Decentralized Cache-aided Offloading in Edge Cloud collaborative environment using Deep Q Reinforcement Learning

by

Vishnu Priya Guddeti

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Many emerging applications such as augmented reality, facial recognition, autonomous cars, and e-health require heavy computation, and the processed results have to be available to the user in the order of milliseconds. Edge computing combined with cloud computing can address this challenge by distributing the load (offloading) on different connected computing resources. However, effective task offloading requires an efficient resource management framework. Many existing offloading methodologies consider only latency and energy consumption in a pre-defined network configuration implemented on a small scale. In addition, the effect of the location of the offloading algorithm has not been extensively studied. This thesis introduces a novel adaptive offloading framework using Online Deep Q reinforcement learning. The proposed framework considers strict latency constraints, high state space, rapidly changing user mobility, heterogeneous resources, and stochastic task arrival rate. The proposed research also highlights the importance of caching and introduces a novel concept called "container caching" that caches the dependencies of popular applications. Therefore, offloading decisions are taken to minimize energy consumption, latency, and caching costs. Moreover, the significance of deployment location of the offloading algorithm is also reviewed, and a distributed offloading method is proposed. Extensive simulations in a discrete event simulator implemented in Java using realistic profiles of tasks have been conducted. Simulation results and comparisons with existing benchmarking algorithms showed remarkable performance in terms of energy consumption, network traffic, task failures, remaining power on a large scale demonstrated the feasibility of the proposed approach.

Acknowledgements

I would like to express my sincere gratitude to my Supervisor, “Prof. Mohamed Atia” for his support throughout my graduate studies. I am very thankful for his valuable and constructive suggestions while formulating the research problem and methodology. His patience, moral support, and encouragement made a significant contribution to the progression of research.

I would like to thank Ericsson for providing fellowship in carrying out the research. I would like to extend my thanks to Sandra Ballarte, and Aroosh Elahi for their support throughout my Masters.

My heartfelt thanks to my parents for their support and encouragement throughout my studies. My masters journey would not have been possible without their guidance. I would also like to thank my friend Anurag who helped me in stimulating discussions, gave continuous moral support, and encouraged me throughout my course. Finally, I would like to thank my friend Madhupreeta who supported me throughout my stay at Canada and provided happy distractions to relax my mind outside the academics.

Table of Contents

List of Tables

List of Illustrations

List of Acronyms

Acronym	Definition
DQRL	Deep Q Reinforcement Learning
IoT	Internet of Things
RL	Reinforcement Learning
RAM	Random Access Memory
MI	Million Instructions
5G	Fifth Generation
MEC	Mobile Edge Computing
ETSI	European Telecommunications Standard Institute
AR	Augmented Reality
VR	Virtual Reality
DL	Deep learning
NN	Neural Network
MDP	Markov Decision Process
QL	Q-Learning
DRL	Deep Reinforcement Learning
RSU	Road-Side Unit
TDMA	Time Division Multiple Access
DNN	Deep Neural Network
MIPS	Million Instructions per second
FDT	Fuzzy Decision Tree
IaaS	Infrastructure as a Service
SaaS	Software as a Service
WLAN	Wireless Local Area Network
WAN	Wide Area Network
SDN	Software Defined Networking

Chapter 1: Introduction

With the rapid development in technology, there has been a tremendous increase in the number of mobiles, laptops, sensors, and electronic gadgets in the last decade. The increase in gadgets can be related to the proliferation of sophisticated IoT applications such as facial recognition, smart health, AR/VR gaming, etc. The data used and processed by these IoT applications [1] is very high compared to the traditional applications such as social media and web-browsing. For instance, industrial automation [2] involves frequent interaction of robots with humans and a range of tasks such as sign recognition, data analysis, speech to text conversion to be carried out to fulfill the functionalities. The available real-time data needs to be collected, analyzed, and processed in the order of milliseconds in these scenarios. A similar situation can be witnessed in smart cities [3] for traffic monitoring, pedestrian detection, sign recognition. Any random movement detection in pedestrians has to be detected accurately, and traffic signs must be changed accordingly. Any delay or failure to detect the movements may lead to severe safety consequences. In Virtual Reality gaming [4], rendering 72 frames per second is the minimum requirement for the smooth running of the application. The user movements need to be accurately captured and processed for an immersive experience. Many other IoT applications such as patient monitoring, biometric recognition, wireless-inventory trackers, smart home require quick data processing. Also, the requirements of all these applications differ based on the purpose. The advent of 5G [5] has turned the existing IoT applications complicated with stringent requirements. Therefore, these applications need a large amount of data to be processed in a short time interval.

Moreover, it incurs ample energy to run these applications in local devices. The limited battery power, storage capacity, and processing speed available in the user device limit the execution of applications locally. This data is usually sent to a central cloud for processing to ensure the quality of service to the user. Central cloud [6] is a data center equipped with large storage space and high processing speed. It hosts various virtual machines to execute applications parallelly and decrease the waiting time of tasks. However, these datacenters are usually located far from the user, and the delay in communicating with the cloud for task execution is high [7]. This may lead to violation of service-level agreements [8] for applications. Also, any disruption in part of the network may cause a global outage interrupting all ongoing network operations.

This led to the emergence of edge computing [9] where servers are located closer to the user. Edge servers are mini-datacenters with limited resources available at the edge of the user [10]. As edge servers act as intermediate layers between user devices and the cloud, selected task requests directed to the cloud can be handled by the edge server. This reduces the load on the central cloud and lessens the burden on network resources to send task information. Also, these geographically distributed edge servers are more suitable for monitoring the real-time data [11] and easing the processing restrictions. Therefore, edge computing combined with cloud computing can serve several users effectively and ensure fast and interactive responses.

1.1 Motivation

The massive data generated by various IoT applications such as connected cars, smart factories, and wearables must be processed, and the users' results must be rolled out. At times, the processing is less and can be done locally. However, due to local devices' limited battery power and computation capacity, this data needs to be transmitted to the edge server or remote cloud for further processing. On the other hand, the resources at edge server are also limited compared to the central cloud due to the cost and viability factors. Also, some complex applications require huge storage space and computation power that may not be available at edge servers and these tasks need to be redirected to cloud. Sending several tasks for cloud computation leads to an increase in delay and high bandwidth utilization for transmitting the data. Therefore, the edge cloud offloading [12] method is challenging [13] due to many factors. Primarily, the frequent mobility of users may lead to frequent disconnections to edge servers; therefore, local and cloud computing is a viable option in this scenario. Secondarily, the heterogeneity in availability of resources such as storage space, battery, computation power at local, edge server, and at cloud requires us to choose the location based on the task type. The available bandwidth, signal power, user density varies rapidly based on the location and available devices in the vicinity. It is not easy to maintain the network's performance due to the fluctuating resources. Therefore, it is crucial to identify a proper offloading location based on the task's network dynamics and requirements.

There is extensive research [14] [15] [16] going on in this field because of the high complexity and numerous variables. The exploration frameworks in this domain can be broadly classified into two types as presented below:

1. Research on finding an ideal algorithm and optimal scheme to choose an offloading location for all the generated tasks and allocate resources to available data centers.
2. Devising a mathematical model to replicate a network comprised of user devices, edge servers, and cloud. Then, ideal locations to place edge servers are identified such that end-to-end service delay in delivering the applications is minimized.

Choosing the placement of edge servers happens initially during the network setup, and the algorithm is used eventually during the expansion of the network. However, the offloading location to execute a task needs to be chosen every time tasks are generated from user devices. This is a scalable approach and can be effective for dynamic environments.

Edge servers and remote clouds usually host a set of virtual machines to enable the parallel execution of tasks. The presence of virtual machines [17] is usually ignored in the existing literature. It is assumed that computation capacity can be allocated for every task, and the delay is ignored. Also, the battery power of a user device should be utilized effectively. Ignoring the battery power and using the user device dedicatedly for computation may cause the device to die, depreciating the quality of service. Storing regularly used data in RAM is usually done in many electronic devices to improve performance and easily access

the data. The same concept can be utilized in this problem, and the most popular application data can be cached to reduce the computations further and save energy.

The concept of caching [18] is not extensively studied by many researchers. Even if considered, it is usually assumed that input and output data of tasks are stored in the memory. However, in exceptional scenarios, identical data is queried. However, the applications used for various tasks usually match, and therefore, it makes more sense to cache the task dependencies rather than the task. It costs a lot of bandwidth and energy to download setup files for computing a task every time. Therefore, this research considers caching software dependencies. Though caching data can incur storage costs, selecting popular tasks can outweigh storage costs. Therefore, it is also important to decide whose software dependencies need to be cached. Therefore, this thesis aims to find an ideal algorithm such that resources are properly utilized, delay requirements are met, tasks are computed, and delivered to users efficiently such that energy consumption, latency, and caching cost are minimized. Also, deploying the algorithm in a central cloud requires frequent network information communication and costs huge bandwidth. Therefore, the significance of deployment location is reviewed, and an ideal deployment location is suggested for the proposed algorithm.

1.2 Contribution

In this thesis, a novel Deep Q Reinforcement Learning algorithm is proposed to choose the offloading locations of numerous tasks generated from various users. The proposed

algorithm considers the task dependencies such as allowed delay, task length, request size and the network resources such as available bandwidth, computation power for executing a task. Moreover, the ability to self-learn the dynamic changes in a network enabled the algorithm to overcome the strict latency constraints, rapidly changing user mobility, and fluctuating available bandwidths. Furthermore, decentralized container-based caching is suggested to utilize the advantage of caching for further reducing the stringent latency requirements of upcoming IoT applications. The recommended approach led to significant reduction in incurred delays, energy consumption, and network traffic.

In this thesis, we consider a discrete simulation environment where tasks of type augmented reality, facial recognition, e-health, infotainment are generated from static and mobile users. The user devices are of type smartphone, laptop, raspberry pi, and sensor. Random tasks of mentioned application types are generated after unequal time intervals. Various virtual machines are created at the edge server and remote cloud to resemble a real environment. Generated tasks can be computed locally or at a virtual machine present in the edge server or remote cloud. A virtual machine amongst several machines at edge server and remote cloud is selected for execution based on the waiting time. Except for the sensor, all other user devices in the considered setup can accommodate local processing. The processing locations of all the generated tasks are chosen based on Online Deep Q Reinforcement learning [19]. It is assumed that edge servers are co-located with base stations for modeling the communication delay. Therefore, edge server for every user will be selected based on the coverage area. At any instance, if the device is under the coverage area of multiple edge servers, then the data rate [20] is chosen to decide a corresponding

edge server for a user. All tasks decided to be computed at the cloud are processed through the edge server. The proposed framework optimizes the energy and latency and considers the advantage of caching to manage the resources effectively. The importance of battery power is valued more for user devices. Therefore, the primary contributions of the proposed research are outlined below:

- A collaborative edge cloud environment is established with tasks arriving from static as well as mobile devices such as smartphones, laptops, single-board computers, and sensors. Network and available resources are updated simultaneously to demonstrate a realistic environment.
- To further reflect a realistic environment, application containers are considered. A container is defined as software package with all dependencies to run a task and should be downloaded before executing any application.
- Generated tasks are of type augmented reality, e-health, and infotainment, and the parameters include permitted latency, length, container, request, and result size.
- Online Deep Q reinforcement learning is applied to identify where to offload a task based on the limited resources available at the user device and edge server and the popularity of a task.
- Proposed container-based caching is reviewed concerning task caching, and the benefits are illustrated using Simulation results.
- Distributed orchestration is proposed for efficient offloading, and it is evaluated against centralized and cluster-based orchestration with and without caching.

Obtained results demonstrated competitive performance over benchmark algorithms.

- The proposed approach is evaluated against four other algorithms based on task failure rate, network performance, and CPU utilization. Results demonstrated the competitive performance of the proposed approach over benchmark algorithms.

1.3 Publication Submissions

Two journal papers about the thesis work are submitted in IEEE Transactions on Emerging Topics in Computational Intelligence (special issue on Computational Intelligence to Edge AI for Ubiquitous IoT Systems), and IEEE Transactions on Mobile Computing. They are currently under review and the information of submitted papers are as follows:

Submitted paper (Manuscript ID: TETCI-2021-0446) about offloading in Edge Cloud Environment using Deep Q Reinforcement Learning. Submitted on 20th Nov 2021.

[1] V. P. Guddeti and M. Atia, "Optimal Offloading in Heterogeneous Edge Cloud Environment using Online Deep Q Reinforcement Learning," *IEEE Transactions on Emerging Topics in Computational Intelligence,* 2021.

Submitted paper (Manuscript ID: TMC-2022-04-0383) about decentralized offloading and container-based caching. Submitted on 27th Apr 2022.

[2] V. P. Guddeti and M. Atia, "Decentralized Offloading and Container based Caching in heterogenous Edge Cloud Environment," *IEEE Transactions on Mobile Computing,* 2022.

1.4 Thesis Outline

The remainder of the thesis is organized as follows:

- Chapter 2 provides some background on cloud computing, edge computing, offloading scenarios and provides existing solutions in this problem. It also elaborates on how the proposed research overcomes existing resource allocation problems and highlights the significance of the recommended approach. It then introduces Deep learning reinforcement learning concepts and addresses how Deep Reinforcement learning can solve the existing issues.
- Chapter 3 presents the system model, along with mathematical equations. The formulation of objective function, constraints, and the proposed Online Deep Q Reinforcement learning parameters are also explained.
- Chapter 4 puts forward the theoretical analysis for using Deep Reinforcement Learning for this problem. Furthermore, it describes the proposed Online Deep Q Reinforcement Learning algorithm to obtain the offloading decisions for various tasks based on latency, energy consumption, and caching cost.

- Chapter 5 reviews simulation results concerning various benchmark policies using plots and provides reasoning.
- Chapter 6 concludes the research and presents the future work that can be carried out in the near future.

Chapter 2: Background and Related Work

This chapter details the current state of the art on offloading and resource management techniques in edge computing. Section 2.1 provides background on cloud computing, edge computing, edge cloud collaborative environment advantages, and the task offloading scenario. Section 2.1.5 introduces the basic concepts of deep learning along with its types and applications. As the considered offloading scenario is stochastic with multiple parameters, reinforcement learning can serve as an ideal approach to learn the network dynamics as described in 2.1.6. Section 2.1.8 provides a detailed background on Deep Q reinforcement learning that will be utilized in this research. Section 2.2 gives a detailed view of existing research on task offloading scenario, their limitations, and how the proposed thesis overcomes them to suggest a novel approach.

2.1 Background

The enormous growth [21] in the number of laptops, mobiles, intelligent wearables, IoT sensors, can be related to the significant improvements in the communication industry, such as increased data rates and diversified products for various applications. With the rapid development of electronic gadgets, various IoT applications such as online gaming, e-health, video surveillance, connected cars have emerged. Most of these applications work in an integrated manner to achieve the desired functionality. The idea of this connected technology is gaining popularity day by day, and the major reasons include data sensing, integration, analysis, and real-time control. For instance, sensors and cameras work

collaboratively to detect and identify any obstacle in smart traffic monitoring. The aggregated data is processed to make inferences about the distance and location of the obstacle. The observations are then used to control traffic lights accordingly. The concept of smart cities can also be understood by correlating to the above example, except that the amount of data processed and analyzed is many times more. Therefore, the main idea in intelligent services is to perform the desired functionality for any application with the most negligible supervision, and the number is increasing swiftly. Moreover, the development of connected services [22] such as smart cities, smart factories will inevitably increase the number of IoT applications used day by day. Such a rapid increase in applications increases the generated data and the computation required to handle the applications.

Due to the small size, low power, and limited processing capacity of a user device, computation-intensive applications cannot be processed locally. Locally processing them may violate the delay constraints and depletes the battery quickly. Therefore, the data generated by various applications is usually forwarded to central cloud data centers [23] for processing.

2.1.1 Cloud Computing

Centralized cloud servers support varied services [24] such as Infrastructure as a Service, Software as a Service, Database as a Service, Information as a Service, and Platform as a Service based on the requirements of various applications. These powerful datacenters with large processing capacities have been beneficial for many applications over the years.

These cloud data centers host many virtual machines with ample storage space and high computational capacity to execute tasks parallelly. Thus, processing the applications in the cloud helped significantly bring down the energy consumption of user devices. Also, many IoT sensors do not have the provision to process the data locally. Therefore, the logged data from sensors is usually sent to the remote cloud for further analysis, and accurate actions are taken based on the predicted results. Nevertheless, the major problem is that these remote clouds are usually located far from the user. Thus, it costs a considerable service delay comprised of backhaul delay and transmission delay [25] to transfer application data to the cloud. At times, the service delay can be more than the permitted delay associated with a specific task depreciating the quality of delivery for various applications [26]. For example, a computer vision application that detects objects in a self-driving car application require milliseconds latency to ensure safety [27]. This makes the cloud to be a non-ideal location for task execution often.

However, emerging 5G applications with ultra-reliable low latency communication, enhanced Mobile Broadband, and Massive machine type communication posed stringent latency constraints with high processing speed. These latency-sensitive and computation-intensive applications pose a great challenge to meet the service-level agreements and are resilient to scalability when remote clouds are used to process the data. Therefore, traditional cloud computing alone failed to meet the rigorous demands in terms of latency posed by the 5G applications [28]. Moreover, an increased number of users trying to reach the cloud for processing their data increases the burden on available bandwidth and affects

the network performance, as shown in Fig. 2.1. Any disruption in the network affects global task execution and may lead to an outage at times.

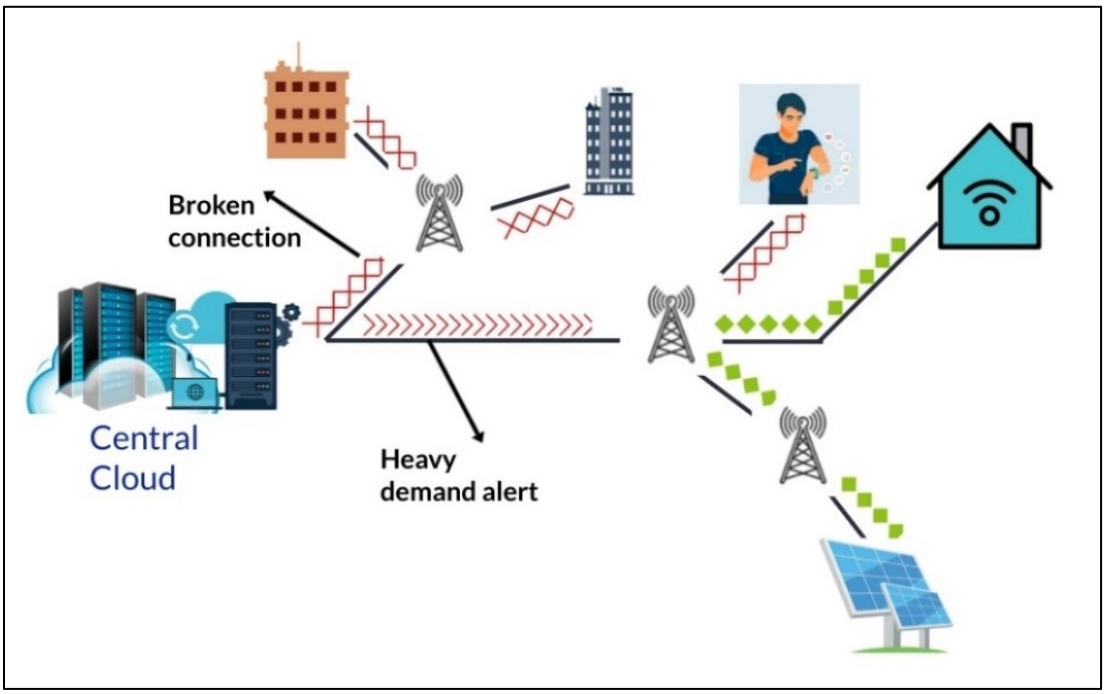

Figure 2.1: Centralized Cloud task execution scenario

2.1.2 Edge Computing

The concept of edge computing has been proposed recently to overcome the challenges mentioned earlier. In edge computing, edge servers are located closer to the user to enhance the user experience by incorporating computational resources near to the user. The edge server is a mini-data center with limited storage and processing capacity for task execution. These edge servers are usually distributed geographically to reduce the load on the remote cloud, enabling load balancing and ensuring a robust network [29]. The ability to run services at the network edge by applying the concepts of cloud computing is termed as

Mobile-Edge-Computing (MEC) by European Telecommunications Standard Institute (ETSI) in 2014 [30], and key characteristics are as follows:

1. **On-Premises:** Edge servers can run in isolated environments utilizing locally available resources. This scenario helps for systems such as the banking sector where security and privacy are essential.
2. **Proximity:** As edge servers are located closer to the user, it is easy to monitor, analyze, and gain real-time data insights. Moreover, the updated information can be directly utilized to perform desired actions for various online applications.
3. **Location Awareness:** The neighboring presence of edge servers enables users to enjoy location-based services such as nearby restaurants, hospitals. Besides, users have access to local aware information and entertainment.
4. **Low latency:** The communication delay incurred to communicate required data and run applications can be reduced by processing them at edge servers. Moreover, bandwidth utilization is also reduced due to the proximity of edge serves. As a result, the quality of delivery and network congestion can be improved by incorporating edge servers.
5. **Network Context information:** The advantage of the proximity of edge servers can be leveraged by aggregating and utilizing the network data along with local contextual information. Quality of experience for various users can be monetized by utilizing the inferences from logged data and by providing customized services.

