy. Higher volume of available resources at the cloud layer results in relatively better average response time in comparison with the edge only strategy. On the other hand, the average response time with the device only strategy is $459ms$, representing the optimal case.

Baseline With the SOTA [105] approach, the average response time remains constant until the number of users is two. This is due to the orchestration decision of distributing the services across edge and cloud layers. As the number of users increase to three, the service requests contend for resources, leading to an increase in the average response time. With the number of users increasing from three through five, the average response time increases, but at a relatively lower rate, exhibiting efficient utilization of the edge and cloud resources. As the number of users increase, the efficiency of the baseline approach over the fixed strategies is more prominent. Both the baseline and fixed strategies are agnostic to model selection and configuration, retaining the maximum prediction accuracy of the inference service. Thus the average accuracy remains constant with the aforementioned strategies, as shown in Figure 3.5.

Our proposed solution Our proposed solution achieves the same average response time in comparison with the baseline for the *Max* accuracy scenario. When the accuracy threshold is relaxed, our reinforcement learning based intelligent orchestrator selects appropriate models (among $d0$-$d7$) to improve the average response time. As the number of users increase, our solution leverages the model selection combined with offloading technique to address the potential increase in response time. With appropriate model selection, our approach reduces the compute intensity, and consequently maintains a lower average response time even with the increasing number of users. Trivially, the average response time with our approach is lower as the accuracy threshold is reduced. However, it should be noted that we enforce the

Table 3.8: Detailed offloading decisions of our agent for different number of active users in all four experiments (Maximum Accuracy Threshold). For example, in Exp-A, the orchestrator offloads the most accurate DL inference execution ($d0$) to the cloud device ($d0, C$ for end-node $S1$). In the presence of five active users, the decisions are $\{d0, E\}$, $\{d0, L\}$, $\{d0, L\}$, $\{d0, C\}$, and $\{d0, L\}$ for end-nodes $S1$ to $S5$, respectively. In this case, $S1$, $S2$, and $S4$ perform DL inference execution of the $d0$ model locally (L). $S0$ and $S3$ offload inference execution of the $d0$ model to the edge (E) and cloud (C), respectively.

			End-node Devices					
	Experiments	**Number of Users**	**S1**	**S2**	**S3**	**S4**	**S5**	**Avg Res (ms)**
Decision	**Exp-A**	1	$d0, C$	–	–	–	–	363.47
		2	$d0, C$	$d0, E$	–	–	–	363.17
		3	$d0, C$	$d0, L$	$d0, E$	–	–	397.53
		4	$d0, L$	$d0, L$	$d0, E$	$d0, C$	–	410.35
		5	$d0, E$	$d0, L$	$d0, L$	$d0, C$	$d0, L$	418.91
	Exp-B	1	$d0, E$	–	–	–	–	403.30
		2	$d0, E$	$d0, C$	–	–	–	416.78
		3	$d0, E$	$d0, C$	$d0, L$	–	–	431.90
		4	$d0, L$	$d0, C$	$d0, E$	$d0, L$	–	457.96
		5	$d0, C$	$d0, E$	$d0, L$	$d0, L$	$d0, L$	472.88
	Exp-C	1	$d0, C$	–	–	–	–	471.65
		2	$d0, C$	$d0, E$	–	–	–	467.80
		3	$d0, C$	$d0, E$	$d0, L$	–	–	488.21
		4	$d0, C$	$d0, E$	$d0, L$	$d0, L$	–	480.70
		5	$d0, L$	$d0, L$	$d0, L$	$d0, C$	$d0, E$	464.59
	Exp-D	1	$d0, L$	–	–	–	–	585.68
		2	$d0, E$	$d0, C$	–	–	–	527.39
		3	$d0, L$	$d0, C$	$d0, E$	–	–	491.77
		4	$d0, L$	$d0, C$	$d0, E$	$d0, L$	–	501.07
		5	$d0, L$	$d0, C$	$d0, E$	$d0, L$	$d0, L$	506.62

boundaries on tolerable loss of accuracy with our model selection decisions. Figure 3.5 shows the average response time and average accuracy with our solution over different scenarios of accuracy thresholds and varying number of users. Our solution provides up to 35% improvement in the average response time in comparison with the baseline, within a tolerable loss of 0.9% accuracy. Table 3.8 shows the orchestration decisions of our agent for different numbers of active users, and also over four different experimental scenarios (Table 3.5). We present the orchestration decision and the average response time achieved with each decision, for the maximum accuracy threshold scenario.

Table 3.9: Results of the proposed framework for different accuracy constraints for different experiments (five users). For example, in Exp-D with 89% average accuracy constraint, our framework orchestrates $S1$, $S2$, $S3$, and $S4$ to execute DL inference using model $d4$ locally and offload inference execution using model $d0$ at the cloud. However, the baseline obtains the maximum accuracy by executing the most accurate DL inference locally for $S1$, $S4$, and $S5$ while offloading $d0$ to the edge and cloud for $S3$ and $S2$, respectively.

			End-node Devices						
	Experiments	**Constraint**	**S1**	**S2**	**S3**	**S4**	**S5**	**Avg Res (ms)**	**Avg Acc (%)**
Decision	**Exp-A**	Min	$d7, L$	$d7, L$	$d7, L$	$d7, L$	$d7, L$	72.08	72.80
		80%	$d7, L$	$d6, L$	$d6, L$	$d6, L$	$d6, L$	103.88	81.11
		85%	$d2, L$	$d6, L$	$d5, L$	$d6, L$	$d5, L$	143.81	85.06
		89%	$d4, L$	$d4, L$	$d4, L$	$d0, E$	$d4, L$	269.80	89.10
		Max	$d0, E$	$d0, L$	$d0, L$	$d0, C$	$d0, L$	418.91	89.90
	Exp-B	Min	$d7, L$	$d7, L$	$d7, L$	$d7, L$	$d7, L$	106.76	72.80
		80%	$d6, L$	$d3, L$	$d6, L$	$d6, L$	$d6, L$	139.92	83.23
		85%	$d5, L$	$d5, L$	$d6, L$	$d6, L$	$d2, L$	176.21	85.05
		89%	$d4, L$	$d4, L$	$d0, E$	$d4, L$	$d4, L$	303.50	89.10
		Max	$d0, C$	$d0, E$	$d0, L$	$d0, L$	$d0, L$	472.88	89.90
	Exp-C	Min	$d7, L$	$d7, L$	$d7, L$	$d7, L$	$d7, L$	119.28	72.80
		80%	$d6, L$	$d6, L$	$d7, L$	$d6, L$	$d6, L$	149.52	81.11
		85%	$d5, L$	$d6, L$	$d5, L$	$d6, L$	$d5, L$	190.76	85.47
		89%	$d4, L$	$d4, L$	$d4, L$	$d4, L$	$d0, C$	318.45	89.10
		Max	$d0, L$	$d0, L$	$d0, L$	$d0, C$	$d0, E$	464.59	89.90
	Exp-D	Min	$d7, L$	$d6, L$	$d7, L$	$d7, L$	$d7, L$	158.53	72.80
		80%	$d6, L$	$d6, L$	$d6, L$	$d7, L$	$d6, L$	182.53	81.12
		85%	$d2, L$	$d6, L$	$d6, L$	$d5, L$	$d5, L$	225.32	85.06
		89%	$d4, L$	$d4, L$	$d4, L$	$d4, L$	$d0, C$	356.75	89.10
		Max	$d0, L$	$d0, C$	$d0, E$	$d0, L$	$d0, L$	506.62	89.90

Table 3.10: Results of the state-of-the-art [105] in all four experiments.

		End-node Devices						
	Experiments	**S1**	**S2**	**S3**	**S4**	**S5**	**Avg Res (ms)**	**Avg Acc (%)**
Decision	**Exp-A**	$d0, E$	$d0, L$	$d0, L$	$d0, C$	$d0, L$	418.91	89.9
	Exp-B	$d0, C$	$d0, E$	$d0, L$	$d0, L$	$d0, L$	472.88	89.9
	Exp-C	$d0, L$	$d0, L$	$d0, L$	$d0, C$	$d0, E$	464.59	89.9
	Exp-D	$d0, L$	$d0, C$	$d0, E$	$d0, L$	$d0, L$	506.62	89.9

Network variation

We consider two possible levels of network connection: (i) a regular network that has low latency, and (ii) a weak network that has high latency. We add 20ms delay to all outgoing packets to emulate the weak connection behavior. With varying network conditions, there is an increased delay with offloading decisions across the network. Both the baseline and fixed approaches are affected by the weak network conditions, resulting in a higher average response time. The fixed strategies employ the trivial device, edge and cloud only offloading decisions, suffering higher latency. The baseline approach is confined to only an intelligent offloading strategy, which also results in higher average response time inevitably. On the other hand, our proposed solution adapts to varying network conditions by opportunistically exploiting the accuracy trade-offs through model selection. This way, we address for the latency penalty levied by weak network conditions by reducing the compute intensity of the workloads, within the tolerable accuracy bounds.

Table 3.9 shows the orchestration decisions made by our intelligent orchestrator, average response time, and average accuracy achieved over varying networking conditions. Each experiment scenario (Exp-A through Exp-D) combines different network conditions for each node in the network (See Table 3.5). For example, in Exp-A, all the nodes are connected with regular network, whereas in Exp-B, nodes $S1$, $S3$, and $S5$ have regular connections and the rest have weak connections. We set the number of active users to five.

Model Selection Within each experiment scenario, the average response is lower as the accuracy threshold is relaxed. $d0$ through $d7$ represent models with different response time and accuracy levels. For instance, models $d0$, $d4$, $d2$, $d7$ and $d7$ are selected respectively for accuracy thresholds ranging from *Max* through *Min* in Exp-A. Our proposed orchestrator explores the Pareto-optimal space of model selection and offloading choice, combining the opportunities at application and platform layers simultaneously. For instance in Exp-A, maintaining an accuracy level of 89% results in an average response time of 269.8ms, by i)

setting the models to $d4$, $d4$, $d4$, $d0$, and $d4$ on devices S1-S5, and ii) device configurations to L (local device), L, L, E (edge) and L for S1-S5. However, the average response time can be improved by sacrificing the accuracy within a pre-determined tolerable level. For instance, by lowering the accuracy threshold by 4% (from 89% to 85%), the average response time can be reduced by 88% (from 269ms to 143ms) by i) setting the models to $d2$, $d6$, $d5$, $d6$, and $d5$ on devices S1-S5, and ii) device configurations to L (local device), L, L, L and L for S1-S5. With varying network conditions, our solution explores the offloading and model selection Pareto-optimal space at run-time to predict the optimal orchestration decisions.

For example, in Exp-D, our framework obtains 356.75ms on average response time with significantly weak network connectivity, while it can adapt to regular connectivity in Exp-A to obtain 269.80ms on average response time. In this case, the average accuracy is 89.1% which shows 0.8% error with the maximum average accuracy. The baseline [105] orchestrates the most accurate DL inference execution to obtain 506.62ms and 418.9ms average response time in Exp-D and Exp-A, respectively. Orchestration decisions of the baseline approach over different experimental scenarios is summarized in Table 3.10. Although our proposed framework and the baseline can adapt to network variability, our agent provides additional trade-off opportunities to deploy different models combined with offloading technique. This leads to up to 35% speedup while sacrificing less than 1% average accuracy.

3.6.2 Overhead Analysis

Developing a global RL agent for optimal runtime orchestration decisions in an end-edge-cloud system incurs overhead to multiple sources. We evaluate the sources of overhead in both exploration and exploitation phases to demonstrate the feasibility of our proposed solution.

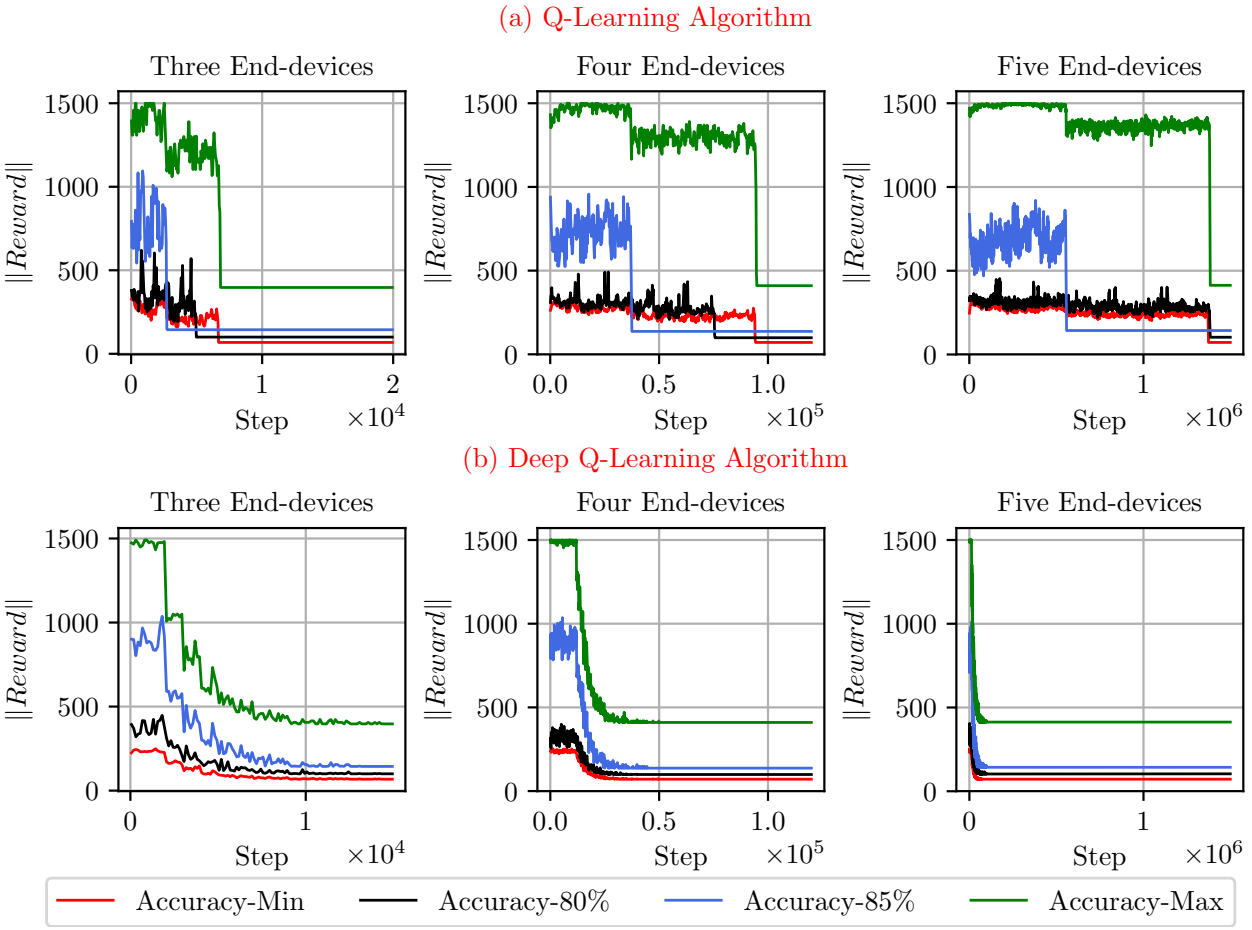


Figure 3.6: Training overhead for multi-user networks with ***Q-Learning*** and ***Deep Q-Learning*** algorithms under different accuracy constraints (See Algorithm 1 and 2, respectively).