An edge server can be a roadside unit that fetches data from cars nearby, or it can be a gateway in smart home and so on. These edge servers act as an intermediate layer between user devices and the cloud in calculating the results by processing and analyzing the generated data. In this way, the network traffic can be reduced by cutting the amount of data transferred to the cloud. For instance, several sensors are installed in a solar grid to monitor the system's performance. These sensors generate massive data every day, and most of this data is about the stable functioning of sensors. Sending this entire data frequently to the remote cloud is not of great use and increases the bandwidth usage. Instead, the local edge server can process the generated data, and a summary report or faults alone can be communicated to the remote cloud for further analysis. In this way, geographically distributed edge servers decrease the data traversal and reduce the load on the remote cloud, as shown in Fig. 2.2.

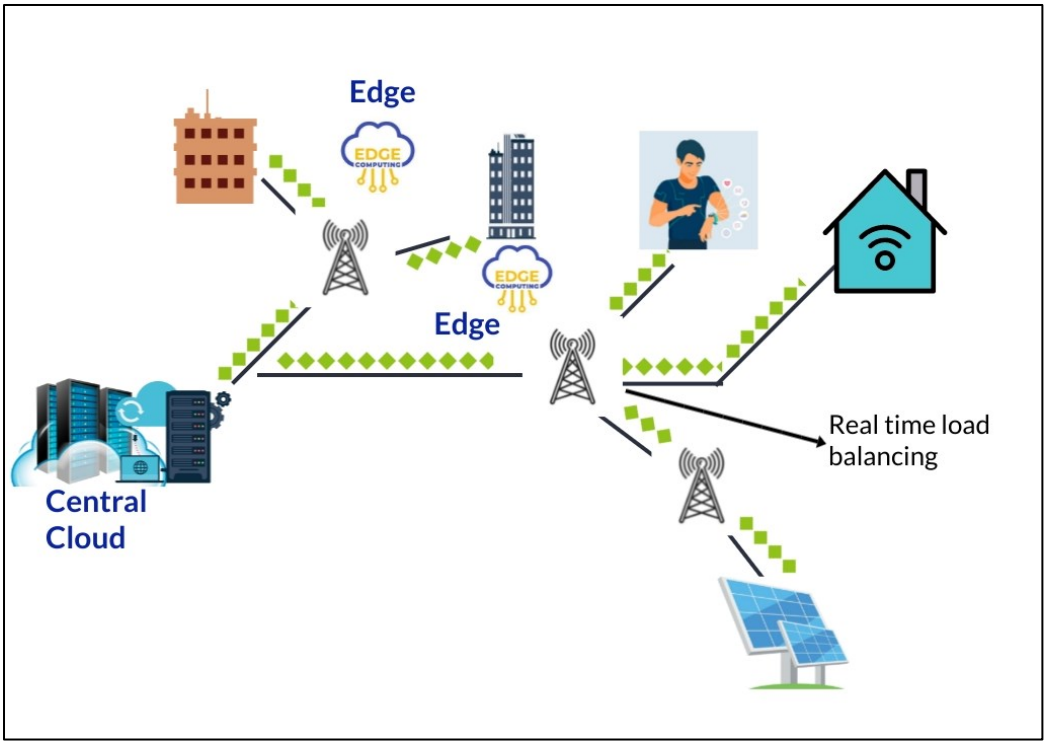

Figure 2.2: Task execution using cloud and edge servers

The main advantages for incorporating edge servers along with remote cloud can be summarized as shown below:

1. **Latency:** The total time to process the data and return the results to the user is known as latency. If computing does not happen at user device, then the time to transfer the input and output information to and from the datacenter should also be added to the latency. Some applications may have service level agreements about acceptable latency. Therefore, latency is a crucial factor for task execution. Since the edge server is closer to the user than the remote cloud, the delay in processing the applications is low.

2. **Available bandwidth:** The available bandwidth determines the wireless data transmission speed between any two devices. Therefore, bandwidth needs to be utilized efficiently to obtain a decent network performance without any disruptions. In a standalone centralized cloud model, frequent data transfers increase the load on available bandwidth. On the other hand, in an edge computing model, smart decisions can be made by analyzing the generated data locally and forwarding only the necessary information to the remote cloud. Therefore, utilizing edge servers makes the network robust and enables load balancing.
3. **Scalability:** The increase in the number of IoT devices may lead to the proliferation of connected applications. Moreover, a high rise in the number of interconnected devices produces massive amounts of data that need real-time processing. Limited available bandwidth and high waiting time at the cloud may restrict the number of applications that can be processed instantly. Therefore, scalability issues in load, network management, and task execution arise when a single centralized cloud server administers the entire network.
4. **Data security and privacy:** Transferring information to a long distance may increase data exposure. This may lead to security breaches or mishandling by hackers. Therefore, transmitting data at short distances through secure servers improves the network's privacy.

However, limited resources available at the edge may limit the number of tasks served in a given time, or the edge server may not possess sufficient storage or computation capacity. This situation mandates the diversion of some tasks to the cloud for quicker computation.

In addition to this, specific tasks generated from applications are pretty small and can be processed in a local device based on the availability of resources. Therefore, to utilize all the advantages mentioned earlier, we need to have a collaborative edge cloud environment for efficient resource management. Thus, it is essential to determine the location of task execution for various tasks for the smooth running of a network.

2.1.3 Task Offloading Scenario

Any generated task can be executed locally, at edge server, or at centralized cloud, as shown in Fig. 2.3. The process of deciding an execution location for various tasks is challenging and is of utmost importance for various reasons:

Firstly, available data centers at the edge of the network, remote cloud, and local server differ significantly in storage space, processing speed, available virtual machines for parallel execution. Also, it should be ensured that the user-device battery is not overly used to run applications locally. Excessive usage of local devices for execution may drain the battery quickly and result in depreciating quality of service. Secondly, the energy consumption differs based on the length of a task and the offloading location. The energy consumption for executing a task comprises of three parts: to send task-related information to the chosen server; to run the applications; to return the results to the user after execution. Thirdly, the delay incurred to transfer task-related information varies based on the location of available data centers and user devices. Fourthly, the available network bandwidth at any instant varies, causing interferences for data transmission. In addition to all the factors

mentioned earlier, there will be multiple users generating tasks simultaneously at any instant, and some mobile users may experience frequent network disruptions.

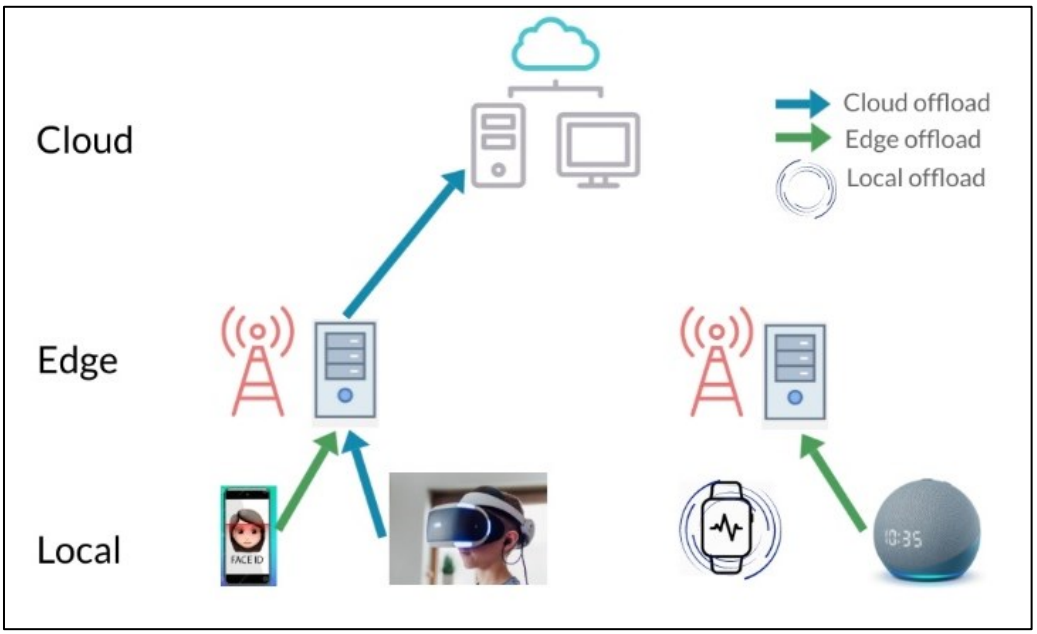

Figure 2.3: Task offloading scenario

Therefore, the scenario of various user devices generating complex tasks, instantaneously changing network parameters, and fluctuating resources make this high-state-space system more complex.

Apart from the above resource-pertinent constraints, the parameters of a task also play a vital role in choosing an execution location. For instance, a task generated for a factory automation application requires results to be returned in the order of milliseconds, but the length of the task might be short. In smart health applications such as remote surgery, large bandwidth, high data rates, and ultra-low latency are mandatory. Any compromises may result in severe consequences. On the other hand, Virtual Reality gaming applications

might generate lengthy tasks with stringent delay requirements. Furthermore, an infotainment application might have relaxed latency requirements but request a huge output size. For that reason, requirements and specifications of tasks should also be considered to determine the execution location of a task in any scenario.

Therefore, it is vital to efficiently decide the execution location of tasks for the smooth running of the network. The offloading process to select an optimal location for task execution can be summarized as follows:

1. The task-related parameters such as length (in Million Instructions), allowed delay, input size, software dependencies should be collected. Along with task-related information, available resources at the datacenter should also be gathered. This includes local processing capacity, edge server, remote cloud processing capacity, number of idle virtual machines, available cores, delay to reach them. Finally, network-related information also needs to be gathered to transfer task-related information to the concerned location.
2. The type of task, such as computation-intensive, delay-sensitive, should be determined based on task-related parameters. It should also be validated if local execution will suffice for short-length tasks. Besides, some local devices such as sensors do not have the facility to execute locally, and the situation necessitates task forwarding. Therefore, potential processing locations should be filtered based on task classification and feasibility analysis.

3. Finally, the execution location of the task is decided based on a constrained optimization framework that achieves the desired goals such as minimum latency, less power consumption, or balanced network bandwidth. Furthermore, a timeslot to schedule the task is chosen, along with a transmission mode.

Therefore, it is essential to decide the execution location of any application based on resource availability, latency constraints, battery power, and task requirements. As we go forward, the number of connected devices will increase exponentially [21], and the generated tasks from these devices will also proliferate, further increasing the complexity. Thus, an optimal offloading methodology is crucial to enable faster, cheaper, and stable task execution. In addition to considering the energy consumed and delay incurred to process an application, storing frequently used data (i.e., caching) can also improve the performance of the network by avoiding repeated downloading of information as explained below:

2.1.4 Caching

According to Cisco's forecast [31], mobile data traffic is expected to increase as much as 800% in the next five years. Intense usage of IoT applications and their stringent requirements lays a significant burden on available bandwidth and affects the quality of service. Therefore, utilizing cache memory of an edge server is recommended to improve the network. Frequently used data can be stored in internal storage or RAM based on access time requirements known as caching.

The software dependencies and required libraries are downloaded from a centralized server to process any task of an application type successfully at an edge server. When multiple users request tasks of the same application type, these setup files must be downloaded repeatedly to run the application. The frequent download of content simultaneously increases the usage of available bandwidth, causing network congestion [32]. Therefore, the popular content can be cached at the edge server to overcome duplicate downloading and improve network performance. Furthermore, as the cost of renting storage space decreases day by day, incorporated cache-storage at edge server is an effective solution. The advantages of deploying cache-based edge servers can be summarized as below:

1. As caching important content at edge servers prevents frequent downloading from the cloud, a large bandwidth to transfer task-related information can be conserved. In addition to this, the popularity of an application varies by location. Therefore, different edge servers can have varied content cached, improving bandwidth utilization.
2. Storing content closer to the user device prohibits sending the task to the remote cloud for execution at times. In this way, running tasks closer to the user minimizes the service latency incurred to provide results to the user.
3. Downloading content from other datacenters or sending tasks to a centralized remote server frequently costs significant energy. As it is essential to minimize energy consumption as much as possible in an offloading problem, caching content can contribute substantially to reducing energy consumption.

4. Furthermore, cached content at one virtual machine of an edge server can be multi-casted to various other machines based on the requirement. Therefore, the spectrum efficiency can be improved by sharing the same spectrum.

In addition to all the above advantages, it is beneficial if dependent libraries and setup files to run any application are cached. The set of libraries, initialization files, configuration files, and dependencies to run an application is called a container [33]. Containerization helps in the smooth processing of applications in any virtual machine. For instance, an application from a Linux machine can be quickly run on a Windows machine with the help of containers. Therefore, it is more feasible and realistic if the setup files of various applications are cached. Edge servers have limited storage allocated to cache data, and the software dependencies of frequently requested applications can be cached.

2.1.5 Deep Learning

With the improved capabilities of electronic devices in terms of hardware and processing speeds, the concept of deep learning is gaining interest for many applications such as object detection, image deblurring, medical imaging. Deep learning is a sub-domain of machine learning that deals with algorithms inspired by the brain's functioning. It consists of processing units called neurons organized as inputs, hidden layers, and outputs in multi-layers. The number of hidden layers can vary based on the complexity of the problem. The word "deep" in Deep learning means the depth measured in the number of hidden layers. Each layer has multiple nodes that are connected to nodes in adjacent layers using a

connection value called weights. An example of a neural network can be seen in Fig. 4. All neurons present in each column can be considered as a layer, and a neural network usually consists of 3 types of layers: input, output, and hidden layers [34]. Each output from a layer is taken as input to the adjacent next layer. Fig. 2.4 consists of two hidden layers and can be called a 2-hidden layer network.

The multi-layer network with many neurons identifies the complex relationships between inputs and outputs by abstracting the data at a high level. For instance, a neural network can be trained with many images to detect objects of interest in an image based on color intensities background information. The massive data generated by IoT applications can be analyzed to extract essential information using deep learning. The deep learning methods [35] can be majorly classified into three types as shown below:

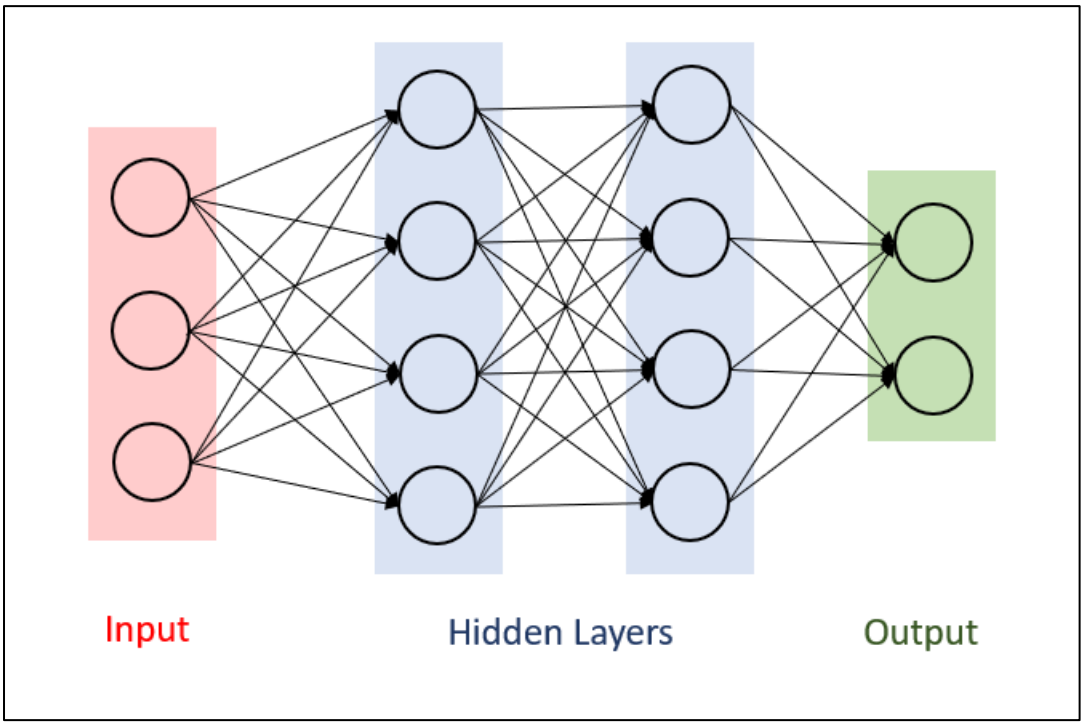

Figure 2.4: Neural Network with two hidden layers

1. **Supervised Deep Learning:** In this technique, a set of inputs and their outputs are fed to the network, and the model modifies the network's weights to obtain the correct outputs. The predicted and actual output difference is used to correct the weights to improve future predictions. The same process is repeated for multiple iterations before the model is deployed to identify relationships for unknown data.
2. **Unsupervised Deep Learning:** This method manages massive data without any labels. It depends on the relationship within the input data itself to infer useful knowledge and discover patterns. Therefore, the designed model works on given data to gain insights and identify relationships between various samples. Some of the operations include clustering and dimensionality reduction.
3. **Semi-Supervised Deep Learning:** Semi-supervised learning can be considered as an intermediate technique between supervised and unsupervised learning. The provided input does not have corresponding labels but can inform if the predicted output is right or wrong. For instance, some information about the generated action is usually informed to the model to correct the weights. A popular example of this is training a model to play a video game. Over time, the model learns how to play based on the rewards and penalties it obtains for performing various actions in a given situation.

Deep learning methods can be further classified based on the working mechanism and type of data, as shown in Fig. 2.5 based on above categories.

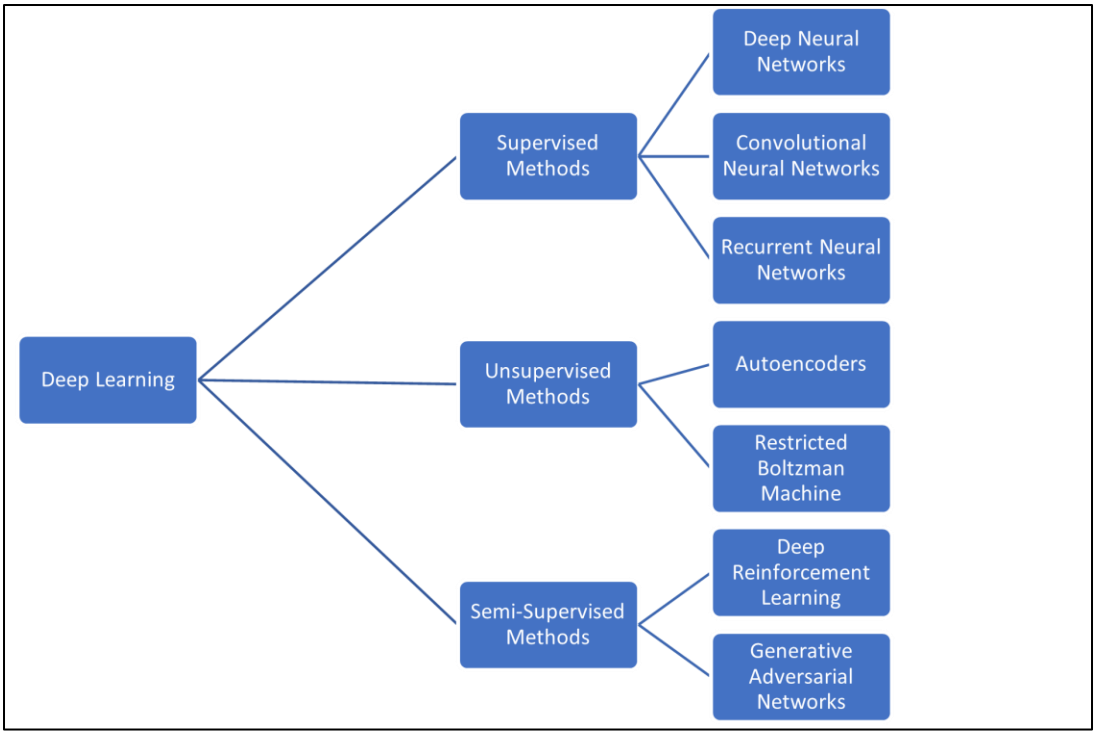

Figure 2.5: Deep Learning methods classification chart

2.1.6 Reinforcement Learning

Reinforcement Learning (RL) is a semi-supervised machine learning technique that works using the trial-and-error method. In this, an agent learns an environment based on the rewards/penalties it gets for performing an action. The overview of RL can be seen in Fig. 2.6. The collective feedback is used to train or correct the model for predicting appropriate outputs in the future. The desired outputs are obtained by exploration in the initial stages and by exploitation of the known environment in subsequent iterations. The main objective in RL problems is to choose the best action such that aggregated reward is maximized. The self-learning mechanism to take a right action based on the rewards and penalties makes

this algorithm well-suited for dynamic environments. Markov Decision Process, as explained below, is usually used for RL modeling.

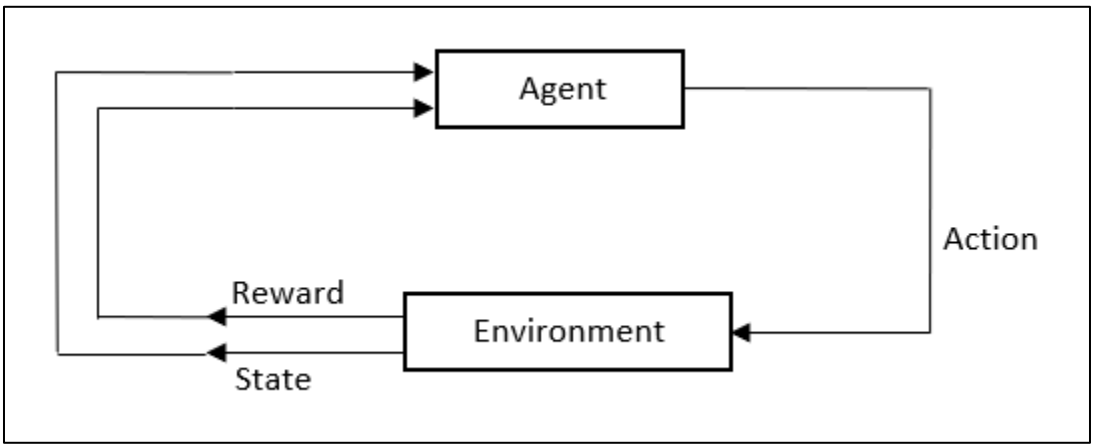

Figure 2.6: Reinforcement Learning overview

2.1.6.1 Markov Decision Process

Markov Decision Process (MDP) [36] inspired from Markov chains is used for decision-making in stochastic environments. Future states are predicted from the present state by identifying a desired action from the current state. No past state information is used for estimating the future states. For instance, a chess game does not require any information about previous states to reach a future state. The possible actions from the current state will suffice to take any action. Moreover, the concept of MDP [37] can be applied to other domains such as operations theory, fleet maintenance, automation. The main components of MDP are explained as below:

1. **Environment:** Environment in RL problem comprises rewards, penalties, actions, possible states. The agent interacts to perform an action, and corresponding outcomes are

achieved in an environment. This can be a boundary within which a robot moves, a chessboard where the chessmen move, an Atari Wall breaker game.

2. **State:** State can be defined as the current or future position of the agent in an environment. For example, the set of coordinates the robot's legs can be termed a state for a line following the robot. For a thermometer, the temperature reading can be called a state. Therefore, a state in the RL environment defines the agent's current situation. Therefore, a set of positions that the agent can attain in the provided environment can be defined as a state.

3. **Action:** Action can be defined as the possible moves of the agent from the given state. For example, in a flappy bird game, the bird can perform three actions: flap upwards, downwards, or move forward. In a line following robot, two actions are possible: forward and backward. Therefore, actions can be defined as moves to navigate the RL agent's current to future state.

4. **Reward/Penalty:** Reward is the numerical value used to define the significance of the present state. All states in an environment are mapped to a value, either positive or negative, to determine the worth of any state at a given time. A higher reward for moving in the desired way and a lower value for an undesired move guide the agent to reach a goal point. Therefore, reward acts as feedback to train the model to take desired actions. For instance, an autonomous car can be navigated to follow a race tracker using reinforcement learning. If the waypoints are properly followed, the agent will be given a positive numeric value

and a negative value if the car crosses the provided boundary. Therefore, the agent tries to maximize the cumulative reward to reach the endpoint of a racetrack.

5. **Policy function:** A policy can be defined as a mathematical function to correlate state and actions at any given time using a reward. The policy function guides the agent to move to further states and reach optimality. A policy function defines the best action taken from the current state, and an example of a policy is the Bellman equation [36].

With the above primary components, RL can be applied to a variety of applications to improve performance based on previous experiences. An optimal policy can be identified to move between different states by performing actions such that reward is maximized. RL can be categorized in many ways. Any RL algorithm can be based on prediction and control, or it can be based on policy function and value. Also, RL algorithms can be classified based on model-based and model-free. Furthermore, a detailed description of model-based classification will be elaborated on the following pages.

Model-based RL: In model-based algorithms, the agent develops the model of the environment over a period based on the transition between states using actions and rewards. The developed model is used and updated gradually during the training to reach a goal point or end state. As the entire information of previous transitions is stored and updated, computational complexity is large in model-based RL, and strong convergence is not guaranteed. They are well-suited for target-driven approaches, and they learn through planning [38].

Model-free RL: Model-free algorithms usually work on trial-and-error methods, and they learn to take the best action from the current state based on past experiences. The model learns ideal weights and parameters iteratively, and the convergence is much slower when compared to model-based RL. Strong convergence can be guaranteed once the parameters are learned. Monte-Carlo and Temporal Differences methods are commonly used algorithms in model-free algorithms. The functioning of Monte Carlo is similar to sampling from a probability distribution. The agent learns to reach the end goal either using optimal policy or by value in multiple episodes. However, the only drawback in Monte Carlo is that the agent should wait till the end of the episode to get the value function which can be used further. Unlike Monte Carlo, which updates after every episode, the temporal differences method works after every time step. This method is ideal for continuous environments and has low variance with some bias. Some of the popular model-free algorithms are SARSA and Q-learning.

2.1.7 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) combines Deep Neural networks and Reinforcement Learning, as shown in Fig. 2.7. The states of the model are given as inputs to the neural network, and outputs of the network are the actions. A desirable action is selected and performed in the environment based on the agent's policy. The agent moves to the next state based on the chosen action, and the obtained reward is fed to the neural network to update weights. DRL overcomes the curse of dimensionality by using function

approximators and finds the correlation between state-action pairs and rewards. They can be used for both offline and online training to learn an unknown environment and then act accordingly. The main advantages of adapting to DRL can be summarized as below:

1. Although DRL takes much time to converge, they can quickly adapt to frequent changes in an environment once converged.
2. As the DRL works by trial-and-error methods, they are ideal for being employed in an unknown environment that requires sequential learning and adaptability.
3. While the conventional RL methods have restricted state space, DRL can be used to solve high-dimensional significant state-space problems by using powerful function approximators.

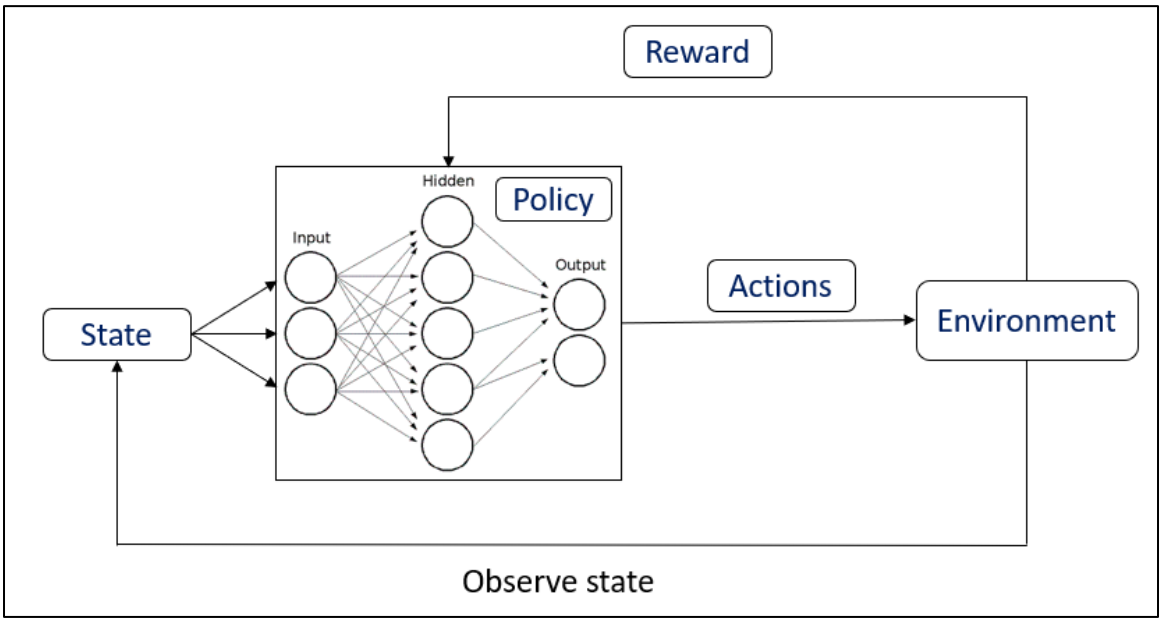

Figure 2.7: Deep Reinforcement learning structure

2.1.8 Deep Q Reinforcement Learning

Deep Q reinforcement Learning (DQRL) combines reinforcement learning and deep neural network (DNN). It uses reinforcement learning to understand the environment based on the rewards obtained while performing an action. It is usually used with the most negligible supervision when automatic learning is required. Unlike supervised learning, where we know the true class a sample belongs to, DQRL methods learns the value of an action based on the reward/penalty it gets as the agent moves. As there are several actions possible from any state, and each action lands the agent in a different destination, the agent should carefully select its move further. The agent will be punished/rewarded based on the actions taken. Model continuously learns in this way, and the best solution is determined based on maximum reward. This type of learning is beneficial for resource allocation and NP-hard problems where human-generated heuristics are needed. Amongst various available algorithms under the umbrella of Reinforcement Learning, Q-learning is more popular with proven significance in many applications. It is a model-free learning technique to learn the environment based on chosen actions iteratively as shown below.

Q-learning: It is the most used technique because of its ability to learn the effect of actions without any model and has a proven success rate for different applications [36]. Q in standard Q-learning (QL) stands for "quality" and is an off-policy learning technique that determines the value of an action based on obtained reward. It uses Q-table to keep track of state, action, reward and maps the state-action pair to a Q-value, which the agent will

keep on learning [39]. Q-value is an expected cumulative reward given the current state (s_t) and action (a_t). It can be denoted by:

$$Q^{\pi}(s_t, a_t) = E(R_{t+1})|[s_t, a_t] \quad (2.1)$$

where $E(R_{t+1})$ is the expected future reward for performing the action (a_t) from the state (s_t). The agent in QL learns to calculate the expected reward in the given environment gradually through multiple iterations using Temporal Differences (TD). Calculated rewards are tabulated along with state and actions throughout the model. The QL process starts by initializing a Q-table with some random Q values. Random action is chosen and performed from the current state. Based on the reward obtained, Q-value [40] for that state-action pair is updated as follows:

$$Q^{*}(s_t, a_t) = Q(s_t, a_t) + \alpha(R_t + \gamma.max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)) \quad (2.2)$$

where $Q^{*}(s_t, a_t)$ refers to updated Q value based on obtained reward (R_t) by performing an action (a_t) from the state (s_t). α denotes the learning rate, and γ is the discount factor. In the initial stages of exploration, actions are taken at random, and Q-table may not be accurate. Over multiple epochs, this Q-table will be updated to learn the environment precisely using (18). The pseudo-code to design the Q-learning algorithm is shown in Algorithm 1. In this way, the agent increases its confidence to identify the correct action from the current state in any complex environment by exploration and exploitation.

The ability to correct the model according to the environment such that desired states are reached makes Q Learning more popular for various applications. The main drawback in Q-learning is that it tabulates the state-action pairs with values and uses the mapping to decide future actions. However, this may not be feasible when working with a large data set. Instead, utilizing neural networks to map state-action pairs with rewards can be beneficial and a great alternative. Neural networks can predict complex relationships between input and output and are used extensively in many applications such as image processing, speech recognition. Deep Q reinforcement Learning (DQRL) can be described as the application of Neural networks on Q-learning [36]. Unlike Q-learning, where state-action values are mapped to a Q-value, the neural network is used to map an input to action and Q-value in DQRL.

Algorithm 1: Q-Learning

1. Initialize Q-table with default values
2. for N iteration, do
3. Repeat
4. start from state, s_t
5. choose a greedy action (a_t)
6. perform action, (a_t) and calculate reward, (R_t)
7. Update Q value based on Eq. 2.1
8. Move to next state (s_{t+1})
9. Store transition (s_t, a_t, r_t, s_{t+1}) into Replay memory.

10. until s_t is terminal

11. end for

DQRL uses two networks (Train and Target Network) to calculate Q-values and for maintaining stability [36]. These two networks have the same architecture but different weights. In the beginning, these two networks will be initialized with some weights. Q-values are calculated using the train and target networks for all possible actions (a_t) from the current state (s_t). An action corresponding to the maximum Q value is then selected, and only train network weights are updated based on the error in approximating Q value. After every epoch, train network weights are again copied to the target network.

Since many researchers leverage neural networks and reinforcement learning techniques to solve task offloading problems, a background on available algorithms along with their inputs, outputs, and use-cases is presented in the above sections.

2.2 Related Work

With the rapid growth in number of mobile devices, the IoT applications that require processing of data is increasing day by day. The data generated by these applications not only require wide bandwidth but also demand high computation speed, quick response, and long battery life. However, the processing capacity and storage available at the user device will not be sufficient to handle the data locally. Therefore, this gap is usually fulfilled by centralized data centers such as Microsoft Azure [41], Amazon Web Services [42] and so

on by computing the data at virtual machines located in these data centers. Nevertheless, the advent of 5G has resulted in much more stringent service-level agreements for recent IoT applications such as autonomous vehicles, Augmented/Virtual reality gaming and so on. The rigorous demands of latency, data storage, and processing speed of IoT applications is challenging the traditional cloud computing model and is laying huge burden on available bandwidth.

Therefore, edge computing has been proposed recently to add network resources closer to the user. Edge computing [3] is a distributed computing approach where datacenters with limited storage and processing capacity are available closer to the user. The close proximity of geo-distributed edge servers decreases the response time and reduces the burden on available bandwidth. However, the limited resource availability at edge servers restricts the amount of data that can be handled in a network. As a result, it is very crucial to decide how data generated by various IoT applications needs to be handle for smooth functioning of a network.

The task offloading problem in an edge cloud environment is highly challenging due to many factors such as heterogeneity of resources, remote data centers, erratic user mobility, dynamic network resources. Numerous variables with ample state space classify the scenario as an NP-hard problem [43] [44] [45]. Therefore, massive research is going on to find a suitable algorithm to decide the offloading location of generated tasks efficiently. Many resource allocation strategies with different design objectives were proposed. In [46], an efficient optimization scheme is designed to minimize radio resource allocation along

with energy consumption. The research classifies the available user devices in a network into three categories based on their storage and computing capacity. The available bandwidth is then allocated in the order of their classification after every iteration such that overall energy consumption will be optimized. Authors in [47] address the offloading problem in the presence of a central cloud with Time Division Multiple Access (TDMA) and Orthogonal frequency division multiple access (OFDMA) network types. Furthermore, authors in [28] reviewed the interaction between edge servers and remote cloud such that energy consumption of the entire network can be minimized. In [33], research is focused on distributed power allocation for user devices and edge servers such that end-to-end latency of IoT applications can be minimized in a multi-user system. Authors in [10] formulated a latency minimization objective function under the constraints of power consumption. Authors in [48] proposed that computation of next generation application data can occur in other user devices along with edge servers. In [49], the inter-cell interference environment of dense small cells and problem of computation sources of MEC are considered. An adaptive scheme to reduce the queuing delay of IoT applications is proposed.

In [50], a low complexity online set of rules referred to as “Lyapunov optimization” is proposed for a MEC system with energy harvesting devices. It is based on dynamic computation offloading set of rules and determines the offloading choice based on the CPU-cycle frequencies and the transmit power for offloading. Authors in [51] studied the partial offloading problem such that device energy consumption and execution delay can be minimized. The scenario of a single-user and multiple users are considered, and the

results demonstrated that local execution is the optimal scenario possible. Moreover, it was inferred that offloading the entire generated data for processing remotely is not ideal for devices with dynamic voltage scaling feature. Researchers in [52] proposed an opportunistic offloading framework to handle online streamed data of 12 users for 15 days.

Authors in [53] reviewed several mobile cloud computing models and summarized their approaches along with their shortcomings. [54] address the problem of task offloading considering different mobile devices, varied cloud vendors offering multiple infrastructures, services through different network topologies. It also reviews how this heterogeneity impacts the problem of computation in the presence of a central cloud. Furthermore, Researchers in [55] presented about security issues in various edge computing models. Authors in [56] focuses on pricing models for various IoT applications that play a vital role in resource management in edge computing.

Authors in [57] proposed an edge orchestration-based computation peer offloading algorithm where Multi-Access Edge Computing (MEC) resources are shared using horizontal offloading. If the load on any edge server is more, tasks are forwarded to an adjacent server for execution as a single edge server cannot handle many tasks, and performance will be degraded. Using fuzzy logic, the target node for computation is selected based on server utilization, task size, and network latency. The edge server processing capacity, task computation amount, type, and requirements are sent to the orchestrator, and it acts as a decision-maker in deciding the computation node. In [58], computing will be done utilizing other vehicle resources (Virtual Edge). Instead of

offloading to RSU or some other base station, vehicles in the vicinity are identified, and then the duration of link with those vehicles their availability for computation is validated. A trajectory equation between two vehicles determines potential virtual edge members. Some virtual edge nodes are selected, and tasks are offloaded to these nodes or vehicles. Companion time is the parameter used to find the available duration of the virtual edge. Since vehicles are mobile, it was taken care that the number of vehicles of a virtual edge is limited, and connections are stable. Vehicle information is obtained through primary safety message exchanges, and available computation resources at each vehicle are also calculated. Image recognition and video retrieval tasks are considered for computation. The objective function was to minimize the execution time of computation at the virtual edge. In [59], a mobility aware computation scheme is proposed by modelling the connection time between any two users with an exponential distribution. In [60], latency and energy consumption are considered as parameters for efficient offloading of tasks in edge computing. Weight factors are assigned such that appropriate weights can be assigned based on the task's requirements. Thus, the objective function is the weighted sum of latency and energy consumption by optimizing the offloading decision, transmit power, offloading time, and available CPU frequency for the vehicle. It is assumed that vehicles generate tasks based on TDMA protocol. The Lagrange multiplier is used to solve the minimization function with constraints to determine the ideal location for offloading the tasks. In [15], fog computing nodes are considered instead of edge computing, and the cost function is comprised of energy consumption and latency.

In all other research works ([61], [62], [63], [40], [64], [65]), weighted sum of energy consumption and latency is considered as objective function. Constraints differed in most of the research, such as the number of computation resources available and computation capacity. Though few researchers limited offloading only to edge servers and remote cloud, some research even considered adjacent servers for computation.

In [66], particle swarm optimization is implemented to find the offloading decision. The particle swarm optimization algorithm considers available storage space, bandwidth and computation capacity of all data centers and maps them to user devices randomly. The total energy consumed for chosen mapping is minimized in multiple iterations to find best allocation of computation capacity and bandwidth. Researchers in [11] work on minimizing the transmission power allocation to edge servers for maximum coverage of user devices. Furthermore, K-means clustering is used to choose the execution locations for various devices. However, the solutions based on heuristic algorithms do not guarantee a global minimum. Also, since heuristic algorithms run for multiple iterations, they are not very well suited for rapidly changing environments.

Immense popularity and advancements in reinforcement learning have inspired researchers to apply reinforcement learning to this problem. Ordinary reinforcement learning algorithms rely more on the previous state than the current state and can work only for finite state-space systems. In [39], an optimal location is found to offload the tasks amongst the edge server, adjacent server, and remote cloud. Processing delay and energy consumption were the considered parameters in determining the location, and offloading

is only considered when local execution time is more significant than remote execution time. SARSA deep reinforcement learning was used for minimizing the objective function, and no information regarding the tools used for implementation was mentioned. In [67], the number of deep learning layers is calculated for different types of tasks to be computed. Based on the latency requirement, available computing power at the edge server for different tasks, computation resources are allocated for vehicles. The number of layers that should be computed at vehicle and edge is determined by using Chemical Reaction Optimization algorithm. Deep Neural Network (DNN) specific profiling estimates latency for task execution. In [68], Residual Recurrent Neural Network is utilized to map available hosts to generated tasks such that energy consumption, response time, and running cost are minimized. In [69], just like other papers, the cost function is a weighted sum of energy consumption and latency. They used Deep learning to minimize the function. As unsupervised deep learning is difficult to handle, the first best costs are calculated for some data, and labeled data is obtained in this way. This is considered as training data, and new data is tested accordingly.