Exploration Overhead

We evaluate the time required by the proposed agent for the training phase to identify an optimal policy. Figure 3.6 shows the training phase for different numbers of end-devices under different accuracy constraints. We train the agent with ***Q-Learning*** and ***Deep Q-Learning*** algorithms under different accuracy constraints (See Figure 3.6.(a) and 3.6.(b), respectively). The convergence time for five devices with different policies are summarized in Table 3.11. Q-Learning agent converges faster than Deep Q-Learning agent for the three End-devices scenario. However, increasing the number of End-devices leads to the more complex problem. Deep Q-Learning agent converges up to 17.5× faster than Q-Learning agent for

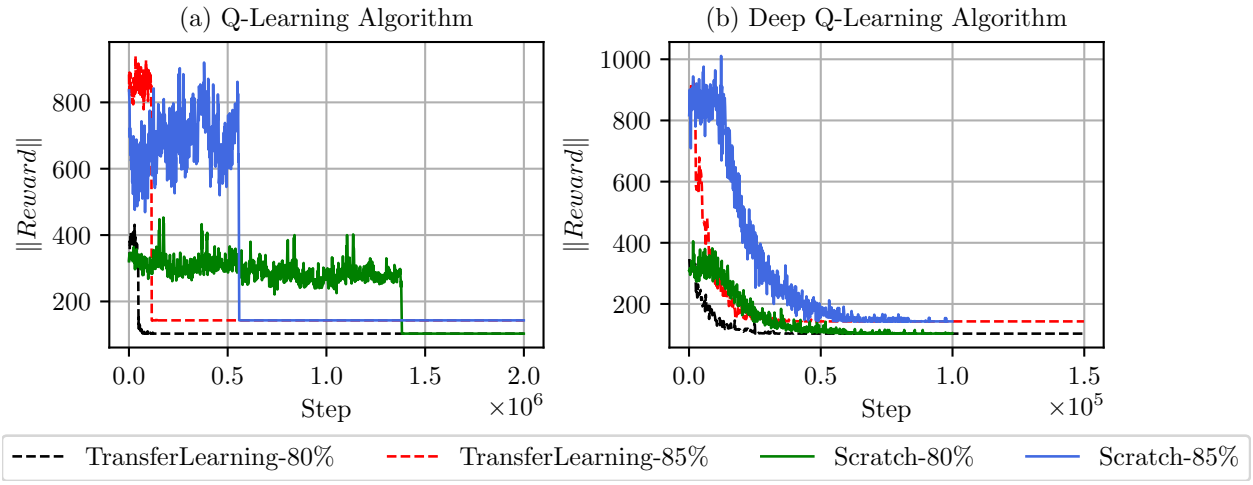


Figure 3.7: Transfer learning strategy can be used to alleviate the convergence time. In our experiments, the strategy improves the convergence time up to 12.5× and 3.3× for Q-Learning and Deep Q-Learning for five End-devices, respectively. For example, the training phase for Q-Learning algorithm under 80% accuracy constraint converges at 10.5×10^5 steps. While, using the transfer learning it converges at 8.2×10^4 steps.

the five End-devices scenario. In other words, Deep Q-Learning algorithm converges faster for high-dimensional space problems. Furthermore, SOTA converges faster since the agent only uses limited actions (3 actions for computation offloading) making a low-dimensional space problem.

In addition, we observe that the training phase can be accelerated by exploiting previous experiences in similar scenarios known as transfer learning strategy. Figure 3.7 shows that the strategy can alleviate the convergence up to 12.5× and 3.3× for Q-Learning and Deep Q-Learning algorithms, respectively. In the transfer learning strategy, we train a model with minimum accuracy threshold from scratch. Then, we initialize model with the trained model to reduce the convergence time. In conclusion, the Deep Q-Learning algorithm with the transfer learning strategy can speedup the convergence time up to 57.7× in comparison with Q-Learning algorithm for the five End-devices scenario.

Table 3.11: Training convergence time for three, four , and five End-devices with Q-Learning and Deep Q-Learning algorithms compared with SOTA [105] and Bruteforce strategy (See Section 3.4)

Number of Users	**Constraint**	**Q-Learning** (step #)	**Deep Q-Learning** (step #)	**SOTA** [105]	**Bruteforce** (step #)
3	Min	6.6×10^3	1.0×10^4	-	6.6×10^8
	80%	1.8×10^3	1.0×10^4	-	6.6×10^8
	85%	0.8×10^3	1.0×10^4	-	6.6×10^8
	Max	6.7×10^3	1.0×10^4	2.0×10^3	6.6×10^8
4	Min	9.0×10^4	3.0×10^4	-	5.3×10^{10}
	80%	8.0×10^4	4.0×10^4	-	5.3×10^{10}
	85%	4.0×10^4	4.0×10^4	-	5.3×10^{10}
	Max	9.0×10^4	4.0×10^4	5.0×10^3	5.3×10^{10}
5	Min	10.5×10^5	6.0×10^4	-	4.2×10^{12}
	80%	10.5×10^5	6.0×10^4	-	4.2×10^{12}
	85%	5.6×10^5	7.0×10^4	-	4.2×10^{12}
	Max	10.5×10^5	7.0×10^4	2.5×10^4	4.2×10^{12}

.

Run-time Overhead

The agent is invoked periodically at runtime, imposing overhead on DL inference execution. We evaluate the following components individually:

(a) *Resource Monitoring:* A continuous resource monitoring service imposes runtime overhead in terms of DL inference response time. Figure 3.8 shows that the latency overhead for all layers is negligible (less than 0.8% of minimum response time overall).

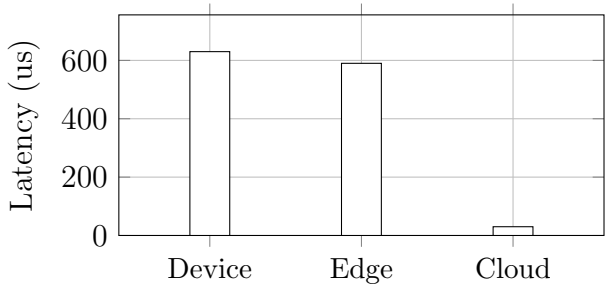


Figure 3.8: Resource Monitoring Overhead

(b) *Message Broadcasting:* Sharing resource usage and orchestration decision information over the network potentially increases DL inference response time. Table 3.12 shows the additional network latency for different network conditions. The *request* is the latency required to send an input image to a higher layer, and dominates the sources of network overhead. We observe that the broadcasting, in total, does not impose more than 2% of overall response

time.

(c) *Intelligent Orchestrator:* The Q-Learning agent's logic itself takes on average 0.6ms to execute in the cloud. While, the Deep Q-Learning agent's step takes $11ms$ on average to execute using NVIDIA RTX 5000 in cloud. During exploitation, our trained agent identifies the optimal orchestration decision within five invocations. We conclude that after an agent is trained, the improvements of 35% in average response time compared to prior art justifies the total overhead of our agent.

Table 3.12: Message Broadcasting Overhead

	Regular	**Weak**
Request	20 ms	137 ms
Update	0.4 ms	2 ms
Decision	1 ms	2 ms
Total	21.4 ms	141 ms

3.7 Summary

Cross-layer optimization that considers both model optimization and computation offloading together provides an opportunity to enhance performance while satisfying accuracy requirements. In this chapter, for the first time, we proposed an online learning framework for DL inference in end-edge-cloud systems by coordinating tradeoffs synergistically at both the application and system layers. The proposed reinforcement learning-based online learning framework adopts model optimization techniques with computation offloading to find the minimum average response time for DL inference services while meeting an accuracy constraint. Using this method, we observed up to 35% speedup for average response time while sacrificing less than 0.9% accuracy on a real end-edge-cloud system when compared to prior art. Our approach shows that online learning can be deployed effectively for orchestrating DL inference in end-edge-cloud systems, and opens the door for further research in online learning for this important and growing area.

Chapter 4

Hybrid Learning for Orchestration Deep Learning Inference at Edge

4.1 Introduction

Deep-learning (DL) kernels provide intelligent end-user services in application domains such as computer vision, natural language processing, autonomous vehicles, and healthcare [104]. End-user mobile devices are resource-constrained and rely on cloud infrastructure to handle the compute intensity of DL kernels [123]. Unreliable network conditions and communication overhead in transmitting data from end-user devices affect real-time delivery of cloud services [51]. Edge computing brings compute capacity closer to end-user devices, and complement the cloud infrastructure in providing low latency services [134]. Collaborative end-edge-cloud (EEC) architecture enables on-demand computational offloading of DL kernels from resource-constrained end-user devices to resourceful edge and cloud nodes [107, 109]. Orchestrating DL services in multi-layered EEC architecture primarily focus on i) selecting an edge node onto which a task can be offloaded, and ii) selecting an appropriate learning

model to accomplish the DL task. Selection of the edge node for offloading a DL task is based on a combination of factors including i) the edge node's compute capacity and core-level heterogeneity, ii) communication penalty incurred in offloading, iii) workload intensity of the task and accuracy constraints, and iv) run-time variations in connectivity, signal strength, user mobility, and interaction. Selecting an appropriate model for a DL task depends on design time accuracy constraints and run-time latency constraints simultaneously. Different learning models for DL tasks expose a Pareto-space of accuracy-compute intensity, such that higher accuracy models consume longer execution time [123].

Orchestrating DL tasks by finding the appropriate edge node for offloading, and configuring the learning model for DL task, while minimizing the latency under unpredictable network conditions makes orchestration a multi-dimensional optimization problem [53]. Therefore, the orchestration problem requires an intelligent run-time management to search through a wide configuration Pareto-space. Brute force search, heuristic, rule-based, and closed-loop feedback control solutions for orchestration require longer periods of time before converging to optimal decisions, making them inefficient for real-time orchestration [116]. Reinforcement Learning (RL) approaches have been adopted for orchestrating DL tasks in multi-layered end-edge-cloud systems [53] to address these limitations. Orchestration strategies using RL can be classified into *model-free* and *model-based* approaches.

Model-free RL techniques operate with no assumptions about the system's dynamic or consequences of actions required to learn a policy. Model-free RL builds the policy model based on data collected through trial-and-error learning over epochs [116]. Existing *Model-free* RL strategies have used Deep Reinforcement Learning (DRL) algorithms for minimizing the latency for multi-service nodes in end-edge-cloud architectures [70]. AdaDeep [66] proposes a resource-aware DL model selection using optimal learning. AutoScale [53] proposes an energy-efficient computational offloading framework for DL inference. Originated from trial-and-error learning, *model-free* RL requires significant exploratory interactions with the environment [116]. Many of these interactions are impractical in distributed computer sys-

Table 4.1: State-of-the-art reinforcement learning-based orchestration frameworks for deep learning inference in end-edge-cloud networks. Approach- Model-free (MF) and Hybrid Learning (HL). Algorithm- Q-learning (QL), DeepQ (DQL), DeepDynaQ (DDQ). Actuation knobs- *CO*: computation offloading, *HW*: hardware knobs, *APP*: application layer knobs.

Technique	**Approach**	**Algorithm**	**Workload**	**Knobs**
AutoScale [53]	MF	QL	Inference	CO,HW
AutoFL [55]	MF	QL	Training	CO,HW
AdaDeep [66]	MF	DQL	Inference	APP
Ours	HL	DDQ	Inference	CO,APP

tems, since execution for each configuration is expensive and leads to higher latency and resource consumption during the learning process [116].

Model-based RL uses a predictive internal model of the system to seek outcomes while avoiding the consequence of trial-and-error in real-time. Existing approaches have modeled the computation offloading problem and then use deep reinforcement learning to find an optimal orchestration solution [138, 60] [122]. Model-based RL approach is computationally efficient and provides better generalization and significantly less number of real full system execution runs before converging to the optimal solution [116]. However, Model-based RL is sensitive to model bias and suffers from model errors between the predicted and actual behavior leading to sub-optimal orchestration decisions.

A hybrid learning approach integrating the advantages of both *model-free* and *model-based* RL is efficient for orchestrating DL tasks on end-edge-cloud architectures [93, 114]. In this work, we adapt such hybrid learning strategy for orchestrating deep learning tasks on distributed end-edge-cloud architectures. We model the end-edge-cloud system dynamics online, and design an RL agent that learns orchestration decisions. We incorporate the system model into the RL agent, which simulates the system and predicts the system behaviors. We exploit the hybrid *Deep Dyna-Q* [93, 114] model to design our RL agent, which requires fewer number of interactions with the end-to-end computer system, making the learning process efficient. Existing efforts to orchestrate DL inference/training over the network are summarized in Table 4.1. We compare our framework with AutoScale [53] and AdaDeep [66]

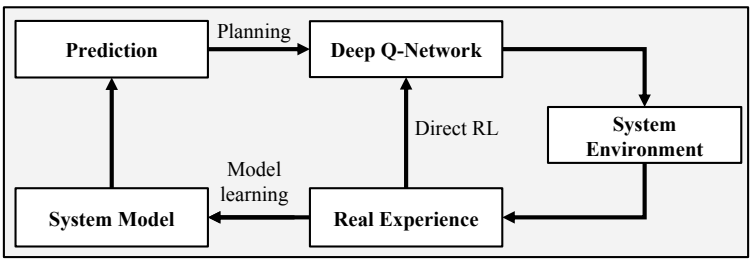


Figure 4.1: Hybrid Learning Architecture.

to demonstrate our agent's performance since they employ RL to optimally orchestrate DL inference at Edge. Our main contributions are:

- A run-time reinforcement learning based orchestration framework for DL inference services, to minimize inference latency within accuracy constraints.
- A hybrid learning approach to accelerate the RL learning process and reduce direct sampling during learning.
- Experimental results demonstrating acceleration of the learning process on an end-edge-cloud platform by up to 166.6× over state-of-the-art *model-free* RL-based orchestration.

4.2 Hybrid Learning Strategy

Hybrid Learning is a combination of *model-free* and *model-based* RL. As illustrated in Figure 4.1, the architecture consists of *System Environment*, *System Model*, and *Deep Q-Network Model*. The training process begins with an initial system model and an initial policy. The agent is trained in three phases viz., ***Direct RL***, ***System Model Learning***, ***Planning***. Algorithm 4 defines the training process, which is composed of three major phases:

(1) Direct RL: In the Direct-RL phase, the agent interacts with the *System Environment* to collect Real Experience for training the Deep Q-Network (DQN) model. Every time the

agent takes a *step*, the *real experience* is pushed into a prioritized replay buffer D_{direct}, and a random replay buffer D_{world}. We sample mini-batches from the buffer D_{direct} and update the DQN by Adam optimizer [140]. Then, we assign new priorities to the prioritized replay buffer D_{direct}.

(2) System Model Learning: We model our system to *Predict* the system's behavior for given pairs of (s_τ,a_τ). *System Model Learning* starts with no assumption about the *System Environment*, and is learned and updated through real experiences. As the agent takes more steps with real experiences, the model ($System(\theta_s)$) learns the system from state-action pairs that have previously been experienced (in Direct RL). $System(s_\tau, a_\tau; \theta_s)$ predicts average response time r_τ and the next state $s_{\tau+1}$. In this phase, we train the model with mini-batches sampled randomly from the buffer D_{world} update the θ_s accordingly.