Nevertheless, the approach is not so reliable as labeled data needs to be prepared each time, and tuned hyperparameters may not be ideal for all types of tasks or scenarios. In addition, test data performance will exclusively depend on scenarios covered by training data. To summarize, most of the research papers incorporated Reinforcement learning in deciding the offloading locations and few researchers considered fuzzy logic because of its simplicity.

One common drawback in all the above-mentioned optimization algorithms is the assumption that tasks are known beforehand. The algorithm is trained with known data for multiple iterations to obtain decent performance. Furthermore, most of the current work mainly considered channel gain as the input to decide the offloading location and failed to consider other critical features of the task (permitted latency, request size, length) and available resources at the data center (edge or remote cloud). To address these limitations, this research recommends an online Deep Q Reinforcement Learning technique that self-learns the environment and takes offloading decisions for new tasks such that latency, energy consumption, and network performance are optimized. The proposed approach is compared against Fuzzy logic, Simulated Annealing, Round Robin, and Trade-off through extensive simulations to highlight the reduction in task failures, network usage, energy consumption, execution delay, and so on.

Due to the increasing applications with stringent timely delivery policies, further reducing the computation latency is a significant challenge. Caching can serve as an excellent solution for this by eliminating the repeated computation of data, reducing bandwidth, and delay in providing the results to users. Caching can be defined as storing important data either in random access memory (RAM) or read only memory (ROM) for easy access during computation. Nevertheless, limited research exploits caching-based offloading strategies for efficient resource allocation. In [70], energy consumption is optimized, making sure that caching tasks do not exceed the maximum available caching capacity. In [71], offloading decisions are taken based on the profit for caching a particular task and energy consumption, but latency is completely ignored. In [72], Bayesian Network theory

is used to select tasks that need to be cached, and selected data is placed at edge servers with lower load. However, the line of the above work caches the tasks generated by user devices. The set of task computations usually differs for every user based on input data even when the applications used are identical. Therefore, caching a task in a network may not be very useful as the output or input data for every user is stochastic and is dissimilar most of the time. The assumption that generated tasks is similar works in exceptional scenarios, which thus motivates our work. We introduce container-based caching to alleviate the burden of recurrent data downloading. Caching a container prohibits us from downloading the required setup to run an application each time, and the task can be directly computed. The proposed container-based caching is reviewed with task-based caching, and the results are illustrated.

Moreover, most of the existing studies [73] [74] [75] [71] either do not mention the location of algorithm deployment or assume a centralized offloading phenomenon. However, the orchestrator location plays a vital role in the utilization of network resources. It incurs much bandwidth and delay for a centralized orchestrator to send the generated tasks information, resource availability, and to forward the results. Therefore, this thesis proposes a decentralized offloading method where offloading decisions are taken locally. The Online DQRL algorithm chooses the computing location of various tasks, and the network weights are updated globally. The recommended approach is compared against centralized and cluster-based offloading techniques to demonstrate the virtuous performance on a large scale.

In addition to this, a realistic simulation environment with stochastic tasks arriving dynamically from various user devices is considered. The network and data center resources are updated periodically to demonstrate the actual network where available storage, computation capacity, battery levels, mobility, etc., will be varying continuously.

2.3 Chapter Summary

The above chapter gave a background on task offloading scenarios in edge computing and presented the literature review. Initially, the concepts of cloud computing and edge computing and their advantages were elaborated. Then, the necessity of task offloading and challenges in managing are explained. Finally, existing approaches based on various techniques were summarized along with their advantages and disadvantages. A novel computation offloading strategy to overcome the existing limitations will be proposed in detail in upcoming chapters.

Chapter 3: Problem Formulation

Section 3.1 presents the detailed system model in this chapter, and Section 3.2 formulates the objective function for task offloading scenarios in an edge cloud collaborative environment. The list of notations is given in Table 1.

Table 3.1: Notations List

Notation	Description
x_1^t, x_2^t, x_3^t	Local computation, offloading at edge server, offloading at remote cloud for task, t
D_{ij}^e, B_{ij}^e	Transmission rate and channel bandwidth between device i and edge server j respectively
D_j^r, B_j^r	Transmission rate and channel bandwidth between edge server j and remote cloud respectively
P_j^i	Signal power between device i to edge server j
I_i	Interference to device i from other edge servers
t_i	Transmission power of device generating task i
d_{ij}	Distance from device i to edge server j
l_t	Length of task (measure in Million instructions (MI))
f_i^l	Computation rate of device (MI/sec)
s_t	Input data size
γ_{jt}	Caching decision of task (k_t) at server j
f_j^e	Computation rate of edge server j (MI/sec)
f^r	Computation rate of remote cloud
α	Path loss
ρ_t	Allowed delay for task
τ	Number of requests for task i
R	Total number of tasks
c_j	Total cache size of server j
c_{av_j}	Available cache size of server j
σ_j	Memory access time of server j
h_{ij}	Channel gain between device i and server j

3.1 System Model

The considered system model assumes that tasks are periodically generated from devices such as smartphones, laptops, sensors, and raspberry pi. The task parameters comprise permitted delay, length (in Million Instructions), input size, dependency files size, and output size to execute tasks of type augmented reality, infotainment, and e-health. The system comprises of a central cloud server, edge server, and several virtual machines at cloud and edge server. A central cloud server [53] is a data center with large storage space and high computation capacity to handle massive data at any instant. A centralized cloud server (R) that can manage the entire network is considered. Along with the cloud, a set of edge servers $E = \{e_0, e_1, \ldots, e_s\}$ are considered. Edge servers [5] are small datacenters closer to user devices with limited storage and computation capacity to handle data generated by IoT applications. At any instance, if the edge server associated with a user device faces disruption in service, then another edge server located near to the user can be utilized for processing and is called as adjacent edge server. In addition to this, it is also considered that remote cloud and edge servers host several virtual machines to enable the parallel execution of tasks. A virtual machine [76] is a software resource that can deploy and process data instead of using a physical computer. It contains a central processing unit (CPU), and memory disk to store and execute files. Multiple virtual machines can be hosted by a single computer host to enable parallel execution. The number of virtual machines available at the central cloud is much more than the edge server to replicate a real situation. The processing capacities also differ significantly along with the number of virtual

machines. The generated task can be executed either locally, at edge server, or remote cloud at any instant [63] as shown below:

$$x_l^t + x_e^t + x_c^t = 1 \qquad (3.1)$$

where $x_l^t, x_e^t, x_c^t \in \{0,1\}$ denote the decision to process locally, at edge server, or at remote cloud, respectively.

It is assumed that edge servers are placed along with base stations to model service delay. A reporting edge server is associated for each edge device based on their distance. The device is associated if the calculated distance is less than the coverage radius of the edge server. At any instance, if the device lies within the coverage of multiple edge servers, then data rate is used to associate an edge server. The wireless data rate [11] between edge server and user device can be calculated using transmission power, distance, interference from other base stations as shown below:

$$D_{ij}^e = B_{ij}^e log_2(1 + \frac{p_j^i}{\sigma^2 + I_i}) \quad (3.2)$$

where D_{ij}^e is the data rate between device i and edge server j, B_{ij}^e denotes channel bandwidth between device i and edge server j, p_j^i denotes signal power of device i to edge server j, σ^2 denotes signal noise, and I_i denotes interference from other edge servers. The signal power of the device can be calculated using channel gain (h_{ij}), transmission power (t_i), and the distance between device and edge server (d_{ij}), and the path loss (α) as follows:

$$p_j^i = t_i h_{ij} (d_{ij})^{-\alpha} \quad (3.3)$$

3.1.1 Caching Model

Frequently used data can be stored in internal storage or RAM based on access time requirements known as caching. Data sent to the remote cloud can be cached locally as numerous communications with the central cloud lay a considerable burden on bandwidth and increase the delay incurred in processing the inputs and results. Unlike the centralized cloud, edge servers have limited memory to store data and need optimal utilization for resource management.

Container-based Caching: The software bundle of dependencies and libraries required to execute an application-type task is a container. These containers need to be downloaded every time by an edge server or a user device for running a task of an application type. Caching these containers locally or at the edge server based on storage availability and frequency of application requests eliminates the recurrent downloading of dependencies every time. Moreover, limited storage space can be managed by replacing existing cached containers with updated popular containers of an application type. As the popularity of an application varies by location, different edge servers can have varied containers cached, improving bandwidth utilization.

Therefore, a proper selection needs to be made to choose if data must be cached on an edge server or at a user device. The decision to store needs to be taken by carefully considering

the benefit of caching a container and storage cost. The benefit of caching [33] a container can be calculated based on the number of requests for an application (τ) and the total number of requests (R) in a given time.

$$P_t = \frac{\tau}{\rho_t R} log \frac{c_{av_j} f_j^e \sigma_j}{c_j} \quad (3.4)$$

where P_t denotes benefit for caching an application (k_t), ρ_t is the permitted latency, c_{av_j} is the available cache size at edge server j, f_j^e is the computation speed of edge server j, σ_j is the memory access time of edge server j, and c_j is the total cache of edge server j. Loss to cache data can be shown in Eq. (3.5), where β is the unit storage cost incurred to cache input data, s_t and x_e^t is the decision to offload the task at edge server.

$$C_l = x_e^t s_t \beta - \frac{\tau}{\rho_t R} log \frac{c_{av_j} f_j^e \sigma_j}{c_j} \qquad (3.5)$$

3.1.2 Offloading Scenarios

At any instance, a user device can execute the task in three locations: locally, at edge server, at remote cloud. In case the device is a sensor, then the generated data should be handled externally as the device does not have the provision to process locally. Along with the offloading location, a virtual machine amongst the available machines should also be selected based on the waiting time if the task is offloaded to edge sever or remote cloud. At any instance, energy consumption is due to the downloading of containers pertaining to

tasks from remote cloud and for executing the task. If a task is decided to be executed locally, then containers are first downloaded at edge server and then sent to user device. Similarly, latency incurred in processing the containers should also be considered. If tasks are executed at edge server, or at remote cloud, then delay sending the task related parameters are also considered.

3.1.2.1 Local Offloading

If the task is offloaded locally, then computation resources of the device are used to process the task, and delay is the time consumed to execute the task. Local processing of tasks can take more time as the computing capacity of the device is limited. If the task is generated from a device such as a sensor, it will not even have the facility to process locally, and the task must be offloaded to a different location. The local energy consumption [63] comprises of downloading the containers of a task from remote cloud through edge server and computing the task. At any instance, if task k_t with length l_t is computed locally, then energy consumption can be calculated as follows:

$$E_t^l = k l_t {f_i^l}^2 + \left(1 - \gamma_{jt}\right)\left(\frac{t_i c_t}{D_{ij}^e} + \frac{t_i c_t}{D_j^r}\right) \qquad (3.6)$$

where k is the switched capacitance, f_i^l is the computation rate of the device, which is measured in Million Instructions per second (MIPS), l_t is the length of task measured in Million Instructions, c_t is the container size, D_{ij}^e denotes the data transmission rate

between device i and edge server j, γ_{jt} is the caching decision, and D_j^r denotes the data transmission rate between edge server j and remote cloud. To execute a task locally, containers need to be downloaded from remote cloud through edge server if they are not cached. Therefore, Eq. 3.6 comprises of energy incurred to download the containers and to process the tasks locally . The delay equation is modelled to capture the time required to process the task locally $\left(\frac{l_t}{f_i^l}\right)$ and the time to download the container (c_t) from remote cloud through edge server if the task is not cached (γ_{jt}). The data rate in between edge server j and user device i is denoted as D_{ij}^e. Similarly, data rate between cloud server and edge server j is denoted as D_j^r. Therefore, time taken (latency) [15] to execute a task locally, including the delay to download the container from the remote cloud, can be expressed as:

$$L_t^l = \frac{l_t}{f_i^l} + \left(\frac{c_t}{D_{ij}^e} + \frac{c_t}{D_j^r}\right)(1 - \gamma_{jt}) \qquad (3.7)$$

3.1.2.2 Edge Offloading

If the device decides to offload a task (k_t) at the edge device, virtual machines at the edge server are utilized to process the task. In this scenario, the time taken to send the task to and from the edge server will also be added along with container downloading time to the latency. Since the result size is much smaller than the request size, the delay experienced to return the results are ignored. There might be several virtual machines available at the edge server that run tasks in parallel, and tasks will be offloaded to a virtual machine that has the least number of tasks waiting in the queue. The computation rate of the selected

virtual machine is denoted by f_i^l, and the size of the task is represented by s_t. At edge server, energy consumption [63] comprises downloading the container (c_t) from remote cloud if task is not cached (γ_{jt}), downloading the data (s_t) from user device and then executing the task. Based on the decision to cache any task, which is denoted by γ_{jt}, the total energy consumption is calculated by:

$$E_t^e = \frac{t_i s_t}{D_{ij}^e} + \frac{t_i c_t}{D_j^r}(1 - \gamma_{jt}) + k l_t {f_i^e}^2 \quad (3.8)$$

where t_i is the transmission power of the device, and f_i^e is the computation capacity of edge server j. Ignoring the time to send the results back to the device, computation delay comprises of three components: sending the task to the edge server (s_t), downloading the container(c_t) from the remote cloud if task is not cached (γ_{jt}), and computing the task. Therefore, latency [15] for offloading a task (k_t) to edge server can be denoted as:

$$L_t^e = \frac{s_t}{D_{ij}^e} + \frac{c_t}{D_j^r}(1 - \gamma_{jt}) + \frac{l_t}{f_j^e} \quad (3.9)$$

D_{ij}^e denotes the data transmission rate between device i and edge server j, D_j^r denotes the data rate between edge server j and remote cloud, and l_t is the task length (k_t).

3.1.2.3 Remote Offloading

If the task is offloaded at remote cloud, backhaul delay [20] to send the task from base station to remote cloud will be added along with latency encountered to send it to the base station (co-located with edge server). Therefore, energy consumption and latency to process the task at remote cloud includes downloading the input data (s_t) from device i through edge server j, and then executing the task. If D_j^r is the data rate between edge server j and remote cloud, D_{ij}^e denotes the data rate between device i and edge server j, and f_j^r is the computation rate of remote cloud, then energy consumption and latency for offloading a task (k_t) at the remote cloud can be calculated by:

$$E_t^r = \frac{t_i s_t}{D_{ij}^e} + \frac{t_i s_t}{D_j^r} + k l_t {f_i^r}^2 \qquad (3.10)$$

$$L_t^r = \frac{s_t}{D_{ij}^e} + \frac{s_t}{D_j^r} + \frac{l_t}{f_j^r} \qquad (3.11)$$

3.2 Objective Function and Constrains

The main objective of offloading and resource allocation in the heterogeneous edge cloud environment is to minimize the total energy consumption ensuring that tasks are processed within allowed delays. Therefore, the objective function comprises energy and latency considering the constraints of computation resources.

$$P_t: min \sum_t (w_1 L + w_2 E + w_3 C_l) \qquad (3.12)$$

$$L = x_l^t L_t^l + x_e^t L_t^e + x_c^t L_t^r \quad (3.13)$$

$$E = x_l^t E_t^l + x_e^t E_t^e + x_c^t E_c^r \quad (3.14)$$

$$C_l = x_e^t s_t \beta - \frac{\tau}{\rho_t R} log \frac{c_{av_j} f_j^e \sigma_j}{c_j} \quad (3.15)$$

L denotes latency, E is the energy consumption, and C_l is the caching loss for computing the task (k_t). w_1, w_2, and w_3 denote latency, energy consumption, and caching loss weights. These weights can be updated based on user requirements. x_l^t, x_e^t, and x_c^t denote the offloading decision to compute locally, at the edge server, and at the remote cloud. The objective function (P_t) needs to be minimized such that

$$x_l^t + x_e^t + x_c^t = 1. \quad (3.16a)$$

$$x_l^t \in \{0,1\}. \quad (3.16b)$$

$$x_e^t \in \{0,1\}. \quad (3.16c)$$

$$x_c^t \in \{0,1\}. \quad (3.16d)$$

$$x_e^t s_t < D. \quad (3.16e)$$

Constraint (16a) shows that any task can be offloaded at only one place (local or edge or remote cloud) at any instance. Constraint (16b) indicates that the data size of the executing task at any edge server should be less than the storage capacity of the edge server. Constraints (16b)-(16e) make the optimization function an NP-hard problem. Therefore, we exploit Deep Q Reinforcement Learning (DQRL) to solve the objective function comprising energy and latency. Section IV explains the significance of DQRL [76] and how it is applied to problem $\boldsymbol{P_t}$.

3.3 Chapter Summary

This chapter presents a system model comprised of static and mobile user devices generating tasks of realistic application types along with edge servers and a remote central cloud. The communication model between different data centers is also explained with the help of mathematical equations. A novel container-based caching to store dependencies of frequently used applications is proposed. Then, the energy consumed, and latency incurred to process the generated tasks locally, at edge server, and at remote cloud are elaborated. Finally, the objective function of energy consumption, latency, and caching cost is formulated under various constraints.

Chapter 4: Proposed Methodology

This chapter introduces caching and highlights its relevance in offloading scenario and Section 4.2 details about the effect of choosing the deployment location for a network. In Section 4.6, a novel Deep Q Reinforcement Learning is proposed to solve the objective function formulated in Chapter 3. Moreover, proposed algorithms are presented in detail along with the theoretical analysis. Finally, comparison with benchmark algorithms to validate the significance of the suggested approach is also introduced.

4.1 Container-based Caching

Frequently used application setup files that can be cached in internal storage or Random-Access Memory (RAM) are defined as caching based on the access time requirements. The software bundle of dependencies and libraries required to execute an application-type task is called a container. These containers need to be downloaded every time by an edge server or a user device for running a task of an application type. Therefore, forwarding tasks frequently to a centralized cloud server can be avoided by storing popular containers locally or at an edge server. Moreover, limited storage available at the edge server can be utilized by proactive replacement of updated popular containers with outdated ones based on the frequency of requests. Therefore, container-based cache offloading minimizes data transmission and improves the quality of delivery for users.

Therefore, an optimal offloading technique to consider the benefit of caching along with energy consumption and latency is required to execute the tasks successfully in any environment. This process of deciding locations for task execution based on various constraints such as energy consumption, latency, caching is known as orchestration. The data center responsible for carrying out this process can be called an orchestrator. Along with choosing an ideal location to run a task, it is vital to choose the deployment location of an algorithm. The location of orchestrator deployment is crucial for the efficient performance of the network.

4.2 Orchestrator Deployment Analysis

The location of orchestrator deployment plays a vital role in the utilization of network resources. For instance, it costs a considerable bandwidth and delay sending the generated tasks information to a centralized orchestrator frequently. Along with the task-related information, resource availability at various data centers should also be sent, and results should be forwarded based on the network update interval. However, frequent update of network-related information increases the burden on available bandwidth and may lead to network congestion. Therefore, incorporating a distributed offloading technique may improve the overall system performance. In this, each user device chooses the execution decision of a generated task locally, decreasing the frequent transfer of task-related data local and neighboring resource availability information. In addition, available bandwidth can be utilized to send selected tasks that require edge servers and centralized cloud computation. In the considered scenario, it is assumed that network parameters are updated

globally. Therefore, every user device and datacenter have the global information about available resources and computation capacity.

4.3 Decentralized Offloading

The decentralized offloading approach is a distributed offloading technique where all the user devices decide the offloading location of a generated task locally. The algorithm is assumed to be deployed locally, and parameters used to run the algorithm are updated globally. This distributed offloading technique utilizes the global network parameters such as available bandwidth, idle edge servers, and their computation power. Rather than, sending all users information along with generated tasks to a centralized cloud, maintaining global network information decreases the load on available bandwidth. Moreover, it overcomes the problem of single point of failure in centralized offloading. This decentralized functioning decreases the frequent transfer of task-related data and local and neighboring resource availability information. In addition, available bandwidth can be utilized to send selected tasks that require edge servers and centralized cloud computation. In distributed offloading technique, the algorithm will be run locally for all user devices and a decision to process the task will be taken followed by task execution. It is assumed that weights of the DQRL algorithm will be available as global parameters and they are updated throughout the task offloading process in background. It is assumed that no additional overhead is required to run the algorithm locally and the incurred storage cost for deploying the algorithm locally is negligible with respect to the stringent service level agreements for IoT applications.

4.4 Online DQRL-based Task Scheduling and Caching Model

The variables such as available bandwidth, permitted latency, offloading location, caching decision, storage space, computation capacity at multi-level data centers (local, edge server and remote cloud) classify the chosen objective function (P_t) as NP-hard problem that cannot be addressed using linear programming. Moreover, the sequence of actions performed for the stochastic tasks further increase the number of possible solutions at any instance. Therefore, Online Deep Q Reinforcement Learning is utilized to address the problem of offloading by learning the network parameters dynamically over a period of time.

Problem P_t (Eq. 3.12) must first be transformed to the DQRL framework for applying the algorithm. The overview of proposed system model is shown in Fig. 4.1. Tasks are scheduled from various devices simultaneously, and DQRL is implemented at the orchestrator as shown in Fig. 4.2.

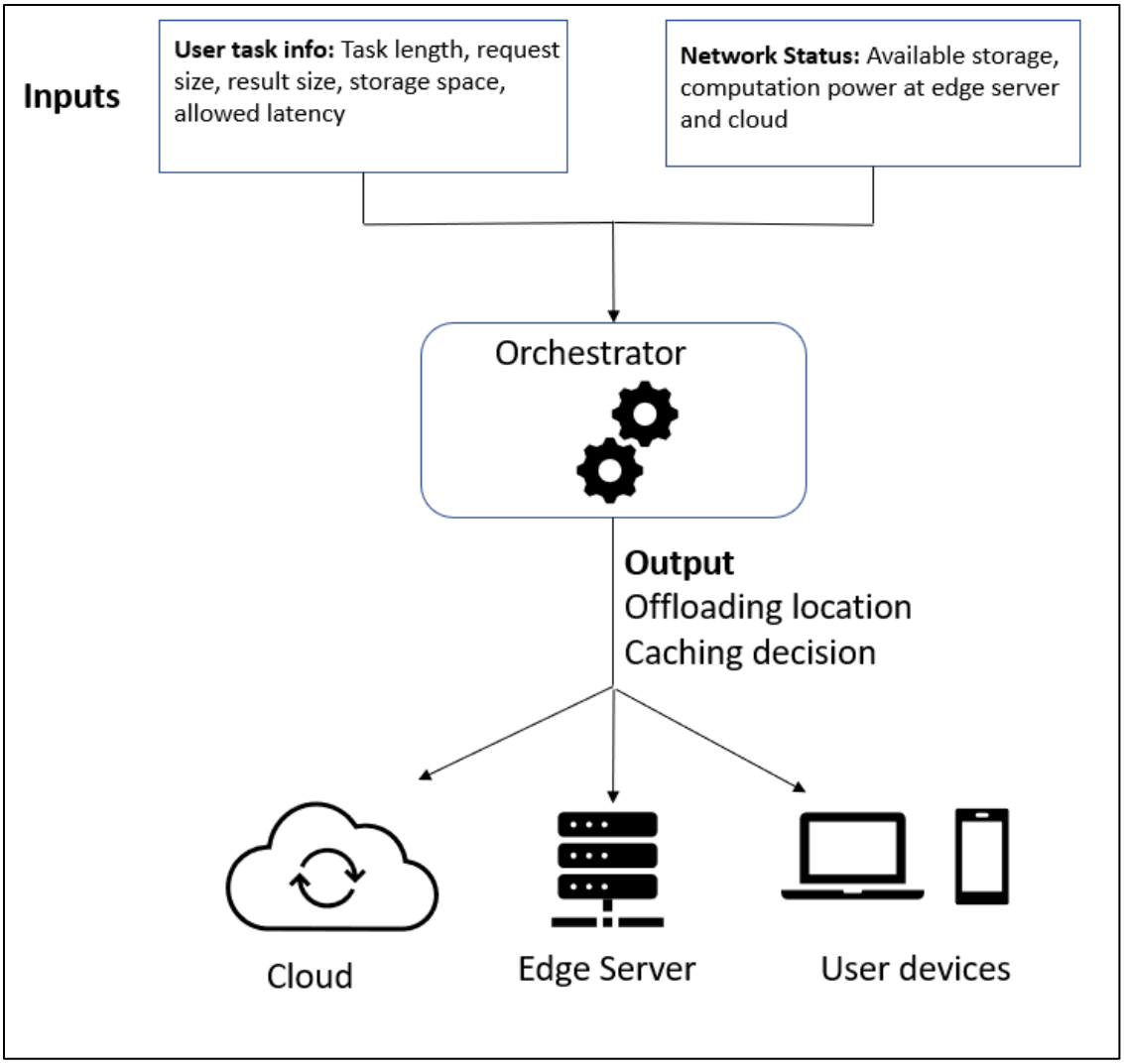

Figure 4.1: Proposed model overview

If the chosen offloading location is edge server, or remote cloud, a virtual machine is further selected to execute a task based on least waiting time to avoid concurrency. The main components of DQRL include State, Action, Reward, and Next State, which can be defined as follows:

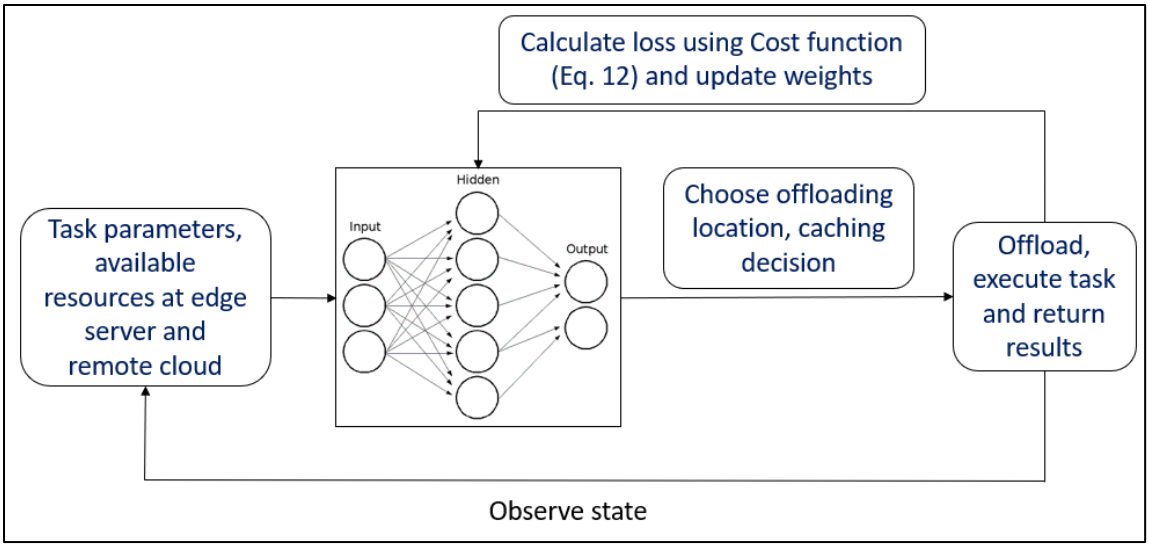

Figure 4.2: Orchestrator overview

1. **State:** The state of a system denotes the parameters that are used to define the system model. The state represents the resources available for computation and parameters associated with the task in the task scheduling problem. Based on the system model presented in Section 3, state for any task (k_i) can be given as shown below:

$$State, s_t = \{l_t, \rho_t, s_t, f_i^l, f_j^e, f_j^r\} \qquad (4.1)$$

where l_t represents length, ρ_t is the allowed latency, s_t is the input size of the task (k_t), f_i^l is the local computation rate, f_j^e is the computation rate at edge server, f_j^r is the computation rate at the remote cloud for the task (k_t).

2. **Action:** Based on the current state (s_t), the agent will select an action to be performed. In our system model, the action comprises of offloading and caching decisions as shown below:

$$Action,\ a_t = \{x_1^t, x_2^t, x_3^t, \gamma_{jt}\} \quad (4.2)$$

where x_1^t denotes local computing, x_2^t is edge computing, x_3^t is cloud computing, and γ_{jt} is caching decision at edge server.

3. **Reward:** The aim of this research is to minimize the cost function given, P_t (Eq. 3.12) . For a given state (s_t), action should be selected such that reward is maximized. Therefore, the reward for applying DQRL can be expressed in terms of cost function as shown below:

$$Reward,\ r_t = -P_t \qquad (4.3)$$

where P_t is the objective function that is to be minimized.

4. **Next State:** The next state for the system can be described as available resources after a task is offloaded, along with the following task's parameters that are waiting at the orchestrator.

$$Next\ State,\ s_{t+1} = \{l_{t+1}, \rho_{t+1}, s_{t+1}, f_{i+1}^l, f_{j+1}^e, f_{j+1}^r\} \quad (4.4)$$

4.4.1 Theoretical Analysis

In theoretical analysis, the generalization error of a neural network and complexity are analyzed. For approximating the value, we represent DQRL using a neural network containing two hidden layers as shown below:

$$Q(\mathrm{s},\mathrm{a};\mathrm{w}_1,\mathrm{w}_2) = \frac{1}{\sqrt{\mathrm{x}}}\sum_{\mathrm{i}=1}^{\mathrm{x}} \mathrm{w}_{2_\mathrm{i}} \mathrm{f}(\mathrm{w}_{1_\mathrm{i}}.(\mathrm{s},\mathrm{a})) \quad (4.5)$$

where x is the number of neurons, $w_1\ and\ w_2$ represent weights at layers 1 and 2, respectively. At any iteration k, minimizing a function of a neural network can be expressed as

$$min_{w_1,w_2} \frac{1}{2n}\sum_{j=1}^{n}(r_j + \gamma max_a Q_k^*(s,a) - Q_k(s,a;w_1,w_2))^2 \qquad (4.6)$$

Eq. (4.5) can be considered as an overparameterized neural network that outputs the global minima of an empirical function as mentioned in [77]. The generalization error of this network can be approximated as $1/\sqrt{n}$ by projecting it on gradient descent where n is the number of samples.

For calculating the complexity in the given heterogeneous simulation environment, three offloading scenarios are considered: local, edge server, and remote cloud execution. It is also decided if a task needs to be cached along with the offloading decision. Therefore, the complexity [77] in the offline computation of the algorithm can be expressed as $O(3^n 2^n)$, which is very large and incurs enormous computation to obtain decent performance. Along with that, weights of the network must be updated based on the simulation scenario and require frequent training. Instead, online DQRL can give a near-optimal solution with less complexity ($O(4)$) and can be implemented for many computation tasks. The algorithm

self-learns the environment and updates the weights based on reward/penalty obtained for offloading every task and needs minor maintenance.

4.5 Proposed Online DQRL Solution

DQRL uses train network, target network, and replay memory to choose to offload and cache decisions for various scheduled tasks. The train network is used to decide the actions to be taken, and the target network is used to correct the train network's weights for better performance. Replay memory is used to store the transitions (s_t, a_t, r_t, s_{t+1}). Replay memory shuffles the transitions and feeds the neural network after every time period such that the network's weights are not affected by the correlation between subsequent states. The concept of DQRL is illustrated in Fig. 4.3 for better understanding.

Train and Target networks are first initialized with random weights, and the Train network calculates Q-values for all the possible actions from the current state. The action corresponding to the maximum Q-value is performed, and the next state is reached. Based on the actual Q-value from the target network combined with obtained reward, weights of the train network are updated, and the transition is added to Experience Replay memory. A similar process is repeated for all the scheduled tasks, and the entire simulation is run for multiple epochs. Experience Replay is used to train the network after every set of iterations to avoid the impact of recent transitions and generalize the network weights. Weights of train networks are copied to the target network after every epoch to have

network stability. A detailed explanation of the proposed DQRL can be seen in Algorithm 2.

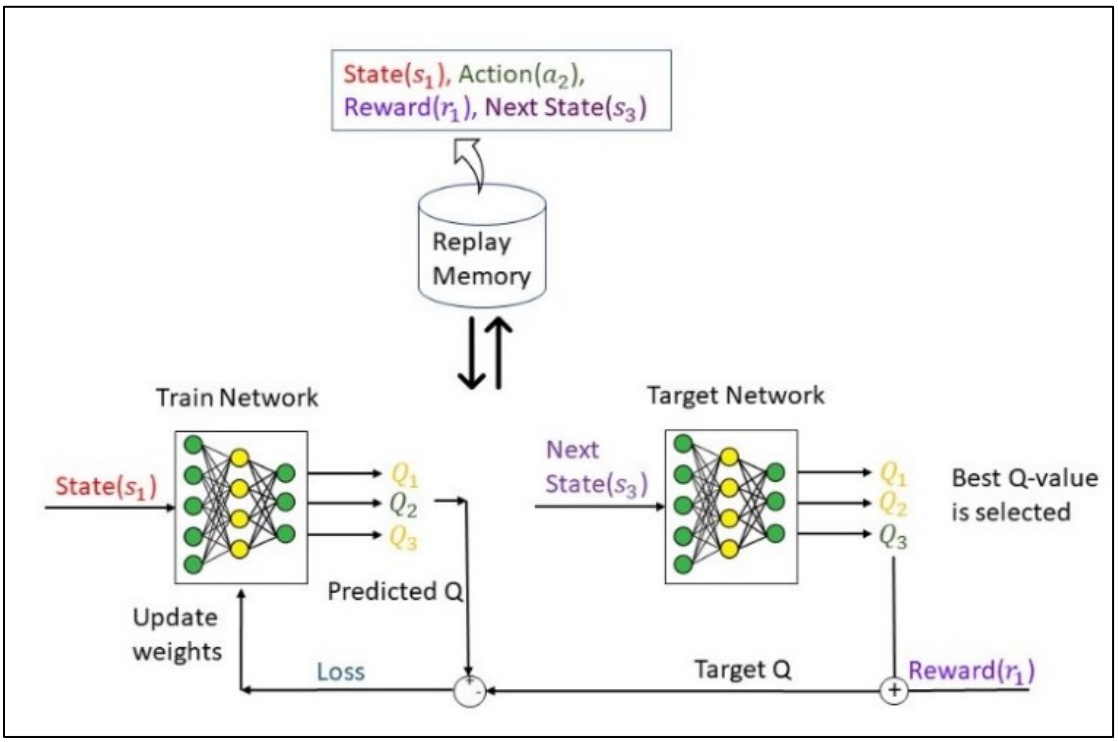

Figure 4.3: Online Deep Q Reinforcement Learning

A mobility pattern is added to the user devices to replicate a realistic environment, and virtual machines are created at the edge server and cloud. Therefore, at any instance, if the selected action is offloading at the edge server or remote cloud, then a virtual machine is also selected based on the number of tasks waiting at different virtual machines [78]. It is again validated that the user device is in the coverage radius of the edge server while offloading the results. Algorithm 3 provides a detailed sequence of steps involved: user devices, data centers, and task generation; task offloading and execution; network updating.

Algorithm 2: Proposed DQRL

1. Initialize N, Train network (Q_{tr}) and Target network (Q_{ta}) with pre-defined weights
2. Initialize Replay memory
3. for each simulation, do
4. Initialize the environment and start task scheduling
5. for each task, do
6. Find action (a_t) to perform using Train network (Q_{tr}) and execute it.
7. Calculate reward (r_t) using Eq.4.3 and update to next state (s_{t+1})
8. Store transition (s_t, a_t, r_t, s_{t+1}) into Replay memory.
9. After every N tasks
10. sample a mini-batch from Replay memory.
11. Find predicted and target actions for mini-batch using Train (Q_{tr}) and Target (Q_{ta}) networks respectively.
12. Update Train (Q_{tr}) network weights based on calculated loss.
13. After every 2N tasks
14. Copy Train (Q_{tr}) weights to Target (Q_{ta}) network.
15. end for
16. end for

Algorithm 3: Offloading decisions using DQRL

1. Initialize number of user devices, simulation time, simulation area, WAN and LAN bandwidth, number of users, edge server coverage radius, and network update interval
2. for each simulation, do
3. Generate user devices, edge servers, remote cloud datacenters. Virtual machines are created for edge servers, remote cloud, and user devices based on their computing resources availability.
4. Random mobility pattern is assigned for all user devices.
5. Tasks are generated for all user devices, and a random time is assigned for scheduling. All tasks are scheduled to be dispatched at assigned time slots.
6. for each task, do
7. Find offloading location and caching decision using Algorithm 1 and send the task.
8. If offloading location is edge server or remote cloud,
9. find a virtual machine at which the least number of tasks are waiting for execution.
10. if the task is not cached,
13. download the container
14. Execute the task and check if the task failed due to long delay, mobility, battery power. Increment tasks failed accordingly.
15. Send the results to the user device after validating the task failures.
11. Update energy consumption, CPU utilization, network model based on offloading location.

12. end for

13. end for

4.6 Benchmark Policies

To validate the performance of the proposed Online DQRL algorithm, a number of benchmarking policies are implemented as shown below:

1. **Round Robin:** This is used in many laptops, tablets, and other electronic gadgets to queue multiple applications requesting computation resources. Whenever a task arrives, this policy iterates through all the available virtual machines to choose one that has the least waiting time for execution [79]. As this policy checks all available machines and the order is O(n), it is well-suited for small-scale applications only. Nevertheless, the policy does not consider the specification of the task into account to choose an offloading location.
2. **Trade-Off:** This is a custom-designed policy to choose execution location amongst available local server, edge server, and remote cloud by assigning weights. Pre-determined weights are assigned to all available computation locations based on the delay to reach them and the incurred energy. Then, the policy chooses the offloading location with the least weight and execution delay [79]. However, this algorithm only captures the features of edge servers and centralized cloud to decide the execution location of any task.

3. **Fuzzy Decision Tree (FDT):** Fuzzy decision tree is a popular technique to solve problems involving numerical and categorical data. They use a pre-defined set of rules to decide the offloading location for any task [80]. In this scenario, a task can be classified into three levels based on the acceptable latency, length (in MI), request size. Based on the category of any task, the execution location is decided. However, frequent supervision is required to change the boundaries for task classification based on the available resources in the network and the types of generated tasks. Therefore, applying this technique in a dynamic network can be tedious.
4. **Simulated Annealing:** This heuristic approach works using probability techniques to find a global optimum. In a task offloading scenario, simulated annealing maps different tasks to the available virtual machines to minimize the objective function [79]. However, this algorithm requires information of upcoming tasks beforehand, or the algorithm will have to wait until the generation of all tasks for a chosen time. This mapping between tasks and hosts has to run for several epochs to perform well.

Moreover, the performance of the proposed Online DQRL is further enhanced by incorporating container-based caching. The performance of the suggested method can be validated by implementing Task-based caching as explained below:

Task-based Caching: For any generated task from an application, storing input and output data (results) can be termed task-based caching. This can be beneficial only when users

query with or request comparable data. However, this is an uncommon scenario as different users may use a similar application but not request similar data. For instance, a building with face recognition authorization may use face recognition application frequently, but the input face to match from a database often differs.

Along with an ideal framework, the deployment location of the algorithm is also crucial in overall system performance and utilization of resources. For this reason, a distributed offloading technique where all users decide the offloading location of a generated task locally is recommended. The impact of orchestrator deployment is examined using the following benchmark policies:

1. **Centralized Offloading:** In this, the centralized cloud determines the execution location for all generated tasks in a network. However, the centralized orchestration paradigm requires the global tasks information, available storage space, computation power, battery levels of user devices, network bandwidth to run the offloading policy periodically. This complete information should be communicated frequently to continue orchestration. However, frequent network information updates increase the burden on available bandwidth and delay processing the task. Moreover, the variations in resource availability between the network update intervals will be ignored and affect the estimated processing time of tasks.
2. **Cluster-based Offloading:** In this approach, a group of user devices forms a cluster based on their location and mobility speed. For every cluster, a cluster-head is selected based on the remaining battery, computation power, number of neighbors. The designed offloading algorithm will be deployed at the cluster head.

Therefore, the cluster head decides the execution location of various tasks and schedules the data transmission. Nevertheless, choosing cluster-head increases the delay of the whole process, and the mobility of any device may disrupt the orchestration and data transfer.

4.7 Chapter Summary

This chapter starts with commonly used deep learning models and reinforcement learning techniques. Owing to the complexity of the considered model, Deep Reinforcement learning, a combination of neural networks and reinforcement learning, was presented. Later, this chapter presents the primary reasons for using deep reinforcement learning for task offloading scenarios and theoretical analysis. The proposed Online Deep Q Reinforcement Learning is detailed mathematically. Moreover, algorithms are provided to demonstrate how the proposed framework can be applied to address the objective function. Finally, benchmark policies were introduced to validate the performance of Decentralized Online DQRL along with container caching. In Chapter 5, the recommended approach will be examined by comparing with benchmark algorithms using extensive simulations.

Chapter 5: Simulation Experiments

In this chapter, PureEdgeSim [79], a Java-based tool, is used to conduct simulation experiments. PureEdgeSim is a discrete event simulator that has the ability to run parallel processes of resource management. Cloud-based services ranging from Infrastructure as a Service (IaaS) to Software as a Service (SaaS) can be implemented using this simulator. The proposed Online DQRL is implemented in the PureEdgeSim using Java and evaluated on realistic profiles of IoT applications and compared to various benchmarking policies introduced in Section 4.4.

5.1 Simulation Setup

We consider a simulation environment with a centralized cloud, multiple edge servers, and user devices such as smartphones, laptops, sensors, and raspberry pi to evaluate the proposed approach. The specifications of user devices and data centers are listed in Tables 5.1-5.3.