(3) Planning: During this phase, the agent uses the *System Model* to predict the system's behavior to improve the policy model $Q(s, a; \theta_Q)$. We train $Q(s, a; \theta_Q)$ with the predicted tuples $(s_\tau, a_\tau, r_\tau, s_{\tau+1})$ in a replay buffer D_{plan}. Given a current state s, we use $System(s, a; \theta_s)$ to generate a set A of K actions $(a_i, i = 0, \cdots, K-1)$ that might yield promising rewards r. For each action a_i in the set A, if a_i does not exist in the buffer D_{plan}, agent will take a step a_i at state s to get a reward r and the next state s'. Then we push the tuple (s, a_i, r, s') into the buffer D_{plan}. If the action a_i already exists in buffer D_{plan}, we only update its corresponding current state in the buffer with the new state. Then, we train the policy model $Q(s, a; \theta_Q)$ in the same way as the Direct RL process but with the generated data sampled from the buffer D_{plan}.

We define α as a parameter to control the portion of *Direct RL* and *Planning* during the training policy. Increasing α over time results in decreasing the number of Direct RL during the training. In this strategy, after sufficient real experience, the *System Model* can predict the system behavior. Therefore, the agent relies more on *System Model* prediction rather than real experiences which emphasizes *model-based* RL.

Algorithm 4 Hybrid Learning Algorithm.

Data: ϵ, C, T, N

Result: $Q(s,a;\theta_Q), System(s,a;\theta_s))$

initialize $Q(s,a;\theta_Q)$ and $Q'(s,a;\theta_{Q'})$ with $\theta_{Q'} \leftarrow \theta_Q$;
initialize $System(s,a;\theta_s)$;
initialize replay buffer D_{direct},D_{world},D_{plan};
for ($epoch \leftarrow 1 : N$) {
  $\alpha \leftarrow \frac{epoch}{N}$;
  # Direct Reinforcement Learning ————————;
  for ($session \leftarrow 1 : (1-\frac{\alpha}{2})N_{direct}$) {
    for ($step \leftarrow 1 : T_{direct}$) {
      with probability ϵ, choose random a, otherwise $a \leftarrow \text{argmin}_{a'} Q(s,a';\theta_Q)$;
      $r, s' \leftarrow take\ step(s,a)$;
      store $(s,a,r,s') \rightarrow D_{direct}, D_{world}$;
      $s \leftarrow s'$;
      sample prioritized minibatch $(S,A,R,S') \subset D_{direct}$;
      update θ_Q via Adam on the minibatch;
  update target model $Q'(s,a;\theta_{Q'})$ by $\theta_{Q'} \leftarrow \theta_Q$;
  # System Model ————————————————-;
  for ($session \leftarrow 1 : (1-\frac{\alpha}{2})N_{world}$) {
    sample random minibatch $(S,A,R,S') \subset D_{world}$;
    update θ_S;
  # Planning ————————————————————-;
  for ($session \leftarrow 1 : (\frac{\alpha+1}{2})N_{suggest}$) {
    for ($step \leftarrow 1 : T_{suggest}$) {
      $a \leftarrow \text{argmin}_{a'} System(s,a';\theta_s)$;
      $s' \leftarrow System(s,a;\theta_S)$;
      # Choose best K actions for current state s;
      $A \leftarrow \{a | a \in argsort_{a'}(System(s,a';\theta_s))[0:K]\}$;
      for (a' *in* A) {
        if $a' \notin D_{plan}$ **then**
          $r, s' \leftarrow take\ step(s,a')$;
          store $(s,a',r,s') \rightarrow D_{plan}$;
        else
          update $(s'',a',r,s') \in D_{plan}$ by $s'' \leftarrow s$;
      $s \leftarrow s'$;
  for ($session \leftarrow 1 : (\frac{\alpha+1}{2})N_{plan}$) {
    sample prioritized minibatch from D_{plan};
    update θ_Q via Adam on the minibatch;
  update target model $Q'(s,a;\theta_{Q'})$ by $\theta_{Q'} \leftarrow \theta_Q$;
}

4.3 Evaluation

We demonstrate the efficacy of our hybrid learning strategy for RL-based inference orchestration compared to two state-of-the-art RL-based inference orchestration [66, 53]: AdaDeep [66] employs the DQL algorithm to optimally orchestrate DL model selection based on available resources; AutoScale [53] applies QL algorithm to optimally orchestrate DL inference in end-edge architecture (See Table 4.1). We evaluate the performance of the agent on a multi-user end-edge-cloud framework (see Chapter 3). We also report the overhead incurred by the agent in our learning process, compared with AdaDeep [66] and AutoScale [53]. We described the experimental setup for our proposed framework in Chapter 2.

4.3.1 Agent Performance

We demonstrate our proposed hybrid learning agent's performance in finding optimal orchestration decisions. At design time, we determine the true optimal configuration for orchestrating a DL task under any given condition of workloads, network, and number of active users using a brute force search. This is used for comparing the orchestration decisions made by our proposed approach and DQL against the true optimal configuration. Both Deep-Q Learning (DQL) and Hybrid Learning (HL) algorithms have yielded a 100% prediction accuracy in comparison with the true optimal configuration (See Chapter 2). Thus, RL-based orchestration decisions always converge with the optimal solution.

4.3.2 Training Overhead

We evaluate the agent overhead during the training phase to demonstrate the efficiency of the hybrid learning algorithm in comparison with the state-of-the-art AutoScale [53] and AdaDeep [66]. To identify an optimal policy, we assess the number of steps required to

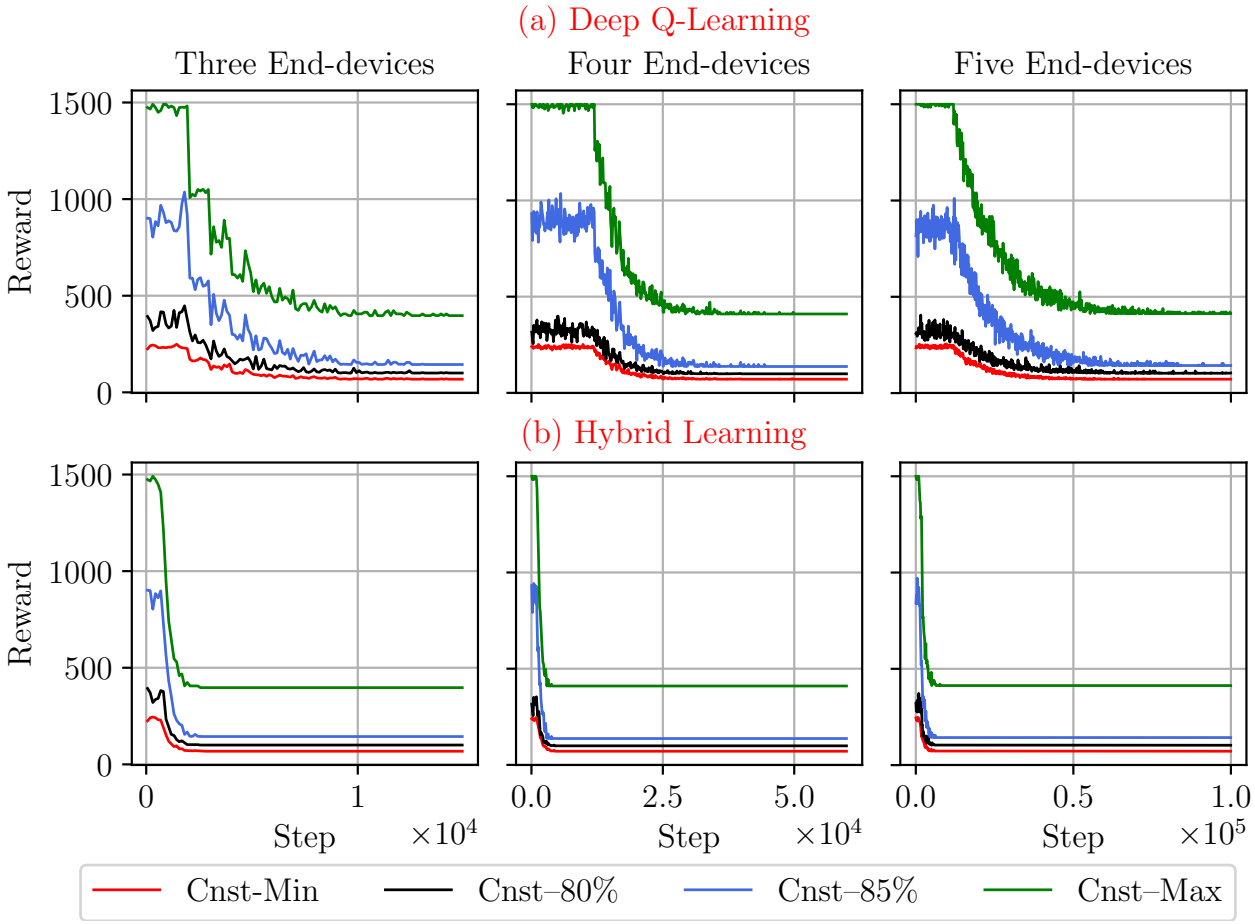


Figure 4.2: Convergence time for up to five users within different constraints. Cnst represents constraint for each experiment. Deep Q-Learning referes to AdaDeep [66] work. The comparison with AutoScale [53] is mentioned in Table 4.2.

interact with the system environment under each approach. Figure 4.2 shows the training phases for different number of users under different accuracy constraints. Each subplot shows the training phase for the system with a different number of users using the DQL and HL algorithms. The agent is trained under different accuracy constraints, which results in converging to different optimal policies for the corresponding constraint (i.e., different converged reward values). Our evaluation shows that the HL algorithm accelerates the training steps up to 11.6× and 166.6× in comparison with AdaDeep [66] and AutoScale [53] respectively. The convergence steps for different number of users are summarized in Table 4.2.

Table 4.2: Training overhead for Hybrid Learning algorithm compared with AdaDeep [66] and AutoScale [53]. Training overhead is presented as number of steps to achieve the optimal policy.

# of Users	Constraint	AutoScale	AdaDeep	Our
3	Min	0.7×10^4	0.1×10^5	0.2×10^4
	80%	0.5×10^4	0.1×10^5	0.2×10^4
	85%	0.3×10^4	0.1×10^5	0.2×10^4
	Max	0.7×10^4	0.1×10^5	0.2×10^4
4	Min	0.9×10^5	0.3×10^5	0.3×10^4
	80%	0.8×10^5	0.4×10^5	0.4×10^4
	85%	0.4×10^5	0.4×10^5	0.3×10^4
	Max	0.9×10^5	0.3×10^5	0.3×10^4
5	Min	0.1×10^7	0.6×10^5	0.6×10^4
	80%	0.1×10^7	0.6×10^5	0.6×10^4
	85%	0.6×10^6	0.7×10^5	0.6×10^4
	Max	0.1×10^7	0.7×10^5	0.6×10^4

Our result shows that the number of agent's interactions with the system environment increases as we increase the dimension of the problem space (increasing number of users). However, our agent with the HL algorithm outperforms the state-of-the-art [53, 66] in the number of interactions with the system environment for the different number of users and under different accuracy constraints. The training time consists of experience time (i.e., time spent in interacting with the system environment to collect data) and computation time for learning the system and policy models. Table 4.3 shows the overall training time for different number of users. Our evaluation shows that the HL algorithm converges up to 7.5× faster in comparison with AdaDeep, and 109.4× faster in comparison with A.. Further, we also present the overhead of the training agent in finding optimal orchestration decisions, with *experience time* and *computation time* metrics. *Experience time* is the total time to execute all taken steps (cost of interaction with the system environment) to identify an optimal policy, while *computation time* is the time to train the agent. The HL algorithm

Table 4.3: Training time (presented in minutes) for different number of users compared with AutoScale [53] and AdaDeep [66]. Comp and Exp represent *Computational Time* and *Experience Time*. *AutoScale employs the QL algorithm which has a low computational overhead.

# of Users	Time (min)	AutoScale*	AdaDeep	Ours
3	Comp	-	1	1
	Exp	1.5×10^2	6.8×10^1	2.6×10^1
	Total	1.5×10^2	6.9×10^1	2.7×10^1
4	Comp	0.3	1.0×10^1	3.6
	Exp	3.7×10^2	1.1×10^2	1.3×10^1
	Total	3.7×10^2	1.2×10^2	1.6×10^1
5	Comp	1.0×10^1	1.5×10^2	3.4×10^1
	Exp	5.8×10^3	1.8×10^2	1.9×10^1
	Total	5.8×10^3	3.3×10^2	5.3×10^1

results in 4.4× and 9.4× speedup in comparison with AdaDeep for *Computation Time* and *Experience Time*, respectively. The computation time per step to train *System Model* and *Policy Model* with the HL algorithm is higher than the DQL and QL algorithms. However, our HL algorithm converges significantly faster than the algorithms (See Table 4.2 and Table 4.3). Therefore, any additional computation cost per step is more than compensated by a significant reduction in the number of interactions to identify an optimal policy.

4.4 Summary

We presented a hybrid learning based framework for orchestrating deep learning tasks in end-edge-cloud architectures. Our proposed hybrid learning strategy requires fewer interactions with the real-time execution runs, converging to an optimal solution significantly faster than state-of-the-art model-free RL approaches. We deployed our proposed framework on enterprise AWS end-edge-cloud system for evaluating MobileNet kernels. Our hybrid learning approach accelerates the training process by up to 166× in comparison with the state-of-the-art RL-based DL inference orchestration, while making optimal orchestration decisions after significantly early convergence. Our future work will explore cross-layer opportunities and more hardware-friendly RL algorithms.

Chapter 5

Hyperdimensional Hybrid Learning on End-Edge-Cloud Networks

5.1 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 [124]. However, this is not a robust solution since network conditions are prone to inconsistencies and affect the real-time of providing these services [52]. To enable a low-latency infrastructure, the introduction of edge computing has brought compute capacity closer to end-user devices [135]. 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 [80].

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 [80]. 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 [137, 131]. RL methods are developed to interact with the environment by trying different actions and reinforcing those that provide higher rewards [29]. 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 [54], 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 [54]. 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 [44], (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 [30, 98]. 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 [44]. HDC abstractly extends this concept as a learning paradigm by modeling and operating on high-dimensional data representations [44].

HDC achieves this by encoding input data into high-dimensional space, outputting what is called *hypervectors* and using well-defined HDC operands to train and execute inference tasks [30]. Each dimension of this hyperspace abstractly models a neuron's functionality in processing external stimuli [44]. The HDC operands are simple arithmetic operations over these dimensions that can be supported on devices with limited resources and computational power [39]. In addition, this class of algorithms is able to accomplish both classification and regression machine learning tasks [31, 89] and have been shown to achieve comparable results to neural networks offering higher learning quality, e.g, faster convergence, learning speed, and computational efficiency [30].

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 [115, 94]. 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 work, we propose HDHL, a novel hyperdimensional hybrid learning model to optimize resource management applications. Our contributions in this work 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 DQL algorithm.
- We also demonstrate HDHL's ability to produce higher quality learning with a faster convergence to the optimal result compared to the DQL 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 work, we tackle the communication-computation co-optimization offloading orchestration problem to an Edge-End-Cloud network. We show that the novel HDHL algorithm is computationally more efficient than DQL 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.

5.2 Hyperdimensional Computing

The concept that motivates Hyperdimensional Computing (HDC) is the theoretical neuroscience observation that the human brain processes information by operating on high-dimensional data representations. This formulates the core idea behind HDC, which is to

map objects into hyperdimensional space by encoding the original data into what is called *hypervectors*, vectors which store thousands of elements. Each dimension of these hypervectors abstractly formulates a neuron's functionality in processing external stimuli [44]. Once an HDC model is trained, the encoded hypervector will have successfully captured the important patterns of its respective class of data across these dimensions. The advantage of working in high-dimensional space is the multitude of nearly orthogonal hypervectors.