Table 5.1: User device specifications

Device type	Mips (Million Instructions per second)	RAM (in GB)	Storage (in GB)
Smart phone	25000	4	128
Raspberry Pi	16000	4	32
Laptop	110000	8	1024
Sensor	70000	4	0

Table 5.2: Edge Server specifications

Edge Server specifications	Host 1		Host 2	
	Virtual Machine 1	Virtual Machine 2	Virtual Machine 1	Virtual Machine 2
Mips (Million instructions/sec)	200000	200000	200000	200000
RAM (in GB)	4	4	4	4
Storage (in GB)	20	20	20	20

Table 5.3: Central cloud specifications

Centralized Cloud specifications	
Hosts	2
Virtual machines/Host	8
Mips of each VM	250000
RAM (in GB)	16
Storage (in GB)	20

In addition to this, smartphones and laptops are battery-powered devices with a capacity of 18.75- and 56.2-Watt-hours, respectively. Raspberry pi and IoT sensors are hardwired devices that work using a power supply. Tasks will be generated after random unequal intervals at a rate of 7 per minute simultaneously from multiple user devices. The generated applications from these devices include Augmented reality, e-Health, heavy computation infotainment, and specification of application types mentioned in Table 5.4. The sensitivity of applications with respect to latency is also included in Table 5.4. The available bandwidth for wireless Local Area Network (WLAN) and Wide Area Network (WAN) is

considered 300 Megabits per second. The available bandwidth will be updated periodically based on the number of tasks utilizing the network. The WAN propagation delay is assumed as 0.2 seconds. The computation speed of a data center is measured in Million Instructions per second (Mips). The total designed simulation area is 200x200 meters to place edge servers, remote cloud, and user devices. The coverage radius in which the user device can communicate to the edge server is assumed to be 20 meters, and the edge server can cover a radius of 200 meters. Energy consumed for every transmitted or received bit, amplifier energy dissipation in free space, amplifier energy dissipation in multipath is considered as $5x10^{-8}$ Joules/bit, $1x10^{-11}$ Joules m^2/bit, $13x10^{-16}$ Joules m^4/bit, respectively.

Table 5.4: Application types

Application	**Latency Sensitivity (%)**	**Request size (in KB)**	**Result size (in KB)**	**Container size (in KB)**	**Task length (MI)**
Augmented Reality	98	1500	100	25000	60000
e-Health	5	10000	10000	13000	300000
Infotainment	98	50	50	9000	15000

The user devices will start from a random initial point within the simulation area and move with a speed of 1.4 meters/second. Available resources at multi-layer network servers such as storage space, battery power, MIPS, and network parameters such as bandwidth will be updated simultaneously based on the offloading decisions. Except for the sensor, all other devices can execute tasks locally.

Once the simulation starts, user devices comprising of smartphone, laptop, raspberry pi, and a sensor type will be generated. The application types listed in Table 5.4 will be scheduled to be generated from the user devices at a random time slot within the provided simulation time. A random second in every minute is chosen to schedule a task. It is maintained that seven tasks are generated per minute from each device. Our proposed DQRL algorithm finds the offloading decision for various devices' generated tasks. The simulation model is run for three iterations starting from 100 to 300 devices, and the simulation time for each iteration is 10 minutes. Simulation time does not include the time allocated for generating the required resources. DQRL will decide the offloading decision for all tasks, and based on the decision, the task is executed, and results are returned to the user.

5.2 Results and Analysis

Initially, the proposed Online DQRL is run without caching and performance is compared against four other benchmarking algorithms mentioned in Section 4.4 to have a fair comparison. Tasks of mentioned application types in Table 5.4 are generated from multiple static and mobile users' devices and are offloaded using Online DQRL.

5.2.1 Online DQRL performance evaluation

Fig. 5.1 shows the number of tasks that failed to be executed on time due to their mobility. At any instance, if a task corresponding to a user is getting executed at edge server and if

the user goes out of range, the task execution is disrupted. The task needs to be re-executed in this case and the latency for processing increases. If the increased latency fails to meet the requirements, then it is considered that task failed to be executed on time due to mobility, and it can be observed that DQRL and Fuzzy decision tree outperformed other algorithms. Though FDT had no failures due to mobility, at least 30% of the tasks could not execute on time because of the pre-defined rules that could not decide the offloading decision in a dynamic environment. The poor performance of FDT can be observed in Fig. 5.2, which shows the number of tasks that failed to be executed due to delay.

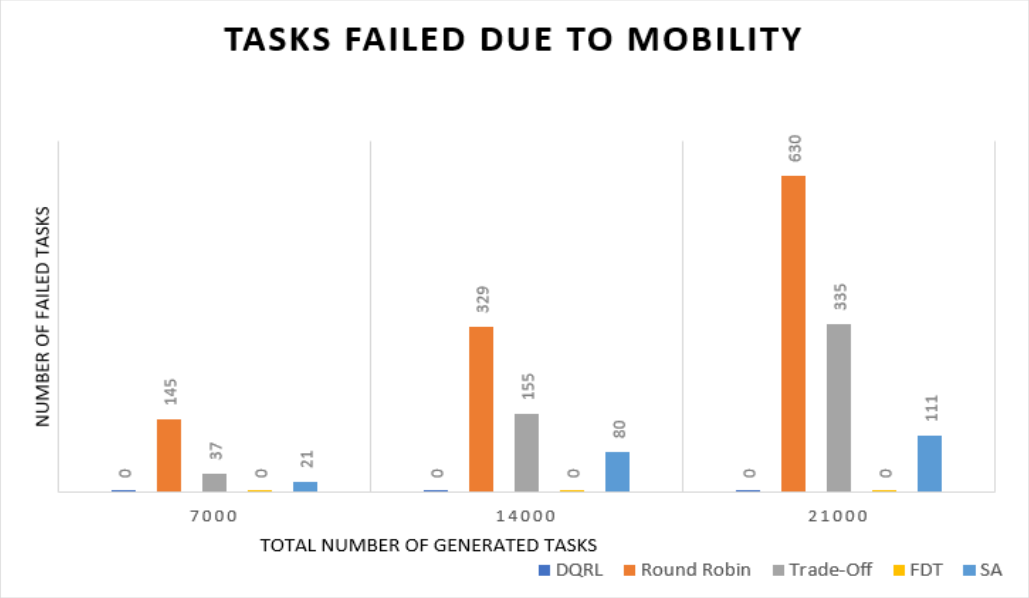

Figure 5.1: Task failures due to mobility

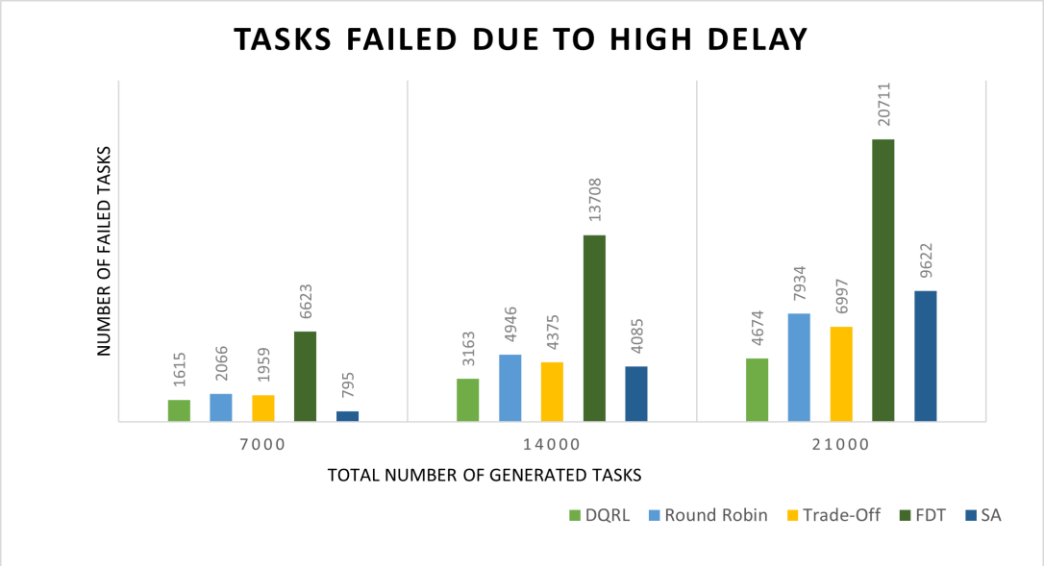

Figure 5.2: Task failures due to delay

It can also be observed that DQRL outperformed other algorithms and reduced the task failures to at most 22% as shown in Fig. 5.2. The undesirable performance of Round Robin and Trade-off can be attributed to their complexity of finding an offloading location. They iterate through the entire list of resources, including the user devices, to decide an offloading location. Though SA performed better than DQRL when the number of devices was 100, SA demands that prior information about upcoming tasks should be noted, and this is not the ideal case in a realistic environment. As SA uses pre-defined mapping to offload tasks and does not consider the network dynamics, efficient resource management may not happen, causing delayed execution times and may lead to task failures. This can be observed when the number of devices increases in Fig. 5.2.

Fig. 5.3 shows the average execution delay for tasks under different algorithms. Average execution delay is the time taken to compute the task at a virtual machine, and we can see

that the Trade-Off algorithm outperformed DQRL by 25% for multiple iterations. This is because the Trade-Off algorithm iterates through the entire list of virtual machines and finds an offloading virtual machine with the slightest execution delay. Iterating through the entire list for every task increases the complexity of the algorithm and results in more time to find an offloading decision as devices increase. Therefore, the Trade-Off objective function minimizes the execution delay and does not consider energy consumption.

Since Round Robin and Trade-Off algorithms run through the entire list and choose a location one by one successively, they are more well-suited to validate DQRL performance. Therefore, we continue our analysis of DQRL over Round Robin and Trade-Off algorithms in the following pages.

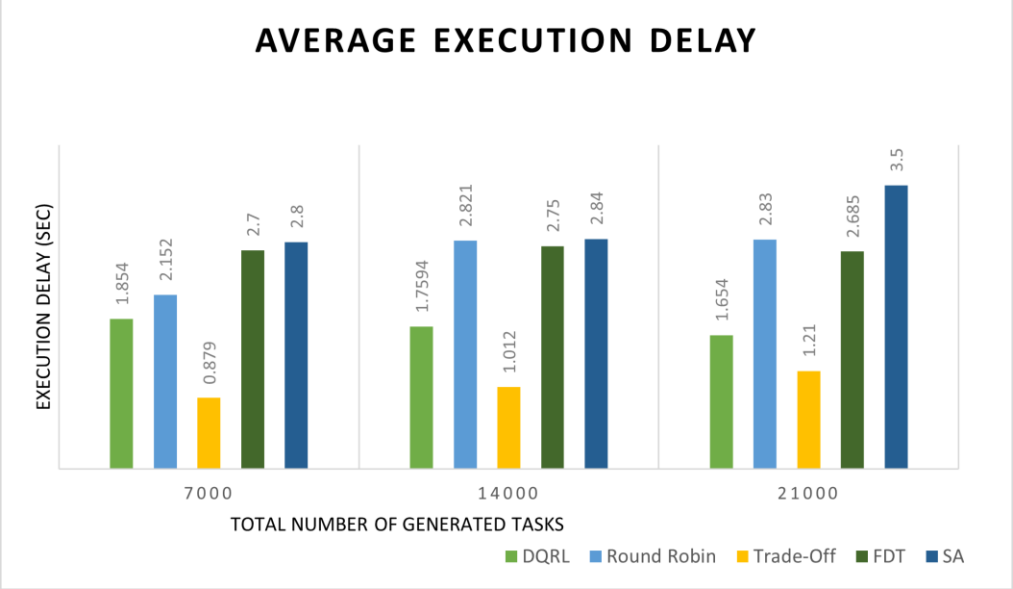

Figure 5.3: Average execution delay

The efficient management of resources available at datacenters and network bandwidth by DQRL has reflected in Fig. 5.4, which plots average energy consumption. The proposed online DQRL outperforms Round-Robin and Trade-off by optimizing the energy consumption by at least 30% while deciding the offloading location.

From Fig. 5.5, we can see that the average bandwidth utilized for offloading a task in DQRL is less than the Round Robin and Trade-Off algorithms. DQRL utilized edge server resources efficiently and offloaded less than 5% of tasks to the remote cloud. Since the Trade-off algorithm minimizes weighted execution time, the bandwidth utilized is more diminutive than Round Robin. However, the performance is insignificant as the weights are constants based on energy consumption and are not updated periodically.

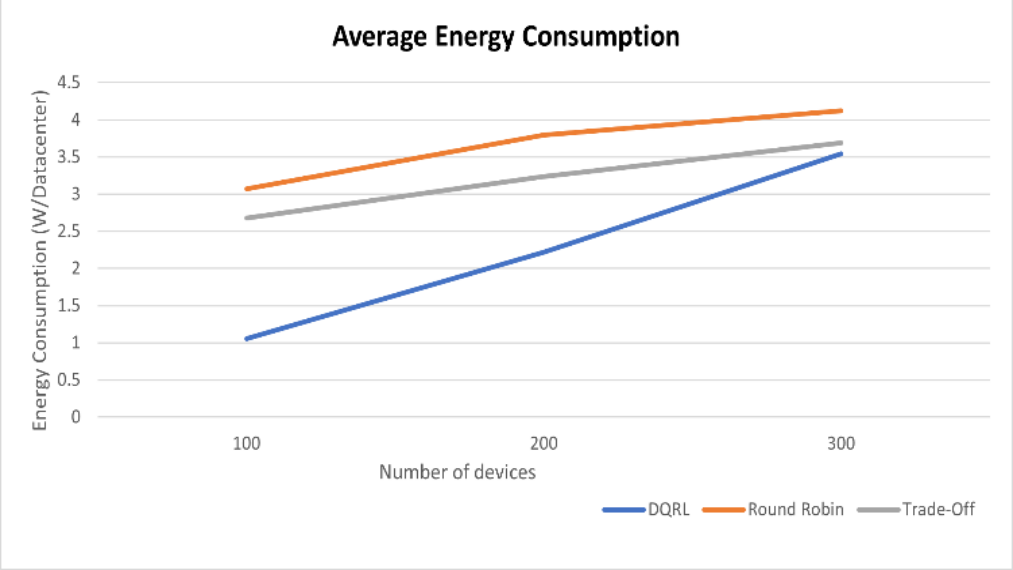

Figure 5.4: Average Energy Consumption

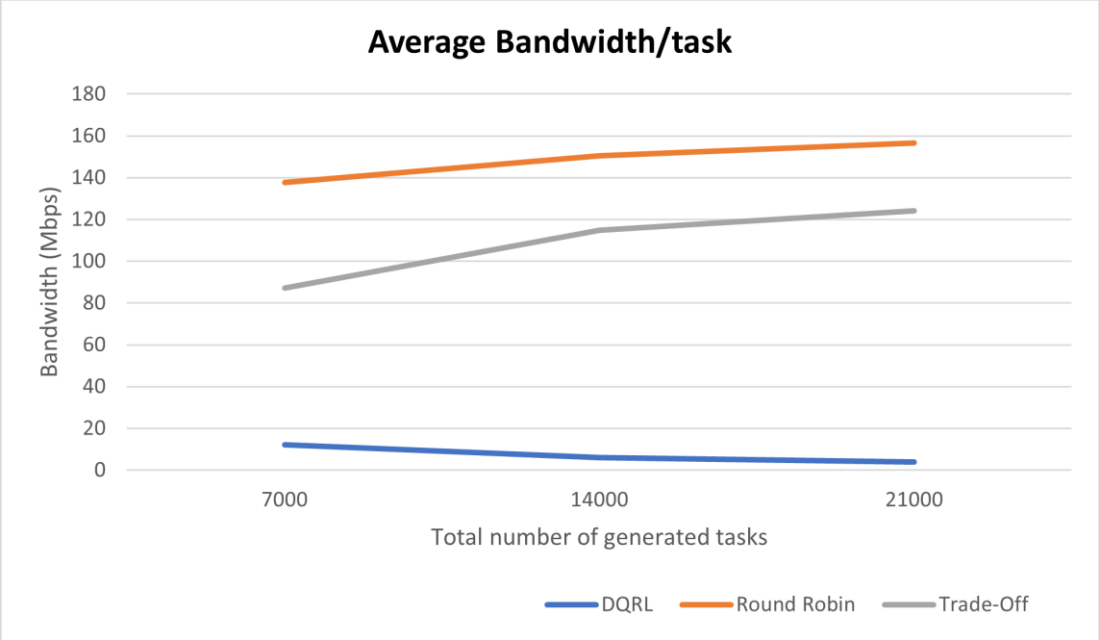

Figure 5.5: Average bandwidth

The effect of not considering the dynamics of the environment and not taking energy consumption into account can be seen in Network traffic shown in Fig. 5.6. Though the number of devices has increased, network traffic did not increase much when DQRL was used. Nevertheless, for Round Robin and Trade-Off, there is a linear relationship between Network traffic and the number of devices. Network traffic plays a vital role in accommodating devices and needs to be minimized as much as possible in any scenario.

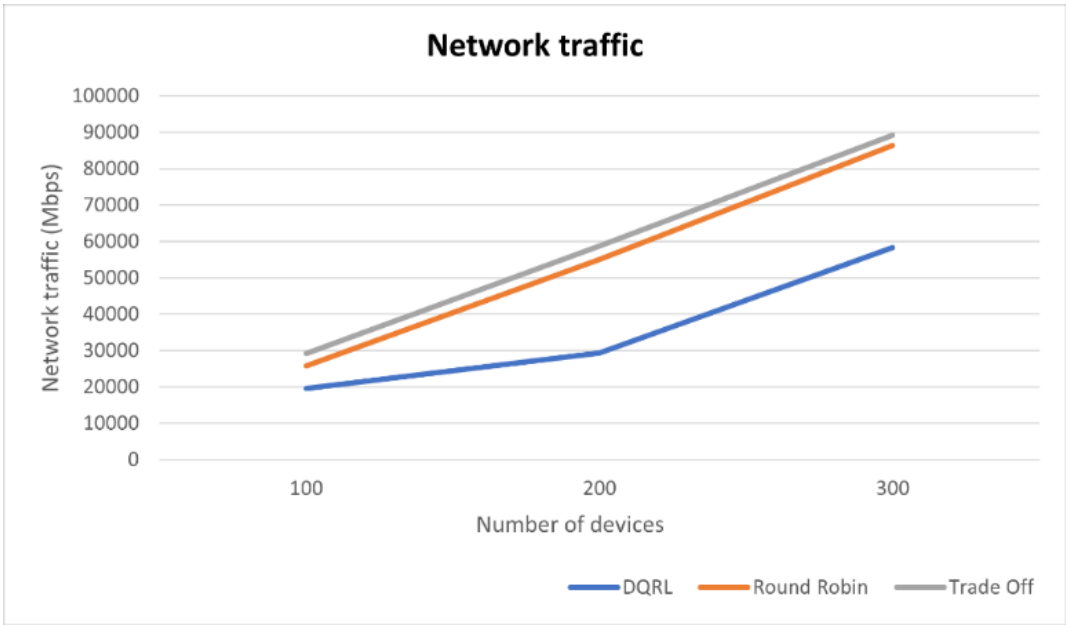

Figure 5.6: Network traffic

Since the proposed algorithm aims to minimize the energy consumption of user devices, devices must not run out of battery power due to local computation. Fig. 5.7 shows the remaining power for the device after the simulation is completed. It can be observed that DQRL executes tasks that require the least computation resources locally. As a result, drained battery power is negligible when the DQRL algorithm is implemented. As Round Robin and Trade-Off algorithm focuses only on execution delay and does not consider features of a task into account, the average remaining power is less.

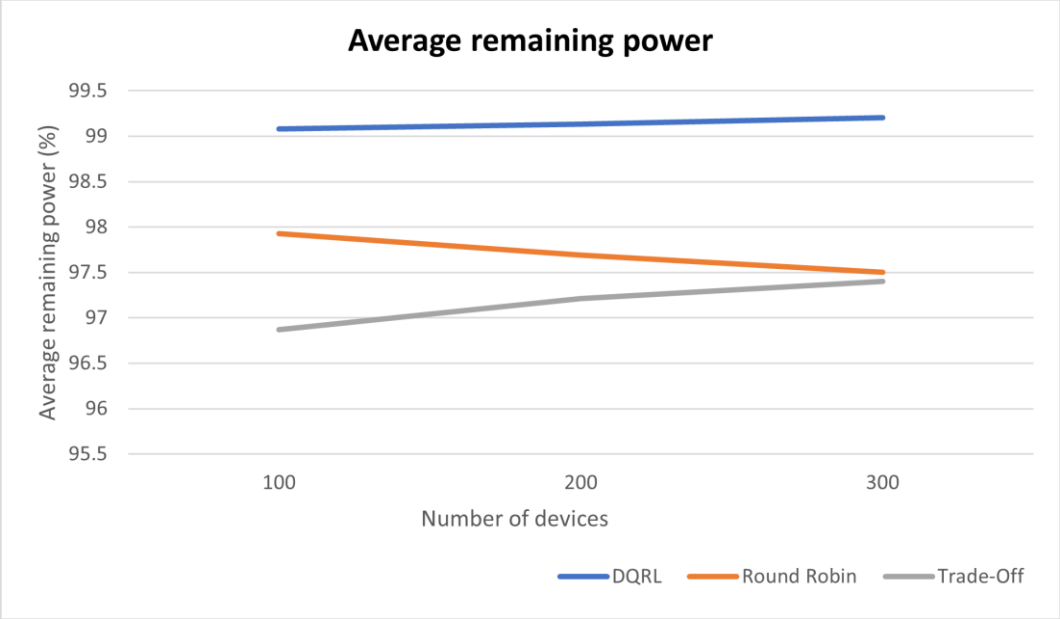

Figure 5.7: Average remaining power

We can observe that the proposed DQRL performs well in offloading tasks despite addressing dynamic contexts involving continuous changes in system workloads. The performance of the considered framework can be further enhanced by caching the containers of popular application types.