HDC utilizes this property to combine hypervectors using well-defined operations, including *bundling* and *association* while maintaining their respective information with high probability. The training process involves encoding the original input data and superimposing them to produce a *model hypervector*, which is the object representation in high-dimensional space. The HDC operation *association* performs the cognitive computation and learning over the encoded data during the training phase to procure highly accurate high-dimensional representations. Given a new data point to classify, the inference algorithm is a similarity search conducted between the model hypervectors and the encoded new data point (e.g., query) to inform the model in its decision-making process. Let us assume $\vec{\mathcal{H}}_1$ and $\vec{\mathcal{H}}_2$ are two hypervectors, $\vec{\mathcal{H}} \in \{0,1\}^d$ and d is 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 cosine similarity. The other HDC operands are described in greater detail below:

Bundling: The component-wise addition operation of hypervectors, where the information from multiple hypervectors are saved into a single hypervector. This formulates the memorization capability of HDC since the bundled *hypervector* is similar to each of the component hypervectors that comprise it i.e., $\delta(\vec{\mathcal{H}}_1 + \vec{\mathcal{H}}_2, \vec{\mathcal{H}}_1) \approx 0$.

Binding: The Binding operation associates items in hyperspace. Given two hypervectors, H_1 and H_2, component-wise multiplication (XOR in binary) denoted as $(*)$ is conducted to produce a new hypervector that is dissimilar to its constituent vectors, i.e., $\delta(\vec{\mathcal{H}}_1 * \vec{\mathcal{H}}_2, \vec{\mathcal{H}}_1) \approx 0$. This is ideal for associating two hypervectors for mapping and variable-value association.

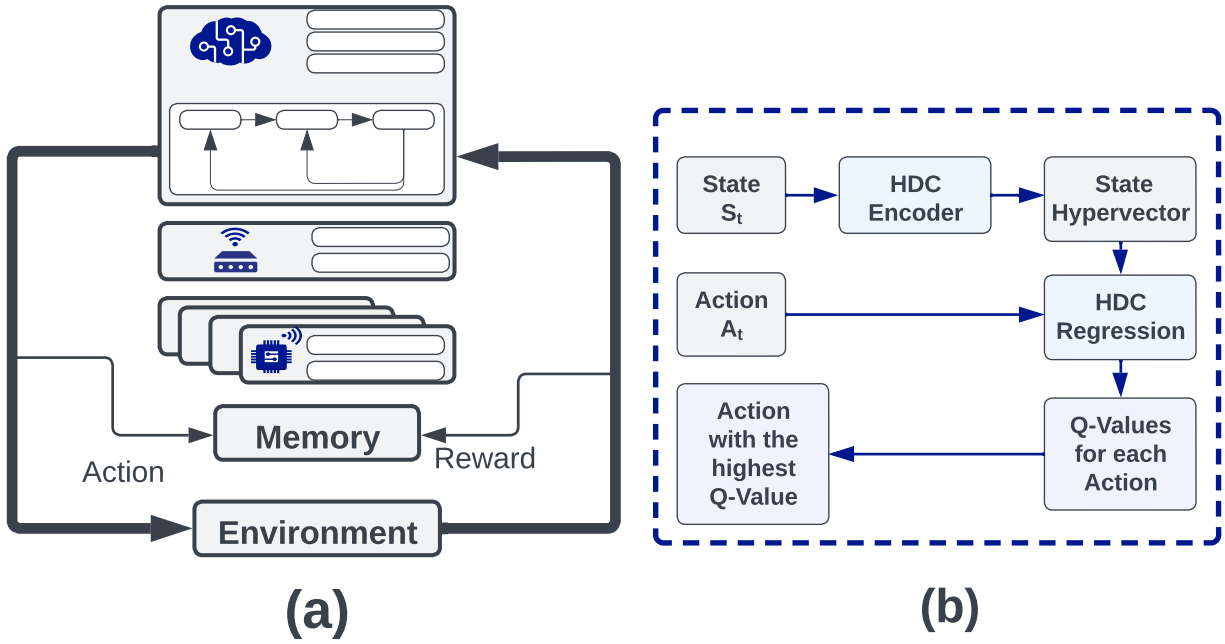


Figure 5.1: Overview of HDHL reinforcement learning.

5.3 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 $\epsilon = 1$ but 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 [34]. 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}. This 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 [13], 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

$$q_{t\_true} = R_t + \gamma max_A \mathcal{Q}'(S_{t+1}, A) \tag{5.1}$$

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 [33]. 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

$$q_{t\_pred} = \mathcal{Q}(S_t, A_t) = \vec{\mathcal{S}}_t^T \cdot \vec{\mathcal{M}}_{A_t} \tag{5.2}$$

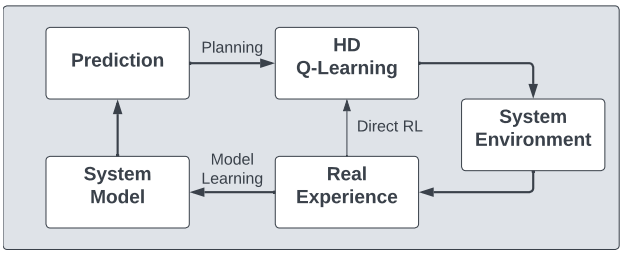


Figure 5.2: Hybrid Learning Architecture.

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

$$\vec{\mathcal{M}}_{A_t} = \vec{\mathcal{M}}_{A_t} + \alpha(q_{t\_true} - q_{t\_pred}) \times \vec{\mathcal{S}}_t \tag{5.3}$$

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

5.4 Hyperdimensional Computing Hybrid Learning

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.

5.4.1 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 device. 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 re-introduce the problem 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.

5.4.2 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 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 the model by the error in predicted response time, R'_t, as shown below

$$M_{A_t} = M_{A_t} + \alpha \times (R_t - R'_t) \times \vec{\mathcal{S}}_t \tag{5.4}$$

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 3.3. 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 similarly 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 correct 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.

5.4.3 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 via 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 together, 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.

5.5 Evaluation

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.

5.5.1 Agent Performance

We demonstrate the proposed agent's performance in finding optimal orchestration decision. 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.

5.5.2 Training Overhead

In this next section, we present the computational efficiency of the hyperdimensional hybrid learning approach for RL-based inference orchestration and compare it to the state-of-the-art DQL algorithm as the comparative baseline [68]. Both Figure 5.3 and Table 5.1 shows the computational efficiency between the baseline DQL and the two HDC-based RL models. Compared to DQL, 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 Hyperdimensional Hybrid Learning 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 DQL and the HDHL models and demonstrate the latter's surpassing performance. These results are presented in Figure 5.4 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 DQL 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.

Table 5.1: Training overhead (presented in number of steps) for different number of users compared with the state-of-the-art [68].

# of Users	**DQL**	**QHD**	**HDHL**
Three	1.2×10^4	5×10^4	0.2×10^4
Four	4.0×10^4	4.5×10^4	0.8×10^4

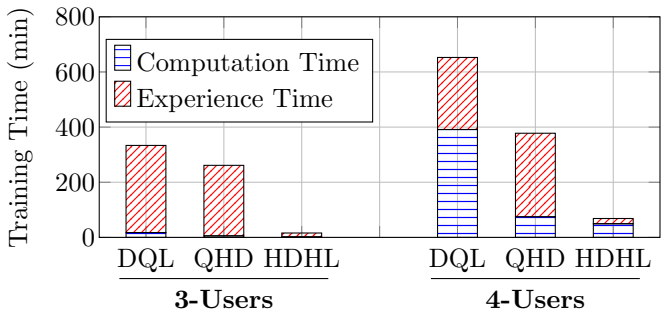


Figure 5.3: Comparison of training times between DQL [68], QHD, and HDHL.

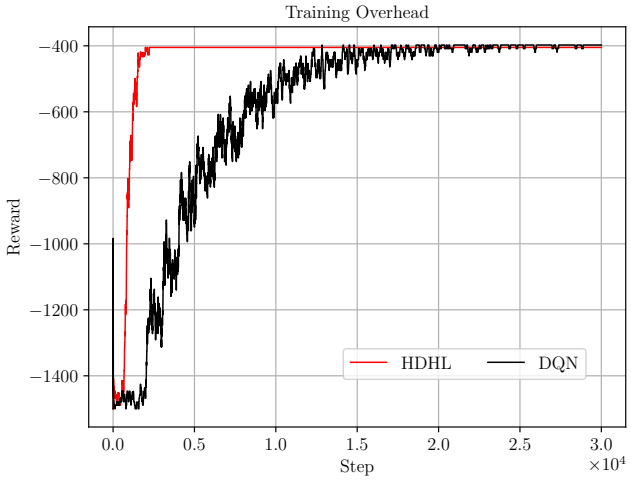


Figure 5.4: Learning Overhead for DQL and HDHL

5.6 Summary

In this chapter, we presented Hyperdimensional Hybrid Learning, the first HDC based hybrid learning algorithm, for orchestrating deep learning tasks in an End-Edge-Cloud architecture. We demonstrated its improved computational efficiency and learning quality compared to the neural network based Q-Learning algorithm. We showed 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 6

IMSER: Intelligent Multimodal Sensing for Energy Efficient and Resilient eHealth Systems

6.1 Introduction

This chapter presents a case study of applying our work in an eHealth system context. eHealth applications provide critical digital healthcare services through continuous monitoring and analysis of multi-modal input data from physiological, and contextual sensors [90]. Smart eHealth applications widely use multi-modal machine learning (MMML) algorithms to analyze input data from different sensor modalities, and generate accurate predictive results [88]. Design of end-to-end smart eHealth applications can be split into two phases viz., *sensing* - acquiring input data from multi-modal sensors, and *sense-making* - analyzing input data through MMML algorithms for predictive results [84]. Efficiency of smart eHealth applications is affected by – i) higher volumes of multi-modal input data from heterogeneous

sensory devices, ii) compute intensity of MMML algorithms for data analysis, and iii) limited energy, and compute resources of wearable sensory devices and mobile computing nodes, particularly at the sensing layer [85].

Further, real-world eHealth applications are designed to monitor patients' symptoms in *everyday settings* through wearable sensors. Typically, continuous longitudinal data acquired in *everyday settings* is prone to higher input data perturbations such as noisy components, unreliable signals, and motion artifacts, in comparison with the data acquired in clinical settings using reliable medical-grade sensors [132]. Processing input data with different perturbations affects the prediction accuracy, and model confidence of MMML algorithms used in eHealth applications [83]. Also, processing non-qualitative data incurs significant performance penalty, and energy drain, with resources spent on un-insightful computations [71]. Input data perturbations originating from the *sensing* phase influences the computational workload, prediction accuracy, model confidence, and energy consumption in the *sense-making* phase. Existing pre-processing, and data filtering techniques monitor the signal quality to selectively sense qualitative input data, and minimize noisy components [101]. Selective sensing aims at reducing the computational effort spent on noisy input data, and consequently reducing the energy consumption of both sensors, and processing elements [95]. State-of-the-art selective sensing techniques use data filtering and compression [101], context-aware sensing [95], and heterogeneity-aware sensing [91] to reduce the total input data volume. Despite the large body of recent work, existing optimizations for end-to-end smart eHealth applications address multi-modal sensing, and sense-making dis-jointly [74, 12, 67].

Improving the system metrics of performance, energy consumption, and prediction accuracy necessitates joint optimization of multi-modal sensing, and sense-making phases. The joint optimization approach allows *adaptive sensing* – to selectively extract reliable, and insightful input data, and features, and *intelligent sense-making* – to choose MMML models

suitable for given input data, and feature set. Together, the adaptive sensing, and intelligent sense-making reduce computational workload, communication penalties, and energy consumption incurred in processing unreliable and/or low-quality data, and improve prediction accuracy, and resilience of eHealth services towards input data perturbations. However, co-optimization of multi-modal sensing, and sense-making phases requires *continuous monitoring* of sensor modalities to detect input data perturbations, *selective feature aggregation* to isolate reliable inputs, and *choice of suitable learning algorithms* for the given input modalities, and feature vectors.

In addition to input data perturbations, varying system environment dynamics such as network connectivity, and signal strength, user mobility, and available energy budget of sensory devices etc., further affect the selection of optimal choices on sensing, and sense-making configuration. Selecting the optimal sensing, and sense-making configuration settings, considering multi-modal input data perturbations, MMML algorithms, and varying system dynamics is an NP-hard problem [111].

Understanding the underlying system dynamics (e.g. network condition), sensing variation (e.g. noisy sensing condition) and intricacies among computation, communication, accuracy, and latency is necessary to find optimal sense-making configuration in eHealth services. Reinforcement learning (RL) is an effective approach to develop such an understanding and interpret the varying dynamics of such systems [92]. RL allows a system to identify complex dynamics between influential system parameters, and make online decisions to optimize objectives such as response time [116]. At the system level, RL has been applied in orchestration of single node IoT devices for eHealth [38], and distributed multi-node edge computing systems [111, 110]. In this work, we propose applying RL for improving the efficiency, and resilience of end-to-end MMML-based smart eHealth services. We present an intelligent multi-modal sensing, and sense-making co-optimization approach, IMSER, to orchestrate eHealth services for improving performance, energy consumption, and prediction accuracy,

and resilience to input data perturbations.

Our novel contributions are as follows:

- A scalable sensor-edge framework for MMML-based smart eHealth applications, capable of monitoring input signal quality, network connectivity, and energy budgets, and detecting input data discrepancies
- Implementation of a reinforcement learning scheme for qualitative feature selection, weighted prioritization of input modalities, input-driven MMML model selection, and sensor configuration setting, based on run-time monitoring, and analysis of input modalities
- Design of an adaptive rule-based controller to enforce the RL agent's intelligent decisions on feature, modality, and model selection, and sensing configuration – for improving performance, energy, and prediction accuracy
- Evaluation of the proposed IMSER framework's efficiency on an exemplar pain assessment eHealth case study.

6.2 Motivation and Significance

In this Section, we present the background on MMML-based eHealth services, challenges in orchestrating smart eHealth services, existing strategies for addressing these challenges, and their limitations, and motivation for our proposed approach.

6.2.1 Motivational Example

MMML-based eHealth services require to meet the following criteria: ***(a) Resiliency*** to recover from any perturbation and provide high quality of experience, ***(b) Energy-efficiency***

Table 6.1: Summary of MMML-based solutions for eHealth services.