5.2.2 Container-based caching performance evaluation

Proposed container-based caching was run using the provided simulation setup and is plotted against task-based caching in Fig. 5.8 for a reasonable comparison. Fig. 5.8 shows the number of tasks that could not be executed on time, and it can be noticed that the failures increase when task-based caching is considered. This is because querying identical tasks is less likely to occur in a realistic scenario, and therefore, cached data remains unused. As a result, caching tasks do not significantly reduce the overall delay compared

to container caching. Container-based caching outperforms task-based caching by more than 50%. Furthermore, caching the setup files of popular application types eliminates repeated container downloading, minimizing the delay, and resulting in fewer task failures (at most 10%), as shown in Fig. 5.8. This can also be observed in average energy consumption, as shown in Fig. 5.9.

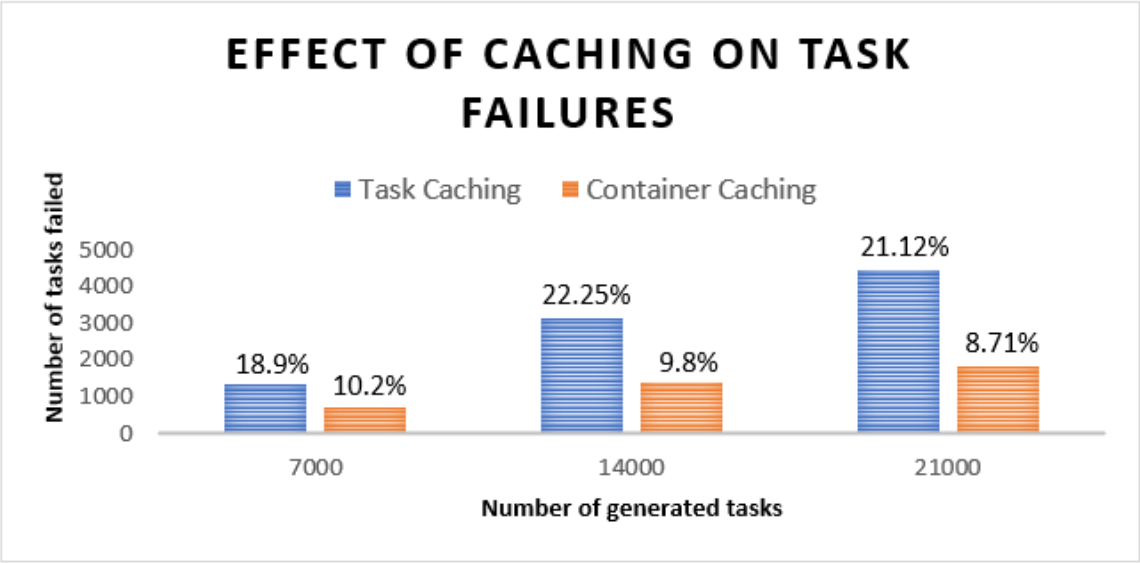

Figure 5.8: Effect of caching on task failures

From Fig. 5.9, it can be observed that average energy consumption per data center is 60-80% less when the frequently accessed application containers are cached. This is because it costs much energy to download the entire setup each time for every new task of an application type. Therefore, caching can eliminate the repeated process and contribute to considerable energy consumption savings. On the other hand, caching a task whose repetition is not so often does not lead to significant energy savings, and this can be noticed in energy consumption for caching a task.

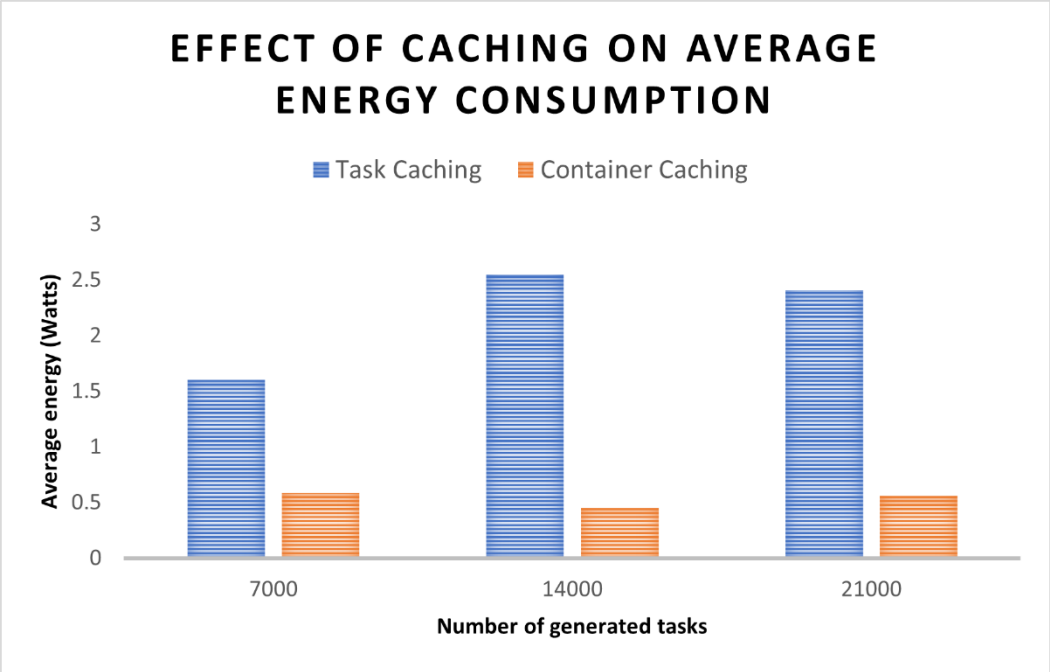

Figure 5.9: Effect of caching on average energy consumption

Therefore, it can be verified that container caching can further improve the task success rate compared to task caching. Furthermore, the significance of orchestrator deployment is also examined with the help of benchmarking policies listed in Section 4.4.

5.2.3 Decentralized offloading performance evaluation

In Fig. 5.10, the number of tasks that fail to be executed on time is plotted when decentralized, centralized, and clustering-based offloading methods are used. The suggested three methods are run with and without container caching to understand the impact of deployment location better and caching. We can observe that failures have been reduced by at least 13% in all three orchestrations when containers are cached, and the decentralized offloading technique had the least number of task failures. The primary

reason is that it eliminates the frequent communications of generated tasks data and network information.

Also, the higher number of failures (at least 25% with cache) in the cluster-based offloading approach is due to users' mobility. In cluster-based offloading, a cluster-head amongst user devices has to be selected for algorithm deployment after every time interval. As user devices are subject to mobility, unexpected movement in user devices leads to disruption in task execution for the entire cluster.

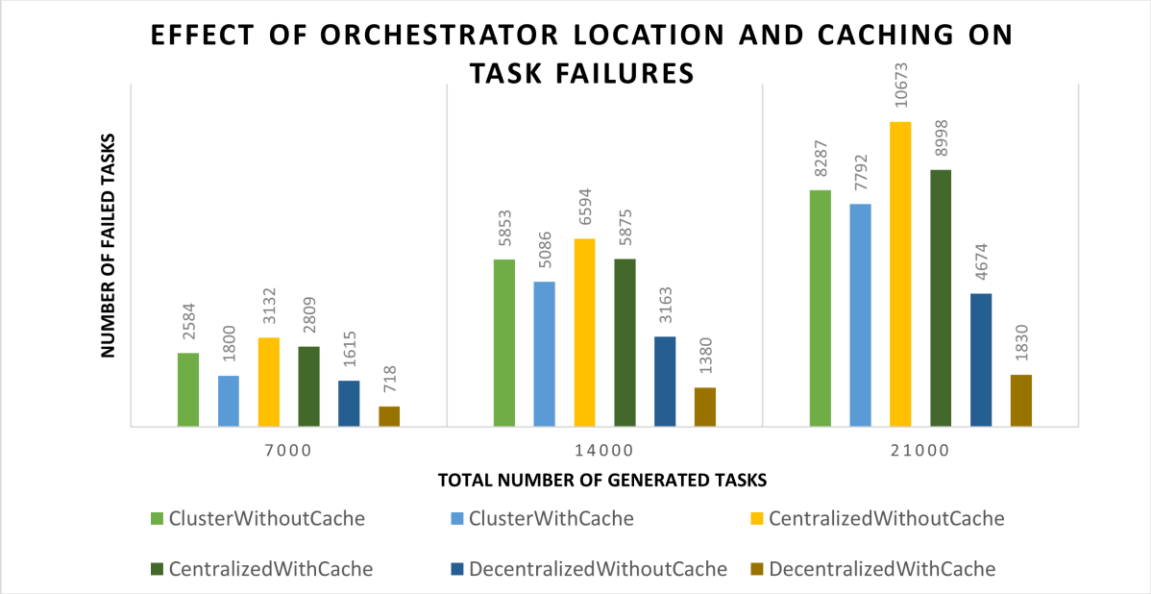

Figure 5.10: Effect of orchestrator location and caching on task failures

Fig. 5.11 shows the average energy consumption for different orchestration types, and we can observe that Decentralized offloading when containers are cached consumes a minimum of 25% less energy when containers are cached. The considerable reduction in

energy consumption can be attributed to independent and distributed algorithm modeling in the proposed offloading technique.

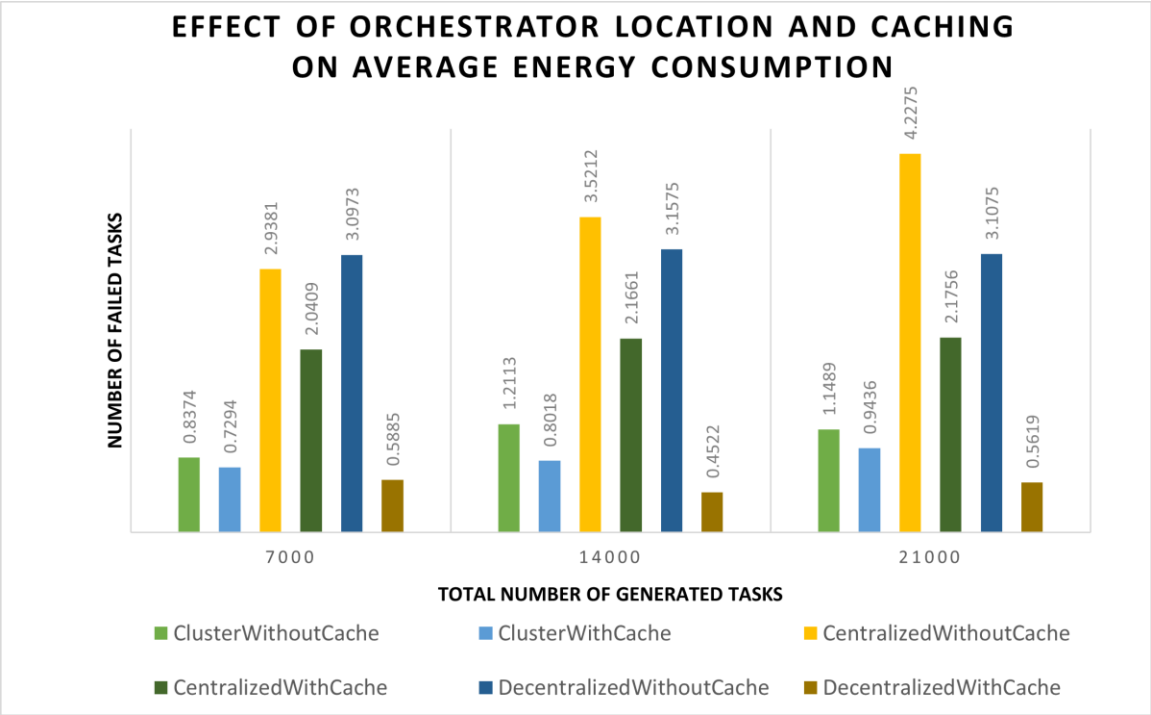

Figure 5.11: Effect of orchestration location and caching on average energy consumption

Fig. 5.12 plots the network traffic with the number of tasks that failed due to delay for all the benchmark algorithms to understand the impact better. The total number of generated tasks was 7000 in this case. It can be observed that Decentralized offloading with container-caching outperformed all other algorithms, and it can also be noted that at least 47% less network traffic was utilized with a reduction in number of task failures (at least 13%). The impact of caching can be seen by comparing network traffic usage with Decentralized offloading without caching. As caching eliminates repeated download of pre-requisite setup files, a large amount of bandwidth can be saved.

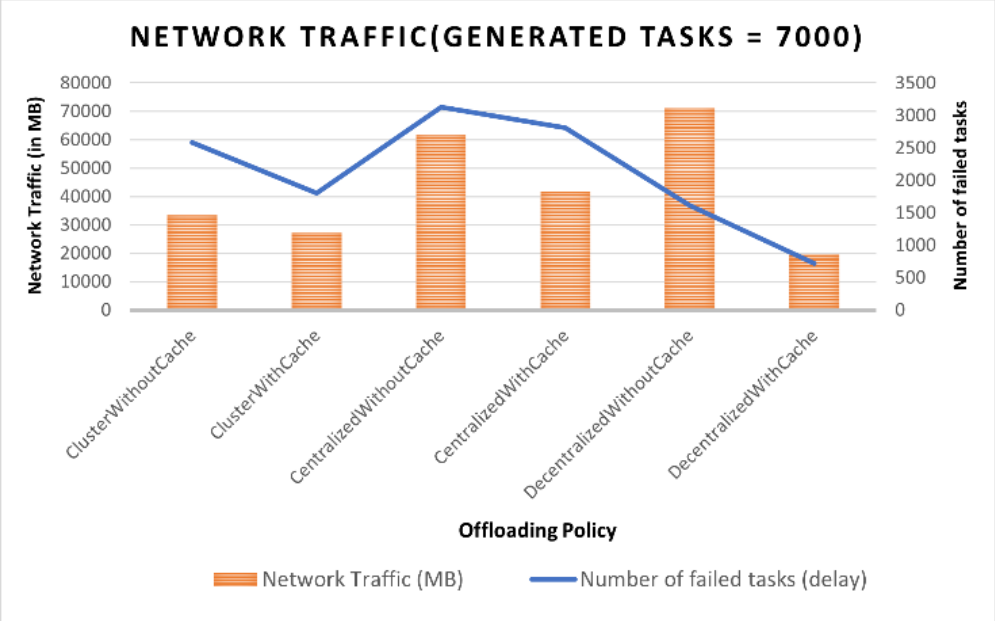

Figure 5.12: Network traffic when 7000 tasks are generated

Similar performance can be observed when the number of generated tasks is increased to 14000 and 21000, as shown in Fig. 5.13-5.14. The network traffic utilized is at least 57% less when compared to other benchmark policies besides reducing the task failures by more than 25%.

It is also essential not to overuse local resources for battery-powered devices. Fig. 5.15 shows the comparison of remaining battery power in user devices with respect to the number of successfully executed tasks.

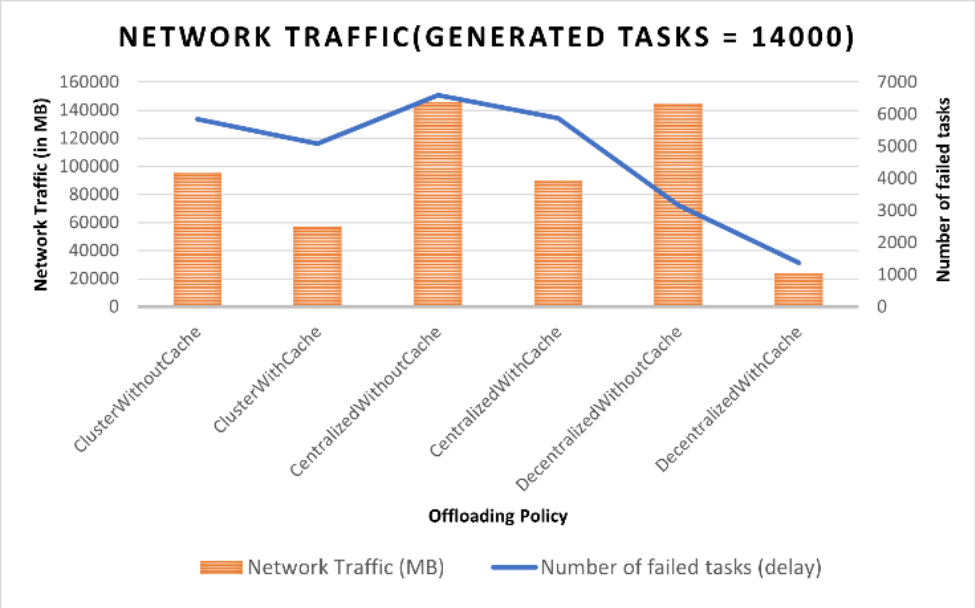

Figure 5.13: Network traffic when 14000 tasks are generated

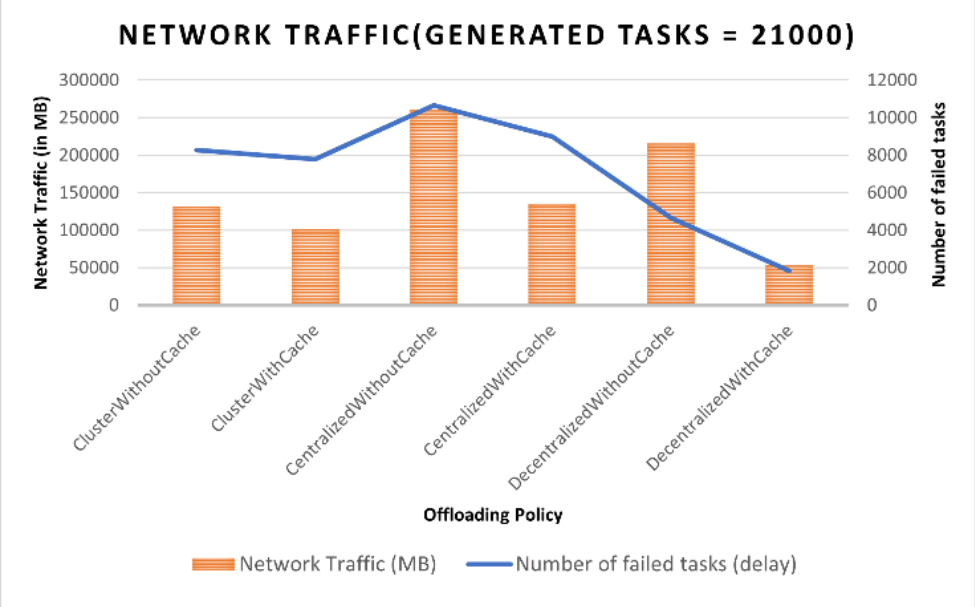

Figure 5.14: Network traffic when 21000 tasks are generated

It can be observed from Fig. 5.15 that the remaining battery power is highest for decentralized offloading without caching the container when compared to decentralized

offloading with the cache. However, it should also be noted that the number of successfully executed tasks is higher when the container is cached. The overall network performance depends on utilizing the available resources optimally and on maximizing the number of successful tasks. Similar behavior can be observed when the number of generated tasks is increased to 14000, 21000 and are illustrated in Fig. 5.16-17 respectively.