Related Works	Selective Sensing	Noise-Awareness	Network Awareness	App Flexibility	Platform Agnostic
[43]	✗	✗	✗	✗	✗
[101, 96, 95, 91]	✓	✗	✗	✗	✗
[3]	✗	✓	✗	✗	✗
AMSER [84]	✓	✓	✗	✓	✗
Ours	✓	✓✓	✓	✓	✓

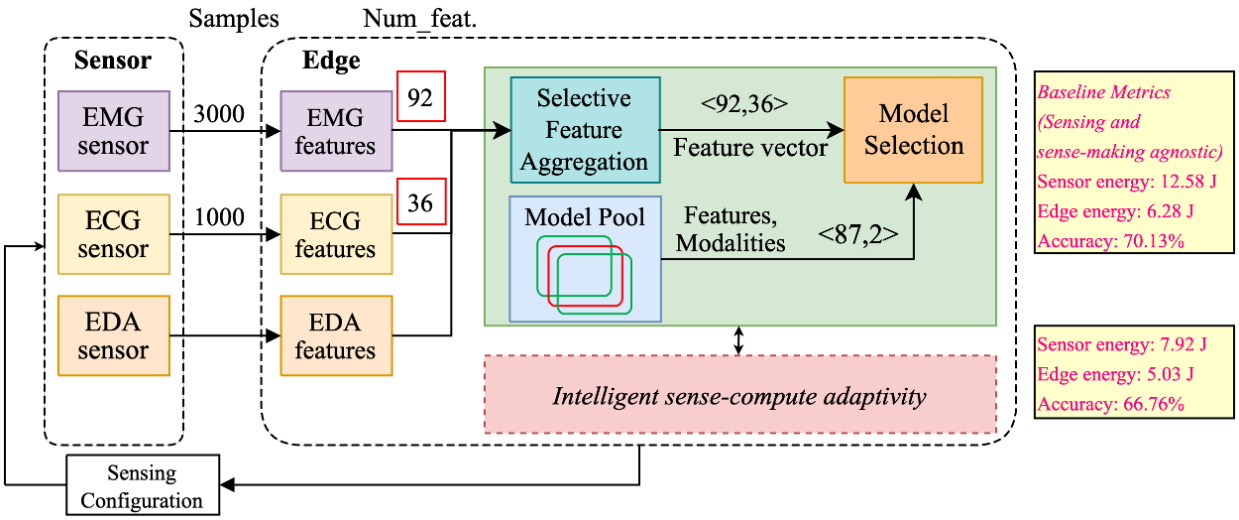


Figure 6.1: Processing with modality selection (selective feature aggregation and model selection) and additional intelligence for adaptive sense-compute

to enable long-term continuous monitoring on battery-powered wearable devices, and ***(c) Performance*** to deliver real-time and accurate services for rapid response to emergency.

To emphasize the importance of resilience challenges in MMML-based eHealth services and the significance of sense-compute adaptivity in addressing those challenges, and for clarification, let us show a motivational exemplar of eHealth monitoring application, pain [127]. Pain monitoring application acquires heterogeneous data from multiple sensory inputs viz., Electromyography (EMG), Electrocardiography (ECG), and Electrodermal Activity (EDA) sensors to capture the autonomic nervous system activity against pain.

The pain monitoring application acquires physiological data from different modalities viz., Electromyography (EMG), Electrocardiography (ECG), and Electrodermal Activity (EDA)

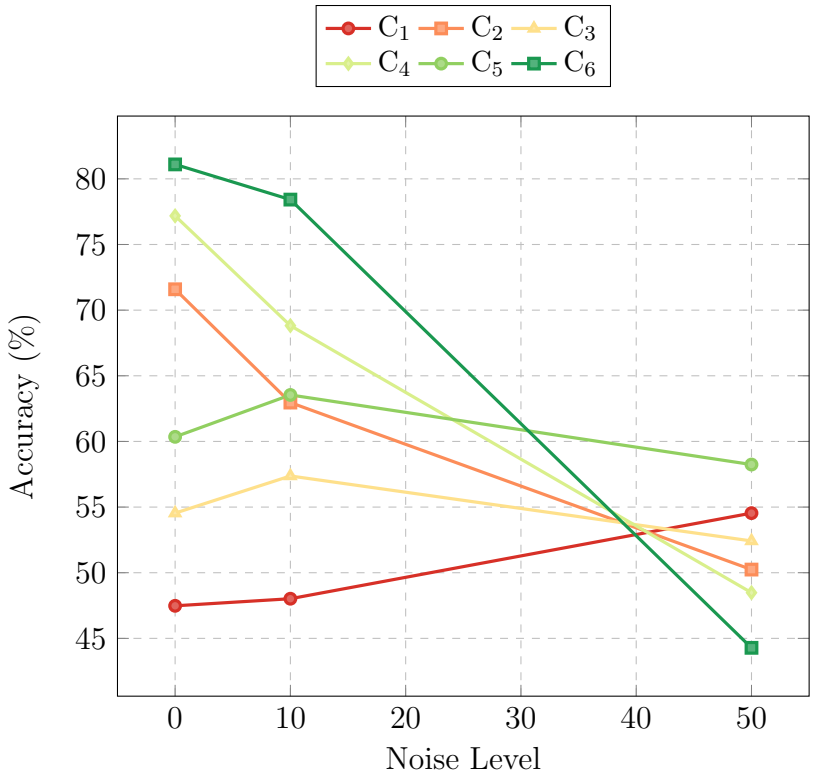
sensors to capture the autonomic nervous system activity against pain. Figure 6.1 shows a complete pipeline of a multi-modal processing, data acquisition from multiple sensors, feature extraction, selective feature aggregation, ML model selection. In this example, EMG, ECG, and EDA sensors are sampled at 500 (6 channels), 500 (2 channels), and 4 Hz (1 channel), respectively. Relevant features from each raw input modality is extracted and using an early fusion technique a single vector of features from all modalities is created. We then predict pain levels using a suitable machine learning models. In practical scenarios, one or more modalities can be distorted due to noises caused by motion artifacts, physical damages, battery shutdown [85]. In this specific example, EDA modality data contains motion artifact type of noise. This figure illustrates the implications of such perturbations on prediction accuracy and energy consumption of sensing and sense-making. The sensing and sense-making agnostic models yield baseline metrics of sensor energy, edge energy, and accuracy of 12.58 J, 6.28 J, and 70.13%, respectively. However, the addition of an intelligent sense-compute adaptivity to the system will result decent improvements on all the performance metrics.

In general, run-time dynamics of the system include connectivity, sensory input data quality, strength of the network, mobility and interaction of a given user to name a few. Such exploration is essential at design-time to achieve optimal configuration for stationary conditions. In reality, an IoT system's behavior dramatically changes over time due to variations in environment and system parameters.

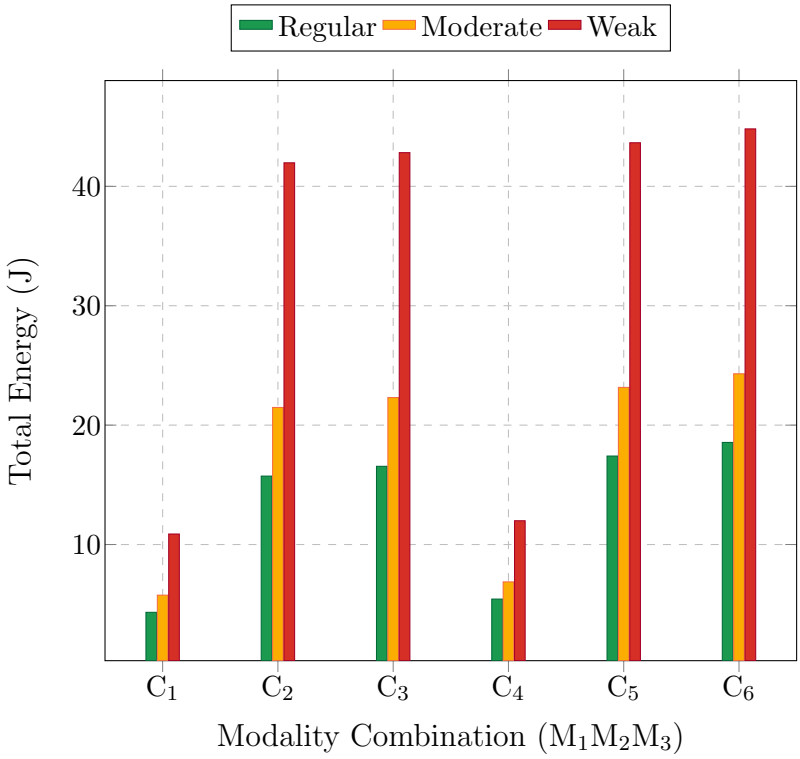
Accuracy We demonstrate the impact of varying MMML inference on performance under different noise levels (NLs). We select between six configuration points with an accuracy between 48.2% and 81.7%. Figure 6.2b shows the accuracy achieved with varying portion of noise. With a single modality, the behavior of the system is different as the noise increases. The accuracy of the inference model when 35% of only one modality is available shows a positive correlation with the noise level, whereas the accuracy when full set of features for only one modality is available is not correlated with the noise level. Moreover, the accuracy

Label	Modality		
	M_1	M_2	M_3
C_1	P	0	0
C_2	F	0	0
C_3	P	P	0
C_4	F	F	0
C_5	P	P	P
C_6	F	F	F

(a) Modality Combinations



(b) Accuracy for various noise levels



(c) Total Energy for various networks

Figure 6.2: Example experimental scenario design points demonstrating effect of (b) noise level on accuracy, (c) network on Total Energy

of the MMML inference is in positive correlation with number of modalities and feature volume when noise is not available. However, as the noise level gets increased the MMML inference accuracy will most likely decrease once the amount of informative features gets corrupted with the noise.

Network. We consider three possible levels of network connections: (i) WiFi connectivity as a low-latency (regular) network that has the signal strength for better connectivity, (ii) 4G as a mid-latency (decent) network, and (iii) 3G as a high-latency (weak) network that has a weaker signal with poor connectivity. Figure 6.2c shows the total energy consumed for the pain monitoring application in different multi-modal volume configurations with all three networks. With a regular network, the energy consumed is lower when 35% of a single modality is available. The energy consumption increases as the reliable modalities are increasing. With a weak network, the energy consumption of the pain monitoring application is higher, as the poor signal strength adds more communication energy. Performance of the MMML inference is independent of the network connection, resulting in lowest response time. This demonstrates the spectrum of energy consumption with different set of reliable modalities, under varying network constraints.

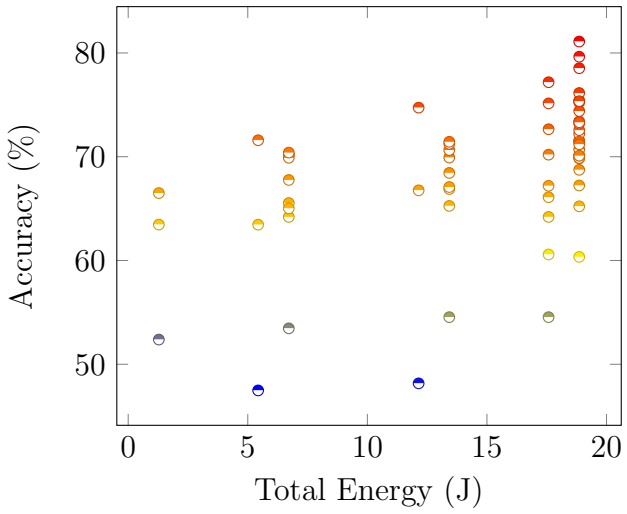


Figure 6.3: Design Space Configuration of accuracy and total energy for various modality combinations

6.2.2 Related Work

State-of-the-art multi-modal applications use selective sensing techniques such as data filtering and compression [101], context-aware sensing [95], and heterogeneity-aware sensing [91] to reduce the total input data volume. Other techniques target model optimization to reduce the compute intensity of sense-making algorithms [3]. This shows these applications address multi-modal sensing and sense-making independently [74, 12, 67]. In addition, existing MMML-based eHealth solutions lacks supporting selective sensing together with noise-awareness, to inspect a modality and decide whether it provides insightful information for the prediction tasks. On the other hand, system environment dynamic significantly affect the sensing and sense-making configuration. For example, the network condition sensors are connected to the edge, needs to be considered to be able to evaluate the influence of data transfer speed and energy consumption on optimal sensing and sense-making configurations. In the following, we illustrate the resilience challenges in MMML-based eHealth services, effect of some environment dynamics on performance and the significance of intelligent and adaptive sensing and control in addressing those challenges through a motivational example of a pain monitoring application [127].

6.2.3 Intelligent Orchestration

Runtime dynamics of the system in addition to requirements and opportunities affect orchestration strategies significantly. Sources of runtime variation across the system stack include connectivity, sensory input data quality, and strength of the network to name a few. Identifying optimal orchestration considering the sensing and sense-making opportunities and requirements in the face of varying system dynamics is a challenging problem. Making the optimal orchestration choice considering these varying dynamics is an NP-hard problem, while brute force search of a large configuration space is impractical for real-time applica-

tions. Understanding the requirements at each level of the system stack and translating them into measurable metrics enables appropriate orchestration decision making. Heuristic, rule-based, and closed-loop feedback control solutions are not efficient until reaching convergence, which requires long periods of time [116]. To address these limitations, RL approaches have been adapted for the joint-optimization decision making [105]. The model-free RL techniques operate with no assumptions about the system's dynamic or consequences of actions required to learn a policy. In other words, Model-free RL builds the policy model based on data collected through trial-and-error learning over epochs [116]. It enables *Platform agnostic* and *Application flexible* feature for RL-based sensing and sense-making frameworks where the agent finds optimal configuration through trial-and-errors during training phase with no assumption on *Platform* and *Application*.

6.2.4 Contributions

The ideal sensing and sense-making configuration provides maximum inference accuracy and minimum energy consumption. The literature lacks a comprehensive solution which is *noise-aware*, *network-aware*, *application flexible* and *platform agnostic* to run a variety of eHealth applications, in particular on multipurpose wearables. The motivational example scenarios presented in Figure 6.1, 6.2, and 6.3 demonstrates the advantages of monitoring input modalities for selective feature aggregation, modality selection, and model selection in presence of dynamics of the environment the importance of an intelligent orchestration. However, this requires intelligent sense-compute adaptivity by continuous monitoring of input modalities, and intelligent control for selective aggregation, modality, and model selection. To this end, we propose an automated framework for intelligent multi-modal sensing to improve resilience and energy efficiency of MMML-based eHealth applications. Our solution presents a joint sensing and sense-making co-optimization using an RL-agent as a general solution for eHealth services considering energy consumption while meeting accuracy requirements.

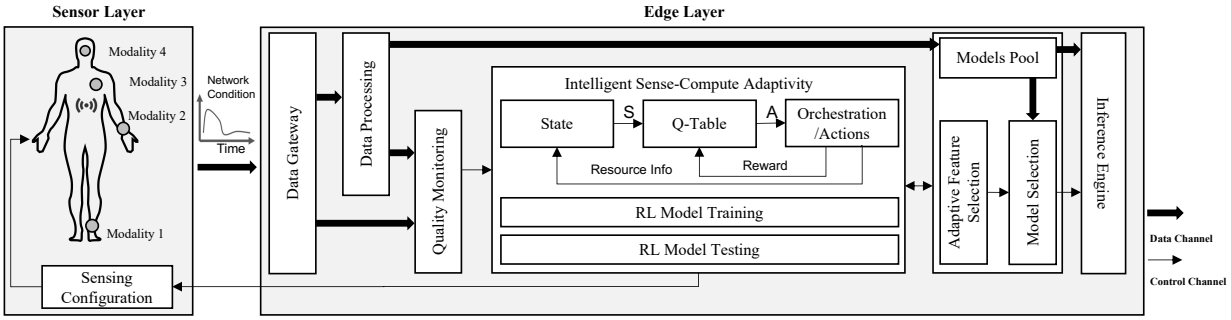


Figure 6.4: System Architecture Overview.

Table 6.1 summarizes and positions our contributions with respect to state-of-the-art.

6.3 Intelligent Multimodal Sensing and Sense-making Framework

We implement our IMSER approach by adopting a widely used generic eHealth system architecture that consists of multiple sensor devices at the sensor layer, and computing resources at the edge layer. Figure 6.4 shows an overview of our proposed framework as a pipeline of sensor, and edge layers for sensing, and sense-making. The *sensor layer* includes the multi-modal sensing capability required for eHealth applications. The edge layer comprises of computing devices, which provide data gateway, data processing, quality monitoring, intelligent orchestration, and adaptive control over feature, and model selection, and inference at the edge. We detail each level of the framework below:

Data Gateway receives multi-modal raw input data from multiple sensors. The physiological signals can potentially contain noise in different levels, and portions of the data window segment from zero (noise-free) to 100% (highly noisy).

Data Processing performs pre-processing and feature extraction, using the data collected from multi-modal sensors. *Pre-processing* sub-module consists of synchronization of multi-

modal signals and labels, filtering sensor modality inputs to produce clean signals, and additional components for affective signal processing (ASP) pipeline (e.g., peak detection, normalization). The ASP pipeline includes input modalities such as EMG, ECG, and EDA, which are required for feature extraction in typical eHealth applications (eg., pain monitoring and stress monitoring). *Feature Extraction* sub-module extracts insightful and informative values from the pre-processed signals. We extract handcrafted features in time domain (mean, standard deviation, RMS), and frequency domain (e.g. power spectral density, median frequency, central frequency) among the automatic features using a variational autoencoder for all modalities [6]. This process facilitates subsequent learning, leading to better interpretation for quality assessment module.

Quality Assessment handles quality assurance of the data, and features to be used in the inference engine. This module observes the system parameters, disruptive events, data quality, and control flow to analyze the situation and context. *Quality Assessment* module assesses signal and its extracted features quality by monitoring key parameters from the sensing phase to identify events and triggers for joint optimization of sensing and sense-making. For example, a trigger at the sensing phase could be any disruptive event in input data of a specific modality (e.g., motion artifacts or sensor detachment).

Intelligent Sense-Compute Adaptivity analyzes inputs from the quality assessment module to intelligently communicate with the adaptive controller to select features and modalities, and configure sensing and computation models. We store pre-trained models for different combinations of signal modalities and aggregated feature vectors in a pool of models. The *Intelligent Sense-Compute Adaptivity* acts as an Reinforcement Learning (RL) agent to holistically and dynamically control the quality of sensing and sense-making as described in the following.

6.3.1 Reinforcement Learning Agent

Reinforcement learning (RL) is widely used to automate intelligent decision making based on experience. Information collected over time is processed to formulate a policy which is based on a set of rules. Each rule consists of three major components viz., (a) state, (b) action, and (c) reward. Among the various RL algorithms [116], Q-learning has low execution overhead, which makes it a perfect candidate for runtime invocation. Figure 6.4 depicts high-level block diagram for our agent. The RL agent is invoked at runtime for intelligent orchestration decisions. Our agent is composed as follows:

State Space: Our state vector is composed of Modality availability, Feature volume per each modality, Network condition, and noise level. Table 6.2 shows the discrete values for each component of the state. Modality availability (represented as MA) is a binary value that states our framework either collects sensory data for that modality or is idle. Feature volume (FM) states what percentage of the feature volume for each modality is incorporated for the further analysis after data processing. FM is considered as discrete value between 0 to 100%. Network condition (represented as NC) shows how the sensors are connected to the edge device. We consider three different network connections which demonstrates different data transfer speed and power consumption. Noise level (represented as NL) states what percentage of each modality is noisy. The state vector at time step τ is defined as follows:

$$S_\tau = \{FM_1, FM_2, FM_3, MA_1, MA_2, MA_3, NL, NC\} \tag{6.1}$$

Action Space: The action vector consists of increase/decrease feature volume for each modality. In other words, the agent takes an action to change feature volume of only one of modalities at each time step. The action vector at time step τ is defined as follows:

$$A_\tau = \{A_1, A_2, A_3\} \tag{6.2}$$

Table 6.2: State Discrete Values

State	Discrete Values	Description
FM_i	0,35%,70%,100%	Modality Feature Volume
MA_i	0,1	Modality Availability
NL	0, 20%, 50%	Modality Noise Percentage
NC	Regular, Moderate, Weak	Network Condition

Reward Function: The reward function is defined as the negative total energy consumption including edge and sensor devices. In our case, the agent seeks to minimize the energy consumption. To ensure the agent minimizes the energy consumption while satisfying the accuracy constraint, the reward R is calculated as follows:

$$\begin{aligned} &\text{if } \overline{Accuracy} > \text{constraint:} \\ &\qquad R_\tau \leftarrow -Energy \\ &\text{else:} \\ &\qquad R_\tau \leftarrow -Max\ Energy \end{aligned} \tag{6.3}$$

To apply the accuracy constraint, the minimum possible reward is assigned when the accuracy threshold is violated. On the other hand, when the selected action satisfies the accuracy constraint, the reward is negative energy consumption in that time step. Algorithm 5 defines our agent's logic with the epsilon-greedy Q-Learning:

Line Description

3: First the agent determines the current system state from the resource monitors.

4-8: Either the state-action pair with the highest Q-value is identified to choose the next action to take, or a random action is selected with probability ϵ.

9-10: The selected action is applied and normal execution resumes. The reward R_τ for the execution period is calculated based on measured consumed energy.

Algorithm 5 Q-Learning Algorithm

1: **while** system is on **do**
2: **From *Resource Monitoring*:**
$S_\tau \leftarrow$ State at step τ
3: **if** $RAND < \epsilon$ **then**
4: Choose random action A_τ
5: **else**
6: Choose action A_τ with largest $Q(S_\tau, A_\tau)$
7: **end if**
8: Monitor total energy consumption
9: Calculate reward R_τ
10: **From *Resource Monitoring*:**
$S_{\tau+1} \leftarrow$ State at step $\tau + 1$
11: Choose action $A_{\tau+1}$ with the largest $Q(S_{\tau+1}, A_{\tau+1})$
12: **To *Updating Qtable*:**
$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[R_\tau + \gamma Q(S_{\tau+1}, A_{\tau+1}) - Q(S_\tau, A_\tau)]$
13: $S_\tau \leftarrow S_{\tau+1}$
14: **end while**

11-12: Based on the resource monitors, the new state $A_{\tau+1}$ is identified, along with the state-action pair with highest Q-value.

13: The Q-value of the previous state-action pair is updated.

14: The current state is updated, and the loop continues.

6.4 Evaluation

We evaluate the proposed intelligent sense-computing framework through a case study of pain assessment [82, 5]. Continuous monitoring of multiple sensor modalities including ECG, EMG, PPG, and EDA signals is required for this case study application to detect pain levels. Our platform for evaluation comprises a sensory node - to collect physiological input modalities of ECG, EMG, PPG, and EDA from the subject, and an edge node - to control sensing and computation and execute the inference. The RL agent's goal is to minimize average response time while satisfying the accuracy and energy constraint. This

enforces quality control by imposing a strict threshold on the average DL model accuracy. The proposed method is implemented on an ODROID-XU3 with an octa-core Exynos processor as the edge device. In the following, we detail the case study and evaluate the case study applications' accuracy, energy efficiency, and performance using our IMSER proposed framework through different scenarios with different levels of noisy input window segments and network strength for all modality. Further, we compare our IMSER approach against AMSER and state-of-the-art multi-modal pain monitoring application that use classification of physiological signals [42] as the baseline ideal scenario.

6.4.1 Hyper-parameter Tuning

An RL agent has a number of hyper-parameters that impact its effectiveness (e.g., learning rate, epsilon, discount factor, and decay rate). The ideal values of parameters depend on the problem complexity, which in our case scales with the number of modalities, noise level, and network condition. In order to determine the learning rate and discount factor, we evaluated values between 0 and 1 for each hyper-parameters. We observed that a higher learning rate converges faster to the optimal, meaning the more the reward is reflected to the Q-values, better the agent works. We also observed that a higher discount factor is better. This means that the consecutive actions have a strong relationship, so that giving more weight to the rewards in the near future improves the convergence time. The selected configuration for hyper-parameters is as follows: $\alpha = 0.99, \gamma = 0.7$, decay rate $= 0.1, \epsilon_{initial} = 0.1$

6.4.2 Case Study and Scenarios

We validate and demonstrate the efficacy of our IMSER approach through an exemplar case study from the affective computing domain, pain assessment, under seven unique scenarios

with varying environment conditions.

Case Study

State-of-the-art pain assessment tools analyze changes in physiological signals indicating different pain levels. In this work, we evaluate IMSER via various unimodal and multimodal pain assessment pre-trained models. These models are trained using iHurt Pain DB, a multimodal dataset from postoperative patients in hospital from 20 patients [81]. Data is collected from face-, chest-, and wrist-worn devices via an eight-channel biopotential acquisition system for EMG and ECG recordings and Empatica E4 for PPG and EDA recordings. We used 500 Hz ECG and EMG, 64 Hz PPG, and 4 Hz EDA sensor modalities during our experiments. We pre-processed each modality and segmented them into 60 second windows. We augmented the raw input training data with real-life noises such as BW and MA at various portions like 20% and 50% of data, to handle potential noisy components in physiological signals for the prediction model.

Then, we extract a combination of time domain, frequency domain and automatic features from this window segment and trained using unimodal and multimodal ML algorithms.

Table 6.3: Experimental Scenarios. Each scenario represent the noise level out of 100 for each modality and the network condition.

Scenarios		**Noise Level**	**Network**
S1	Scenario 1	0	Regular
S2	Scenario 2	20	Regular
S3	Scenario 3	20	Moderate
S4	Scenario 4	20	Weak
S5	Scenario 5	50	Regular
S6	Scenario 6	50	Moderate
S7	Scenario 7	50	Weak

Scenarios

In this work, we conduct experiments under seven unique scenarios with varying noise levels and network conditions for our case study. The experimental scenarios are summarized in Table 6.3, representing a combination of noise level among with a regular, moderate and weak network signal strength. The regular network has no transmission delay, while we add 10ms and 20ms delay to all outgoing packets to emulate the moderate and weak connection behavior, respectively. Putting together the noise level and the network condition creates unique experimental scenarios. Scenario 1 is the ideal baseline case with no noise and a regular network, Scenario 2 has signal components with 20% noise level among with a regular network condition. Scenario 3 and 4 are similar to scenario 2 in terms of noise level percentage but with moderate and weak network condition, respectively. Scenario 5 has modalities with 50% noise level with a regular network condition. Scenario 6 and 7 are similar to scenario 5 in terms of noise level percentage but with moderate and weak network condition, respectively. The proposed IMSER framework has the capability to choose the right configuration with a constraint on accuracy. We define Min and Max accuracy constraints empirically. We set Min to be the median of the accuracy of all the potential decisions. Within the presence of such constraint, IMSER will choose the configuration that provides an accuracy with the smallest number greater than Min. On the other hand, Max refers to the situation when there is no constraint on accuracy and IMSER will try to maximize the accuracy value. In Scenario 1, there is no noise component and network condition is regular, and thus no modality selection or feature selection is applied. Scenario 2 comprises of 20% level of noise in addition to the original data for all modalities within a regular network. In this scenario when there is a Min accuracy constraint, our proposed IMSER approach drops ECG modality and selects 66% of the features from EMG modality, reducing the total number of features from the original feature vector, while it will not drop any modality and select the most significant features from ECG modality when there is no constraint on accuracy. Scenario 3 and 4 demonstrates

Table 6.4: Decision made by AMSER and IMSER during each scenario.

Method	Modality	Scenarios													
		S1		S2		S3		S4		S5		S6		S7	
		Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
AMSER	ECG	✓	✓	✓	✓	N/A	N/A	N/A	N/A	✓	✓	N/A	N/A	N/A	N/A
	EMG	✓	✓	✓	✓	N/A	N/A	N/A	N/A	✗	✗	N/A	N/A	N/A	N/A
	EDA	✓	✓	✓	✓	N/A	N/A	N/A	N/A	✓	✓	N/A	N/A	N/A	N/A
	Feat Vol	100%	100%	63.8%	63.8%	N/A	N/A	N/A	N/A	31.3%	31.3%	N/A	N/A	N/A	N/A
IMSER	ECG	✓	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
	EMG	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	EDA	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓
	Feat Vol	100%	100%	55.6%	77.7%	55.6%	77.7%	55.6%	77.7%	33.3%	55.6%	33.3%	55.6%	33.3%	55.6%

that network condition does not have any effect on prediction model accuracy. Scenario 5, comprises of 50% noise level added to the original data for all modalities within a regular network. In this scenario, when there is a Min accuracy constraint, the EDA modality is completely unreliable. Thus, IMSER will drop the EDA modality and select 33.3% of the features from other two modalities, reducing the total number of features from original feature vector, while it will not drop any feature or modality when there is no constraint on accuracy. Scenario 6 and 7 replicate the same scenario in a moderate and weak network condition, respectively.

We evaluate the accuracy and energy efficiency of the pain monitoring application (which estimates the pain intensity using 3 physiological signals ECG, EMG, and EDA) using the iHurt Pain DB [81]. We trained different models for each levels of pain intensities, varying in the level of noise, network condition and types of modalities combined for the decision-making system. After pre-processing, we extract a set of unique features from each modality viz., 52 features of ECG, 152 features of EMG, and 42 features of EDA [82, 5].

6.4.3 Accuracy Evaluation

We compare the accuracy achieved by the model in each scenario with noisy components (S2-S7) against the baseline ideal scenario $S1$ with no intelligent and adaptive modality and feature selection. Figure 6.5 shows the accuracy (in %) for pain assessment case study under

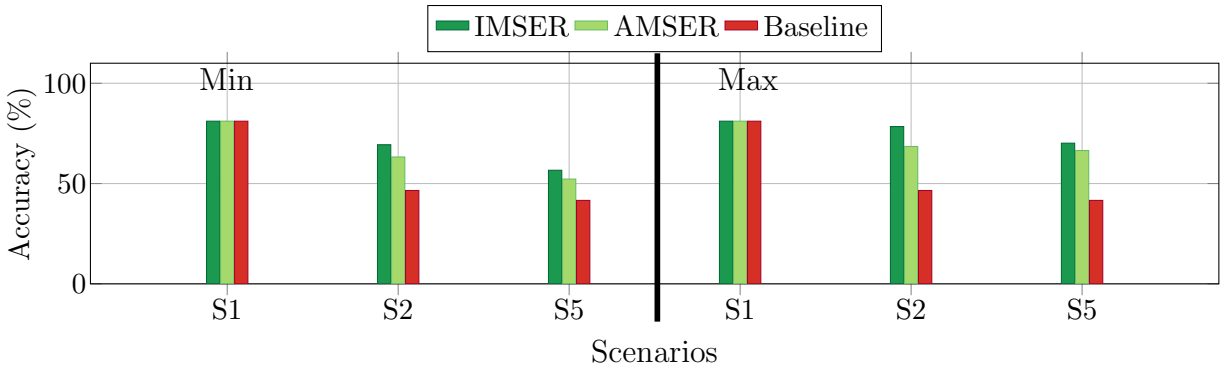


Figure 6.5: Accuracy analysis for IMSER vs. AMSER vs. Baseline [42] for Pain application.

different scenarios. Our proposed intelligent multimodal sense-compute technique provides an improved accuracy in each of the scenarios with a maximum gain of 23% and 32% in S2 with Min and Max accuracy constraint for pain assessment. In the presence of a lower level of noise i.e., 20%, our proposed IMSER approach drops ECG and utilizes the EMG with selected features and drops features from only ECG modality to meet the Min and Max accuracy constraints, instead of holding the entire feature sets of the noisy modalities. This approach leads to a 23% and 32% accuracy improvement in scenario $S2$ with Min and Max accuracy constraint for pain assessment application in comparison with the baseline model which uses all the features from the noisy modalities. With a higher level of noise i.e., 50%, in scenario $S5$, our proposed IMSER approach leads to 15% and 28% accuracy improvement for Min and Max accuracy constrains, respectively, compared to the baseline which uses all the modalities.

6.4.4 Efficiency and Performance Evaluation

We evaluate the efficiency and performance of IMSER in comparison with the baseline. Figure 6.6 shows the energy efficiency improvement during seven scenarios for edge and sensor devices. We report the results for Pain application with Min and Max accuracy constraint. In Scenario $S4$, the energy gain to meet Min accuracy constraint is 1.36× and 1.33× higher

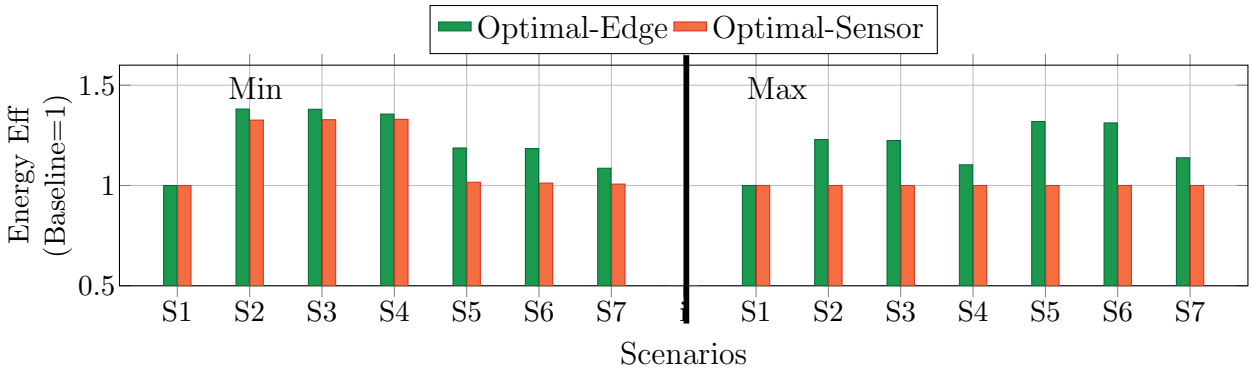


Figure 6.6: Energy efficiency analysis of the edge and sensor device for IMSER and AMSER vs. Baseline [42] for Pain application.

than the baseline for the edge and sensors, respectively. In this case, IMSER provides 22.82% better accuracy by dropping the noisy modality (ECG) and selecting informative feature from EMG modality (See Figure 6.5 and Table 6.4).

Figure 6.7 (a) shows the performance gains at the edge device for the pain assessment application using our proposed IMSER approach, as compared to the baseline Scenario $S1$. In Scenarios $S2$-$S7$, our proposed IMSER approach reduces the number of features and/or drops input data from *noisy* modalities. This lowers the total computational effort, leading to an increase in performance, as compared to the baseline. Figure 6.7 (b) shows the data volume transferred between the sensory devices and edge node using our proposed IMSER approach, in comparison with the baseline. Our IMSER approach intelligently selects features and/or drops *noisy* modalities, which reduces the total input data volume significantly. For instance, in Scenario $S4$, one noisy modality is entirely dropped and another modality contribute with two third of its most informative features, leading to more than $6\times$ reduction in transmitted data volume. Such reduction in data volume to be transmitted further reduces the communication latency and energy expense incurred in transmitting noisy data.

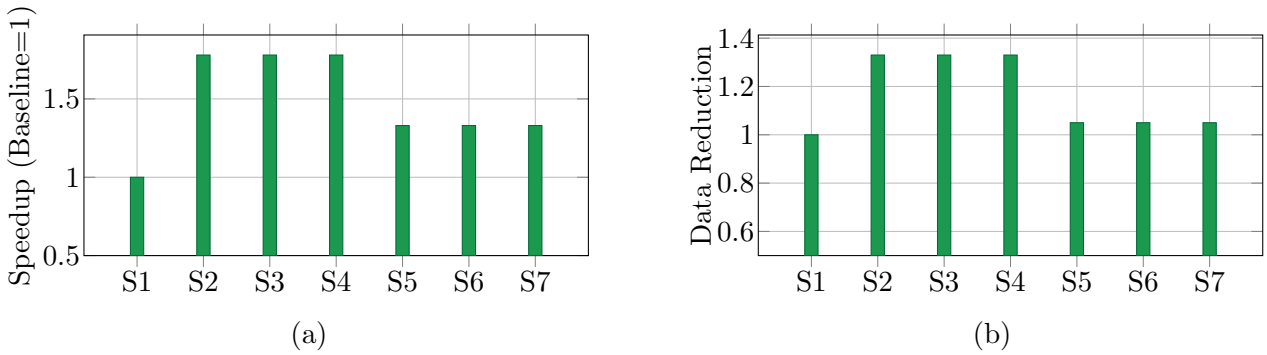


Figure 6.7: (a) Performance analysis of the edge device for IMSER vs. Baseline [42] for Pain application. (b) Data volume transferred between the sensors and edge for IMSER vs. Baseline.

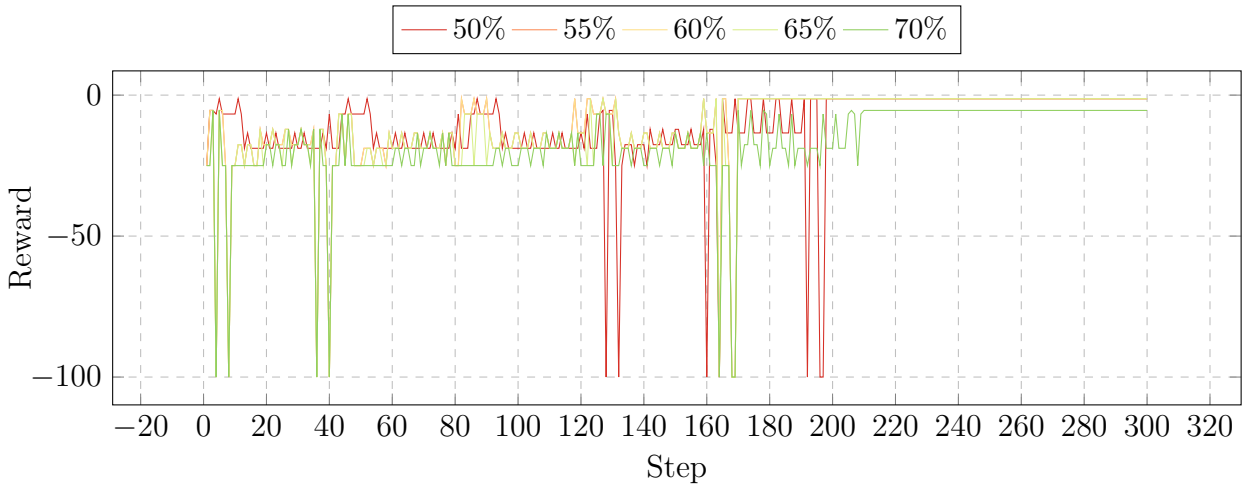


Figure 6.8: Training overhead for Q-Learning algorithm under different accuracy constraints

6.4.5 Overhead Analysis

Exploration Overhead: We evaluate the time required by the proposed agent for the training phase to identify an optimal policy. Figure 6.8 shows the training phase under different accuracy constraints using Q-Learning algorithm. This shows that when training a model from scratch, the reward converges after about about 170 inference runs on average. However, increasing the accuracy constraint leads to a more complex problem and therefore increase in convergence time.

Runtime Overhead: To demonstrate the viability of mobile inference deployment, we eval-

uate the IMSER runtime overhead. The performance overhead of RL algorithm in IMSER is, on average, 20 μs for training, excluding the time for inference execution. It corresponds to 1.2% of the lowest inference latency. In addition, when using the trained Q-table, the overhead can be reduced to 7.3 μs with only 0.3% overhead. This result means it takes 18.1 μs to measure the inference results, calculate the reward, and update the Q-table. The energy overhead is only 1% and 0.2% of the total system energy consumption, when training the Q-table and exploiting the trained Q-table, respectively.

6.5 Summary

This chapter proposed IMSER, a novel RL-based intelligent and adaptive, multi-modal sensing framework for energy efficient and resilient eHealth applications. IMSER guides sensing and sense-making by monitoring input signal and feature quality via an intelligent sense-compute adaptive framework that reduces non-informative data to improve energy efficiency while meeting an accuracy constraint. We demonstrated effectiveness of the IMSER framework with an exemplar eHealth application of pain assessment, achieving up to 1.8× and 1.4× performance and energy gains respectively, compared to the baseline with no optimization, while achieving up to 32% better accuracy. Our approach shows that RL-based frameworks can be deployed effectively for joint sense-compute adaptivity, and opens the door for further research devising more intelligent resource management and computation-offloading policies for adaptive control and co-optimization of both sensing and sense-making.

Chapter 7

Conclusion and Future Works

This work showed Deep Learning inference orchestration considering cross-layer optimization presents an opportunity to improve performance while maintaining accuracy. This thesis proposed an online learning framework that coordinates tradeoffs between application and system layers to enhance performance of Deep Learning inference in end-edge-cloud systems. The reinforcement learning-based framework integrated model optimization techniques and computation offloading to achieve the minimum average response time for DL inference services while meeting accuracy constraints. The approach achieved up to a 35% improvement in average response time with less than a 0.9% decrease in accuracy when compared to state-of-the-arts. Additionally, we proposed a hybrid learning-based framework that optimized Deep Learning inference in end-edge-cloud architectures. The approach converged to an optimal solution faster than state-of-the-art model-free RL approaches and accelerated the training process by up to 166x. This thesis presented Hyperdimensional Hybrid Learning, the first HDC-based hybrid learning algorithm for orchestrating Deep Learning inference in an end-edge-cloud architecture. The HDC approach achieved a 21.0× speedup in total training time and required 4.8× fewer direct interactions with the system environment than the neural network-based Q-Learning algorithm.

Furthermore, this thesis applied the proposed approaches in the healthcare domain and introduced IMSER, an RL-based intelligent and adaptive, multi-modal sensing framework for energy-efficient and resilient eHealth applications. IMSER monitors input signal and feature quality through an intelligent sense-compute adaptive framework to reduce non-informative data and improve energy efficiency while maintaining accuracy. This work demonstrated the effectiveness of IMSER using a pain assessment eHealth application and achieved up to 1.8× and 1.4× performance and energy gains, respectively, compared to the baseline with no optimization while achieving up to 32% better accuracy. The case study proved that online learning techniques can be deployed effectively for joint sense-compute adaptivity, and open the door for further research devising more intelligent resource management for adaptive control and co-optimization of both sensing and sense-making.

Open research directions are suggested as follows:

Holistic Cross-layer Optimization Collaborative sensor-edge-cloud architectures can benefit from orchestration techniques that improve several key metrics, including performance, turnaround time, energy efficiency, accuracy, and network utilization. This work presented orchestration techniques that utilize an RL-based agent to make intelligent compute placement, offloading, and model selection decisions. However, to further enhance efficiency and performance, more investigation is needed into cross-layer system parameters. Holistic orchestration that considers opportunities and constraints across all layers, including the user, device, application, network, edge node, and platform layers, can maximize efficiency. Thus, investigating holistic cross-layer optimization is a promising future research direction.

Advanced RL strategies In our investigation, this work found that dealing with high-dimensional problems involves significant interactions with the environment. However, these interactions are not feasible in distributed computer systems because executing each configuration is costly, leading to higher latency and resource consumption during the learning

process. To provide sample and computational efficiency, this work utilized Q-Learning, Deep Q-learning, Hybrid Learning, and Hyperdimensional RL. Despite the efforts, the problem space becomes even more complex with holistic cross-layer orchestration, requiring more sophisticated RL algorithms for efficient learning processes. Therefore, it is worth exploring state-of-the-art RL algorithms for orchestration problems as a future research direction.

Online Learning for DL Distributed Training To conduct data-parallel distributed training, it is essential for each worker to possess comparable computational capabilities. Otherwise, the straggler problem may arise, wherein machines with higher performance complete the local update faster than other machines in each iteration. Consequently, the aggregation process must wait for all workers to finish before aggregating and updating the weights, which exacerbates the straggler problem. The straggler problem is not limited to normal distributed GPU training but is also applicable to data-parallel distributed training. As a result of the straggler problem, companies or research groups with a small-scale cluster of different GPU types may be unable to fully utilize their machines. This issue is not unique to these scenarios and is also prevalent in other heterogeneous environments and homogeneous systems that have unstable communication. To address the straggler issue, online learning can be used to dynamically adjust the workload of each worker by monitoring the dynamic environment and resource availability. This indicates that investigating online learning for distributed training would be a worthwhile research direction for the future.

Sense-Compute Co-optimization Cross-layered sense-compute co-optimization is the most effective strategy for improving sensor dependant, edge-based ML applications. In this chapter, we presented adaptive sensing and sensing-aware computing techniques that uses system-wide monitoring and intelligence for sense-compute co-optimization. This approach focuses on selecting appropriate ML models based on quality of input modalities, exploiting the inherent resilience of multi-modal ML applications. Adaptive feature selection, and ML model selection enforce the idea of sense-compute co-optimization at a coarse-grained level,

and require multiple pre-trained models. Incorporating fine-grained edge-layer ML model configurations such as early exit, greedy feature selection, neural network model attention, and saliency maps etc., can complement the model selection strategy. The feasibility of such edge-layer based ML model tuning in collaboration with cloud-layer based model selection is another open research direction.

Bibliography

[1] 2019 speedtest u.s. mobile performance report by ookla.

[2] M. G. R. Alam, M. M. Hassan, M. Z. Uddin, A. Almogren, and G. Fortino. Autonomic computation offloading in mobile edge for iot applications. *Future Generation Computer Systems*, 90:149–157, 2019.

[3] D. Amiri et al. Context-aware sensing via dynamic programming for edge-assisted wearable systems. *ACM HEALTH*, 2020.

[4] P. A. Apostolopoulos, E. E. Tsiropoulou, and S. Papavassiliou. Risk-aware data offloading in multi-server multi-access edge computing environment. *IEEE/ACM Transactions on Networking*, 28(3):1405–1418, 2020.

[5] Aqajari et al. Pain assessment tool with electrodermal activity for postoperative patients: Method validation study. *JMU*, 2021.

[6] Aqajari et al. pyeda: An open-source python toolkit for pre-processing and feature extraction of electrodermal activity. *ANT*, 2021.

[7] L. Atzori et al. The internet of things: A survey. *Computer networks*, 54(15), 2010.

[8] I. Azimi, A. Anzanpour, A. M. Rahmani, T. Pahikkala, M. Levorato, P. Liljeberg, and N. Dutt. Hich: Hierarchical fog-assisted computing architecture for healthcare iot. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):174, 2017.

[9] I. Azimi et al. Empowering healthcare iot systems with hierarchical edge-based deep learning. In *IEEE/ACM CHASE*, 2019.

[10] W. Bao, W. Li, F. C. Delicato, P. F. Pires, D. Yuan, B. B. Zhou, and A. Y. Zomaya. Cost-effective processing in fog-integrated internet of things ecosystems. In *Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems*, pages 99–108, 2017.

[11] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *2013 Proceedings Ieee Infocom*, pages 1285–1293. IEEE, 2013.

[12] M. Baruah and B. Banerjee. Modality selection for classification on time-series data. In *MileTS*, 2020.

[13] 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.

[14] J. Bi, H. Yuan, S. Duanmu, M. Zhou, and A. Abusorrah. Energy-optimized partial computation offloading in mobile-edge computing with genetic simulated-annealing-based particle swarm optimization. *IEEE Internet of Things Journal*, 8(5):3774–3785, 2020.

[15] X. Cao, F. Wang, J. Xu, R. Zhang, and S. Cui. Joint computation and communication cooperation for mobile edge computing. In *2018 16th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, pages 1–6. IEEE, 2018.

[16] N. Cardwell, S. Savage, and T. Anderson. Modeling tcp latency. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 3, pages 1742–1751. IEEE, 2000.

[17] V. Chamola, C.-K. Tham, and G. S. Chalapathi. Latency aware mobile task assignment and load balancing for edge cloudlets. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 587–592. IEEE, 2017.

[18] Z. Chang, Z. Zhou, T. Ristaniemi, and Z. Niu. Energy efficient optimization for computation offloading in fog computing system. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[19] M.-H. Chen, B. Liang, and M. Dong. Joint offloading and resource allocation for computation and communication in mobile cloud with computing access point. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[20] S. Chen, Q. Li, M. Zhou, and A. Abusorrah. Recent advances in collaborative scheduling of computing tasks in an edge computing paradigm. *Sensors*, 21(3):779, 2021.

[21] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis. Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal*, 6(3):4005–4018, 2018.

[22] Z. Chen and X. Wang. Decentralized computation offloading for multi-user mobile edge computing: A deep reinforcement learning approach. *arXiv preprint arXiv:1812.07394*, 2018.

[23] B. Cheng, Z. Zhang, and D. Liu. Dynamic computation offloading based on deep reinforcement learning. In *12th EAI International Conference on Mobile Multimedia Communications, Mobimedia 2019*. European Alliance for Innovation (EAI), 2019.

[24] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[25] CISCO. *Internet of Things At a Glance*, 2016. https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf.

[26] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[27] A. E. Eshratifar, M. S. Abrishami, and M. Pedram. Jointdnn: an efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Transactions on Mobile Computing*, 2019.

[28] F. Firouzi et al. Internet-of-Things and big data for smarter healthcare: From device to architecture, applications and analytics. *FGCS*, 2018.

[29] 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.

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

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

[32] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

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

[34] 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.

[35] H. Hoffmann et al. SEEC: A Framework for Self-aware Computing. Technical report, MIT, 2010.

[36] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[37] X.-L. Huang, X. Ma, and F. Hu. Machine learning and intelligent communications. *Mobile Networks and Applications*, 23(1):68–70, 2018.

[38] Y. Huang, R. Cao, and A. Rahmani. Reinforcement learning for sepsis treatment: A continuous action space solution. *ML4HC*, 2022.

[39] 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.

[40] 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. IEEE, 2022.

[41] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 401–411, 2018.

[42] M. Kächele et al. Multimodal data fusion for person-independent, continuous estimation of pain intensity. In *EANN*, 2015.

[43] M. Kächele et al. Multimodal data fusion for person-independent, continuous estimation of pain intensity. In *International Conference on Engineering Applications of Neural Networks*, pages 275–285. Springer, 2015.

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

[45] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[46] Y.-H. Kao, B. Krishnamachari, M.-R. Ra, and F. Bai. Hermes: Latency optimal task assignment for resource-constrained mobile computing. *IEEE Transactions on Mobile Computing*, 16(11):3056–3069, 2017.

[47] A. Kattepur, H. Dohare, V. Mushunuri, H. K. Rath, and A. Simha. Resource constrained offloading in fog computing. In *Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets*, pages 1–6, 2016.

[48] H. Ke, J. Wang, H. Wang, and Y. Ge. Joint optimization of data offloading and resource allocation with renewable energy aware for iot devices: A deep reinforcement learning approach. *IEEE Access*, 7:179349–179363, 2019.

[49] K. Khalil, K. Elgazzar, and M. Bayoumi. A comparative analysis on resource discovery protocols for the internet of things. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2018.

[50] S. Khalili and O. Simeone. Inter-layer per-mobile optimization of cloud mobile computing: a message-passing approach. *Transactions on Emerging Telecommunications Technologies*, 27(6):814–827, 2016.

[51] 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, 2018.

[52] 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.

[53] 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. IEEE, 2020.

[54] 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.

[55] Y. G. Kim and C.-J. Wu. Autofl: Enabling heterogeneity-aware energy efficient federated learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–198, 2021.

[56] A. Kouloumpris, M. K. Michael, and T. Theocharides. Reliability-aware task allocation latency optimization in edge computing. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 200–203. IEEE, 2019.

[57] A. Kouloumpris, T. Theocharides, and M. K. Michael. Metis: Optimal task allocation framework for the edge/hub/cloud paradigm. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, pages 128–133, 2019.

[58] J. Kwak, Y. Kim, J. Lee, and S. Chong. Dream: Dynamic resource and task allocation for energy minimization in mobile cloud systems. *IEEE Journal on Selected Areas in Communications*, 33(12):2510–2523, 2015.

[59] J. Li, H. Gao, T. Lv, and Y. Lu. Deep reinforcement learning based computation offloading and resource allocation for mec. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2018.

[60] B. Lin et al. Computation offloading strategy based on deep reinforcement learning for connected and autonomous vehicle in vehicular edge computing. *Journal of Cloud Computing*, 2021.

[61] L. Lin, X. Liao, H. Jin, and P. Li. Computation offloading toward edge computing. *Proceedings of the IEEE*, 107(8):1584–1607, 2019.

[62] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[63] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief. Delay-optimal computation task scheduling for mobile-edge computing systems. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 1451–1455. IEEE, 2016.

[64] L. Liu, Z. Chang, and X. Guo. Socially aware dynamic computation offloading scheme for fog computing system with energy harvesting devices. *IEEE Internet of Things Journal*, 5(3):1869–1879, 2018.

[65] L. Liu, Z. Chang, X. Guo, S. Mao, and T. Ristaniemi. Multiobjective optimization for computation offloading in fog computing. *IEEE Internet of Things Journal*, 5(1):283–294, 2017.

[66] S. Liu, J. Du, K. Nan, A. Wang, Y. Lin, et al. Adadeep: A usage-driven, automated deep model compression framework for enabling ubiquitous intelligent mobiles. *arXiv preprint arXiv:2006.04432*, 2020.

[67] S. Liu et al. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *International Conference on Mobile Systems, Applications, and Services*, 2018.

[68] 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.

[69] A. Ltd. Ip products: Arm nn.

[70] H. Lu, C. Gu, F. Luo, W. Ding, and X. Liu. Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning. *Future Generation Computer Systems*, 102:847–861, 2020.

[71] G. M. et al. A comprehensive survey on multimodal medical signals fusion for smart healthcare systems. *Information Fusion*, 2021.

[72] P. Mach and Z. Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[73] Y. Mao, J. Zhang, S. Song, and K. B. Letaief. Power-delay tradeoff in multi-user mobile-edge computing systems. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[74] Masinelli et al. Self-aware machine learning for multimodal workload monitoring during manual labor on edge wearable sensors. *Design&Test*, 2020.

[75] B. McDanel, S. Teerapittayanon, and H. Kung. Embedded binarized neural networks. *arXiv preprint arXiv:1709.02260*, 2017.

[76] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang. Learning-based computation offloading for iot devices with energy harvesting. *IEEE Transactions on Vehicular Technology*, 68(2):1930–1941, 2019.

[77] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[78] S. S. Mousavi, M. Schukat, and E. Howley. Deep reinforcement learning: an overview. In *Proceedings of SAI Intelligent Systems Conference*, pages 426–440. Springer, 2016.

[79] 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.

[80] 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.

[81] Naeini et al. Prospective study evaluating a pain assessment tool in a postoperative environment: Protocol for algorithm testing and enhancement. *JRP*, 2020.

[82] Naeini et al. Pain recognition with electrocardiographic features in postoperative patients: Method validation study. *JMIR*, 2021.

[83] E. K. Naeini et al. A real-time ppg quality assessment approach for healthcare internet-of-things. *ANT*, 2019.

[84] E. K. Naeini et al. Amser: Adaptive multimodal sensing for energy efficient and resilient ehealth systems. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1455–1460. IEEE, 2022.

[85] E. K. Naeini, S. Shahhosseini, A. Subramanian, T. Yin, A. M. Rahmani, and N. Dutt. An edge-assisted and smart system for real-time pain monitoring. In *2019 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 47–52. IEEE, 2019.

[86] M. Nakhkash et al. Analysis of Performance and Energy Consumption of Wearable Devices and Mobile Gateways in IoT Applications. In *COINS*, 2019.

[87] Y. Nan, W. Li, W. Bao, F. C. Delicato, P. F. Pires, and A. Y. Zomaya. A dynamic tradeoff data processing framework for delay-sensitive applications in cloud of things systems. *Journal of Parallel and Distributed Computing*, 112:53–66, 2018.

[88] J. Ngiam et al. Multimodal deep learning. In *ICML*, 2011.

[89] 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.

[90] C. O'Keeffe et al. Role of ambulance response times in the survival of patients with out-of-hospital cardiac arrest. *EMJ*, 2011.

[91] S. M. Oteafy. A framework for heterogeneous sensing in big sensed data. In *GLOBECOM*, 2016.

[92] J. Park, S. Samarakoon, M. Bennis, and M. Debbah. Wireless network intelligence at the edge. *Proceedings of the IEEE*, 107(11):2204–2239, 2019.

[93] B. Peng et al. Deep dyna-q: Integrating planning for task-completion dialogue policy learning. *arXiv preprint arXiv:1801.06176*, 2018.

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

[95] C. Perera et al. Context aware computing for the internet of things: A survey. *IEEE communications surveys & tutorials*, 2013.

[96] B. Pourghebleh et al. Data aggregation mechanisms in the internet of things: A systematic review of the literature and recommendations for future research. *Journal of Network and Computer Applications*, 2017.

[97] G. Qiao, S. Leng, and Y. Zhang. Online learning and optimization for computation offloading in d2d edge computing and networks. *Mobile Networks and Applications*, pages 1–12, 2019.

[98] 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.

[99] A. Rahmani et al. *Fog Computing in the Internet of Things - Intelligence at the Edge*. Springer, 2017.

[100] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1421–1429. IEEE, 2018.

[101] M. Rani et al. A systematic review of compressive sensing: Concepts, implementations and applications. *IEEE Access*, 2018.

[102] J. Ren, G. Yu, Y. Cai, and Y. He. Latency optimization for resource allocation in mobile-edge computation offloading. *IEEE Transactions on Wireless Communications*, 17(8):5506–5519, 2018.

[103] R. Roman et al. A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, 78, 2018.

[104] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[105] T. Sen and H. Shen. Machine learning based timeliness-guaranteed and energy-efficient task assignment in edge computing systems. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10. IEEE, 2019.

[106] D. Seo, S. Shahhosseini, M. A. Mehrabadi, B. Donyanavard, S.-S. Lim, A. M. Rahmani, and N. Dutt. Dynamic ifogsim: A framework for full-stack simulation of dynamic resource management in iot systems. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6. IEEE.

[107] S. Shahhosseini, A. Anzanpour, I. Azimi, S. Labbaf, D. Seo, S.-S. Lim, P. Liljeberg, N. Dutt, and A. M. Rahmani. Exploring computation offloading in iot systems. *Information Systems*, page 101860, 2021.

[108] S. Shahhosseini, A. Anzanpour, I. Azimi, S. Labbaf, D. Seo, S.-S. Lim, P. Liljeberg, N. Dutt, and A. M. Rahmani. Exploring computation offloading in iot systems. *Information Systems*, 107:101860, 2022.

[109] S. Shahhosseini, I. Azimi, A. Anzanpour, A. Jantsch, P. Liljeberg, N. Dutt, and A. M. Rahmani. Dynamic computation migration at the edge: Is there an optimal choice? In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 519–524. ACM, 2019.

[110] S. Shahhosseini, T. Hu, D. Seo, A. Kanduri, B. Donyanavard, A. M. Rahmani, and N. Dutt. Hybrid learning for orchestrating deep learning inference in multi-user edge-cloud networks. *arXiv preprint arXiv:2202.11098*, 2022.

[111] S. Shahhosseini, D. Seo, A. Kanduri, T. Hu, S.-S. Lim, B. Donyanavard, A. M. Rahmani, and N. Dutt. Online learning for orchestration of inference in multi-user end-edge-cloud networks. *ACM Transactions on Embedded Computing Systems (TECS)*.

[112] Z. Sheng, C. Mahapatra, V. C. Leung, M. Chen, and P. K. Sahu. Energy efficient cooperative computing in mobile wireless sensor networks. *IEEE Transactions on Cloud Computing*, 6(1):114–126, 2015.

[113] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner. Optimized iot service placement in the fog. *Service Oriented Computing and Applications*, 11(4):427–443, 2017.

[114] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*. 1990.

[115] 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.

[116] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[117] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[118] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang. Adaptive deep learning model selection on embedded systems. *ACM SIGPLAN Notices*, 53(6):31–43, 2018.

[119] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339. IEEE, 2017.

[120] I. A. *et al.* Hich: Hierarchical fog-assisted computing architecture for healthcare iot. *ACM Transactions on Embedded Computing Systems*, 16(5), 2017.

[121] B. Varghese and R. Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.

[122] J. Wang et al. Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning. *Communications Magazine*, 2019.

[123] X. Wang, Y. Han, V. C. 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.

[124] 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.

[125] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li. Mobile-edge computing: Partial computation offloading using dynamic voltage scaling. *IEEE Transactions on Communications*, 64(10):4268–4282, 2016.

[126] Z. Wei, B. Zhao, J. Su, and X. Lu. Dynamic edge computation offloading for internet of things with energy harvesting: A learning method. *IEEE Internet of Things Journal*, 6(3):4436–4447, 2018.

[127] P. Werner et al. Automatic recognition methods supporting pain assessment: A survey. *IEEE Transactions on Affective Computing*, 2019.

[128] P. Winder. *Reinforcement Learning*. O'Reilly Media, 2020.

[129] J. Xu and S. Ren. Online learning for offloading and autoscaling in renewable-powered mobile edge computing. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[130] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu. Deepwear: Adaptive local offloading for on-wearable deep learning. *IEEE Transactions on Mobile Computing*, 19(2):314–330, 2019.

[131] 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.

[132] G. Yang et al. A health-iot platform based on the integration of intelligent packaging, unobtrusive bio-sensor, and intelligent medicine box. *IEEE Transactions on Industrial Informatics*, 2014.

[133] C. You and K. Huang. Exploiting non-causal cpu-state information for energy-efficient mobile cooperative computing. *IEEE Transactions on Wireless Communications*, 17(6):4104–4117, 2018.

[134] 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. 2018.

[135] 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.

[136] H. Yuan and M. Zhou. Profit-maximized collaborative computation offloading and resource allocation in distributed cloud and edge computing systems. *IEEE Transactions on Automation Science and Engineering*, 2020.

[137] 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.

[138] W. Zhan et al. Deep-reinforcement-learning-based offloading scheduling for vehicular edge computing. *Internet of Things Journal*, 2020.

[139] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, and Y. Zhang. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE access*, 4:5896–5907, 2016.

[140] Z. Zhang. Improved adam optimizer for deep neural networks. In *IWQoS*, 2018.

[141] X. Zhao, L. Zhao, and K. Liang. An energy consumption oriented offloading algorithm for fog computing. In *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*, pages 293–301. Springer, 2016.