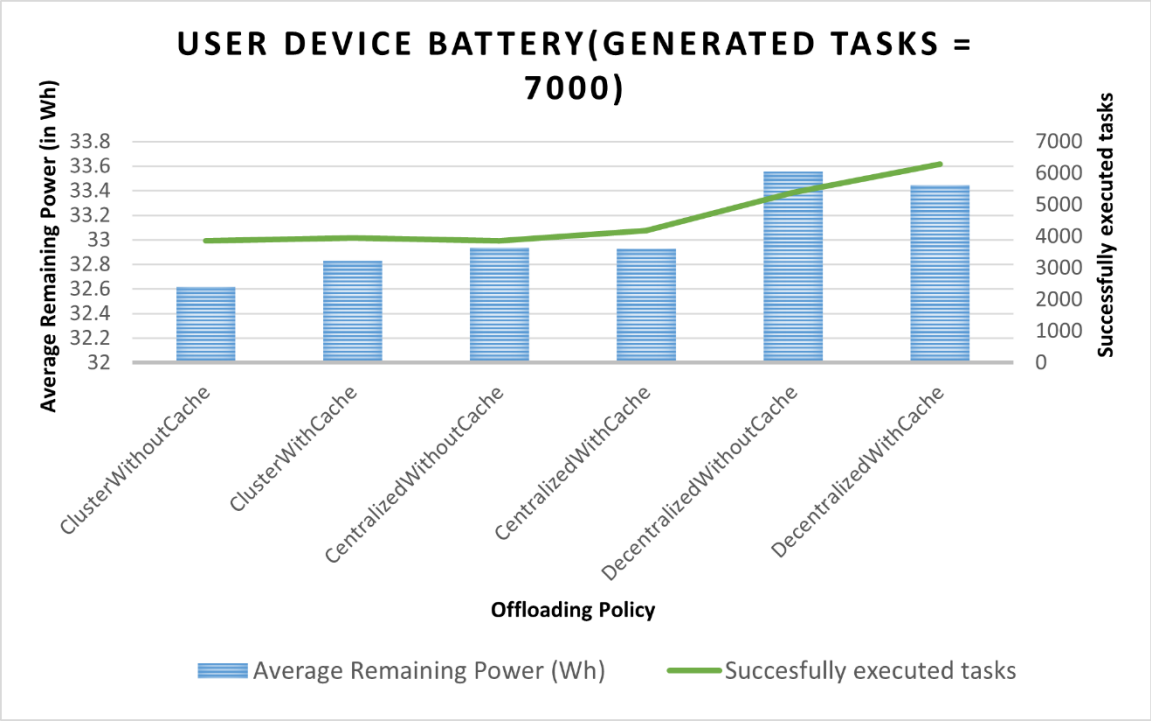

Figure 5.15: User device battery levels when 7000 tasks are generated

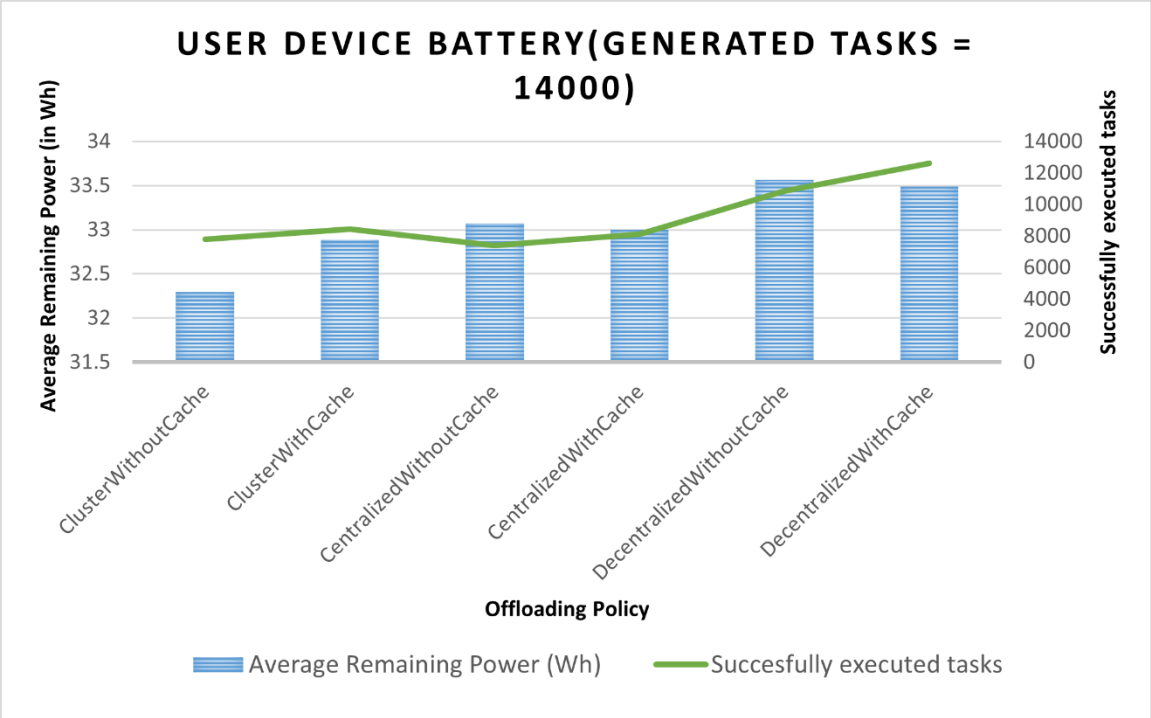

Figure 5.16: User device battery levels when 14000 tasks are generated

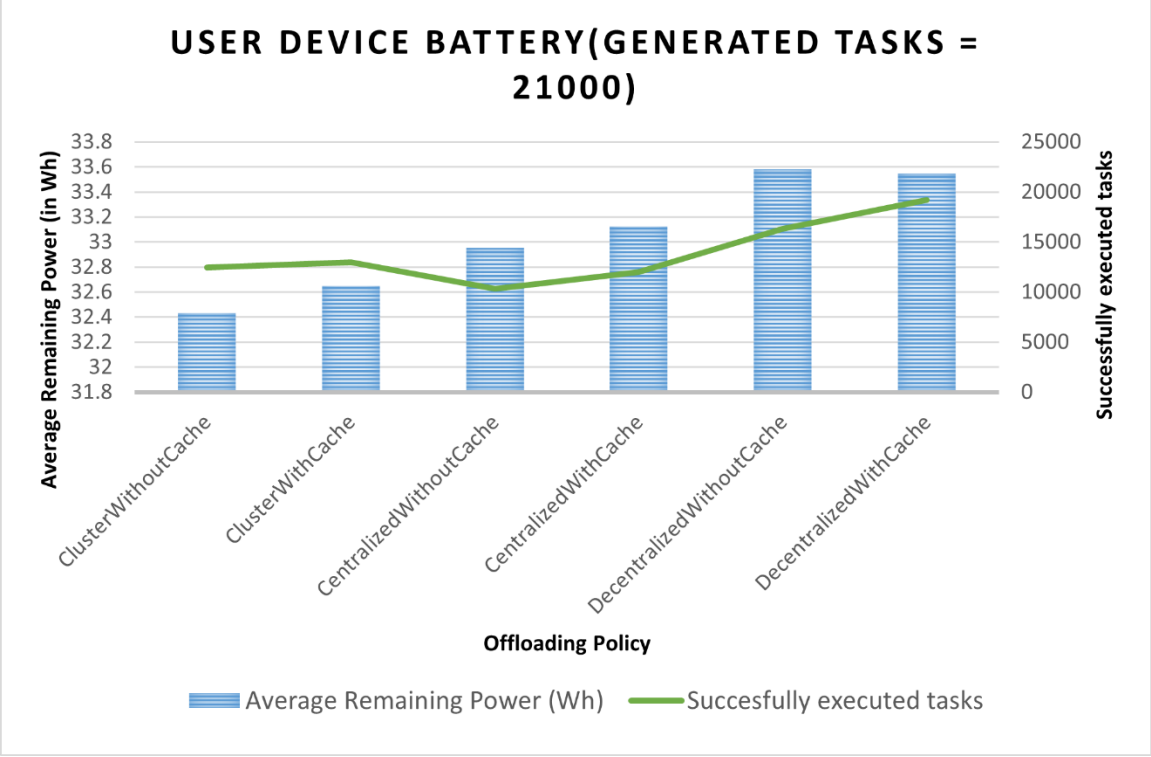

Figure 5.17: User device battery levels when 21000 tasks are generated

Therefore, proposed decentralized offloading with container caching outperforms all the mentioned benchmark algorithms in successfully offloading tasks in a heterogeneous edge computing environment. The best performance in a discrete event simulator on a large scale shows the feasibility of using this method in a realistic environment.

5.3 Chapter Summary

This chapter presents the considered simulations setup along with the specifications of user devices and datacenters. Then the performance of the proposed Online DQRL without caching is validated by comparing the results with Fuzzy Decision Tree, Simulated Annealing, Round-robin, and Trade-off algorithms. Moreover, proposed container-based caching is compared to task-based caching to understand better the significance of storing the dependencies. Finally, the suggested Decentralized Online DQRL with container caching is compared to centralized and cluster-based orchestration frameworks to analyze the impact of deployment location and caching. The feasibility of applying the suggested framework in a realistic environment is validated through extensive simulations.

Chapter 6: Conclusion and Future Work

6.1 Conclusion

This thesis considered the problem of resource allocation and task offloading in an edge cloud environment with the dynamic arrival of tasks from static and mobile users. Stochastic tasks were scheduled from user devices simultaneously, and offloading decisions were taken to replicate a realistic scenario. Task parameters comprised allowed delay, length (Million Instructions Per Seconds), container size, request size, result size, and the number of cores. Online Deep Q Reinforcement Learning (DQRL) was utilized as it can solve complex problems with high-dimensional state space. The objective function comprised energy consumption and latency considering resource availability constraints. Scheduling overhead, task failures due to delay and mobility, network performance, energy consumption, scalability, remaining power of users' devices were analyzed for the implemented model. Simulation results demonstrated that Online DQRL performs at least 60% and 33% better than Round Robin and Trade-Off algorithms in energy consumption and network traffic, respectively. A minimum of 33% improvement against all the other benchmark algorithms was observed in terms of task failure rate by incorporating Online DQRL.

Limited resource availability and the stringent timely delivery requirements in various 5G applications motivated us to utilize caching for improving the performance further. Container-based caching to store the software dependencies for popular applications based

on the incurred storage cost and frequency of application requests was proposed. The recommended approach is validated by comparing the results with Task-based caching, where the task-related input and output data is used to improve task offloading performance. Container-based caching reduced task failures and energy consumption by at least 50% and 60%, respectively, compared to task-based caching in multiple simulation scenarios. In addition to that, the location of orchestrator deployment is also reviewed, and a distributed offloading strategy is proposed. Decentralized offloading mechanism decreased the utilization of available bandwidth by a minimum of 47%, besides reducing the task failures by at least 55% compared to other orchestration frameworks. Also, the recommended decentralized approach utilized only 3% of user device battery power while maintaining at least 70% task success rate compared to centralized and cluster-based orchestration. The exceptional performance in terms of energy consumption, network traffic, remaining battery power, and task failures on a large scale demonstrated the reliability of using Decentralized container-based caching in a realistic environment.

6.2 Future Work

Although the proposed thesis covers various aspects of the real environment to address the task offloading problem, many more scenarios and directions can be researched further and optimized. In addition to that, the massive growth of IoT applications and their service level agreements may result in new challenges which are worthy of research. Therefore, the shortcoming and future directions of the proposed thesis can be summarized as below:

1. **Cache Replacement Strategy:** The proposed framework introduces container-based caching to avoid recurrent downloading of popular application containers. However, recommended caching strategy frees up space to cache a new application based on the queue principle. Instead, more criteria can be developed for improved cache replacement [81] to avoid losing popular data. Therefore, an algorithm can be devised to decide the deletion of already existing containers based on data volatility, cache locality, and data size.

2. **Task handover:** The recommended approach considers user mobility and distance to an edge server to ensure stable data transmission. However, mobility of users and outage of an edge server may interrupt the data transfer resulting in a failed execution of tasks. Tasks generated from users may fail due to delay, mobility, and depletion of the battery. Therefore, handing over tasks can be an excellent alternative to overcome failures and to continue successful task execution. The task handover [82] process is out of the scope of the proposed thesis and is not incorporated with the current simulation setup. In addition to that, the data transmission and data sharing between edge servers was not considered in the proposed approach.

3. **Deep Learning layer assignment:** Many tasks are usually computed at edge server or locally as it incurs enormous delay to transfer the task-related information to a central cloud. This problem can be addressed by compressing the data locally or at the edge server and then sending the compressed information to the cloud for further execution. Distributed task execution makes the process faster and effectively utilizes all the available resources.

Therefore, an ideal algorithm to identify the number of layers [83] [84] that have to be computed at various levels (local, edge server, and central cloud) can improve the network efficiency.

4. **Data privacy and security:** As tasks from many user devices are run simultaneously at edge servers and at remote cloud, it should also be ensured that the transmitted data is not corrupt. However, security cannot be compromised for achieving ultra-low latencies [85]. It is essential to maintain data privacy and security through proper encryption. The encryption mechanism and protocols used to transfer information between various data centers are out of the scope of the proposed thesis and can be investigated to detect misbehavior and anomalies in the future.

5. **Network Architecture:** The considered simulation topology assumes a limited-service area with constant bandwidth. The change in available bandwidth, interferences experienced, and data loss are out of the scope of this thesis. Therefore, the proposed offloading algorithm can be validated in a software-defined networking (SDN) architecture [86] with detailed modeling of various network components in the future.

Bibliography

[1] S. Chaudhary, R. Johari, R. Bhatia, K. Gupta and A. Bhatnagar, "CRAIoT: Concept, Review and Application(s) of IoT," in *2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU)*, India, 2019.

[2] F. Liang, W. Yu, X. Liu, D. Griffith and N. Golmie, "Toward Edge-Based Deep Learning in Industrial Internet of Things," *IEEE Internet of Things Journal,* vol. 7, no. 5, 2020.

[3] L. U. Khan, I. Yaqoob, N. H. Tran, S. M. A. Kazmi, T. N. Dang and C. S. Hong, "Edge-Computing-Enabled Smart Cities: A Comprehensive Survey," *IEEE Internet of Things Journal,* vol. 7, no. 10, 2020.

[4] J. Li, Y. Ma and Y. Wang, "Environment Construction of Virtual Reality Technology Based on Edge Computing in Immersive Communication," in *2021 International Conference on Computer Technology and Media Convergence Design (CTMCD)*, Sanya, 2021.

[5] N. Hassan, K.-L. A. Yau and C. Wu, "Edge Computing in 5G: A Review," *IEEE Access,* vol. 7, 2019.

[6] V. C. Emeakaroha, N. Cafferkey, P. Healy and J. P. Morrison, "A Cloud-Based IoT Data Gathering and Processing Platform," in *2015 3rd International Conference on Future Internet of Things and Cloud, IEEE*, Rome, Italy, 2015.

[7] M. R. Rahimi, N. Venkatasubramanian, S. Mehrotra and A. V. Vasilakos, “On Optimal and Fair Service Allocation in Mobile Cloud Computing,” *IEEE Transactions on Cloud Computing ,* vol. 6, no. 3, 2018.

[8] M. B. Monir, T. Abdelkader and E.-S. M. Ei-Horbaty, “Trust Evaluation of Service level Agreement for Service Providers in Mobile Edge Computing,” in *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*, Cairo, 2019.

[9] T. Qiu, J. Chi, X. Zhou, Z. Ning, M. Atiquzzaman and D. Oliver, “Edge Computing in Industrial Internet of Things: Architecture, Advances and Challenges,” *IEEE Communications Surveys & Tutorials,* vol. 22, no. 4, p. 3, 2020.

[10] N. K. Giang, R. Lea, M. Blackstock and V. C. Leung, “Fog at the Edge: Experiences Building an Edge Computing Platform,” in *2018 IEEE International Conference on Edge Computing (EDGE)*, San Francisco, 2018.

[11] T. K. Rodrigues, K. Suto and N. Kato, “Edge Cloud Server Deployment with Transmission Power Control through Machine Learning for 6G Internet of Things,” *IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING,* p. 2, 2019.

[12] K. Zhang, “Task Offloading and Resource Allocation using Deep Reinforcement Learning,” University of Ottawa, Ottawa, 2020.

[13] S. Naveen and M. R. Kounte, “Key Technologies and challenges in IoT Edge Computing,” in *2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, Palladam, 2020.

[14] T. ALFAKIH, M. M. HASSAN, A. GUMAEI, C. SAVAGLIO and G. FORTINO, "Task Offloading and Resource Allocation for MEC by Deep Reinforcement Learning Based on SARSA," *IEEE ACCESS SPECIAL SECTION ON CLOUD - FOG - EDGE COMPUTING IN CYBER-PHYSICAL-SOCIAL,* vol. 8, 2020.

[15] O.-K. Shahryari, H. Pedram, V. Khajehvand and M. D. TakhtFooladi, "Energy and task completion time trade-off for task offloading in fog-enabled IoT networks," *ELSEVIER Pervasive and Mobile Computing,* 2020.

[16] H. Wu, K. Wolter, P. Jiao, Y. Deng, Y. Zhao and M. Xu, "EEDTO: An Energy-Efficient Dynamic Task Offloading Algorithm for Blockchain-Enabled IoT-Edge-Cloud Orchestrated Computing," *IEEE Internet of Things Journal,* vol. 8, no. 4, p. 4, 2021.

[17] C.-C. Lin, P. Liu and J.-J. Wu, "Energy-Aware Virtual Machine Dynamic Provision and Scheduling for Cloud Computing," in *2011 IEEE 4th International Conference on Cloud Computing*, Washington, 2011.

[18] J. Yao, T. Han and N. Ansari, "On Mobile Edge Caching," *IEEE COMMUNICATIONS SURVEYS & TUTORIALS,* vol. 21, no. 3, p. 2, 2019.

[19] W. Xu, L. Chen and H. Yang, "A Comprehensive Discussion on Deep Reinforcement Learning," in *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)*, Beijing, 2021.

[20] E. C. S. D. w. T. P. C. t. M. L. a. 6. I. o. Things, "Tiago Koketsu Rodrigues; Katsuya Suto; Nei Kato," *IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING,* vol. 9, no. 4, 2019.

[21] P Jonsson; S Carson; G Blennerud; J Kyohun Shim; B Arendse; A Husseini, "Ericsson mobility report," Ericsson, Stockholm, Sweden, 2020.

[22] C. Pallasch, S. Wein, N. Hoffmann, M. Obdenbusch, T. Buchner, J. Waltl and C. Brecher, "Edge Powered Industrial Control: Concept for Combining Cloud and Automation Technologies," in *2018 IEEE International Conference on Edge Computing (EDGE)*, San Francisco, 2018.

[23] S. Kaur and T. Sharma, "Efficient load balancing using improved central load balancing technique," in *2018 2nd International Conference on Inventive Systems and Control (ICISC)*, Coimbatore, 2018.

[24] K. Gai and S. Li, "Towards Cloud Computing: A Literature Review on Cloud Computing and Its Development Trends," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, Nanjing, China, 2012.

[25] B. P. Rimal, D. P. Van and M. Maier, "Mobile-Edge Computing Versus Centralized Cloud Computing Over a Converged FiWi Access Network," *IEEE Transactions on Network and Service Management,* vol. 14, no. 3, 2017.

[26] A. H. A. Bafghi, M. Mirmohseni, F. Ashtiani and M. Nasiri-Kenari, "Joint Optimization of Power Consumption and Transmission Delay in a Cache-Enabled C-RAN," *IEEE Wireless Communications Letters,* vol. 9, no. 8, 2020.

[27] S. Panwar, "Breaking the millisecond barrier: Robots and self-driving cars will need completely reengineered networks," *IEEE Spectrum,* vol. 57, no. 11, p. 2, 2020.

[28] S. S. D. Ali, H. P. Zhao and H. Kim, “Mobile Edge Computing: A Promising Paradigm for Future Communication Systems,” in *TENCON 2018 - 2018 IEEE Region 10 Conference*, Jeju, Korea (South), 2018.

[29] B. Panchali, “Edge Computing- Background and Overview,” in *2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, Tirunelveli, 2019.

[30] Milan Patel; Brian Naughton; Caroline Chan; Nurit Sprecher; Sadayuki Abeta; Adrian Neal; Peter Cosimini; Adam Pollard; Guenter Klas, “Mobile-Edge Computing – Introductory Technical White Paper,” ETSI, 2014.

[31] Y. Koren, R. Bell and C. Volinsky, “Matrix Factorization Techniques for Recommender Systems,” *Computer (IEEE),* vol. 42, no. 8, 2009.

[32] J. Yao, T. Han and N. Ansari, “On Mobile Edge Caching,” *IEEE Communications Surveys & Tutorials,* vol. 21, no. 3, pp. 5-6, 2019.

[33] J. Zhang, X. Zhou, T. Ge, X. Wang and T. Hwang, “Joint Task Scheduling and Containerizing for Efficient Edge Computing,” *IEEE Transactions on Parallel and Distributed Systems,* vol. 32, no. 8, p. 1, 2021.

[34] H. Wan, “Deep Learning:Neural Network, Optimizing Method and Libraries Review,” in *2019 International Conference on Robots & Intelligent System (ICRIS)*, Haikou, 2019.

[35] N. Praveena and K. Vivekanandan, “A Review on Deep Neural Network Design and Their Applications,” in *2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS)*, Coimbatore, 2021.

[36] W. Xu, L. Chen and H. Yang, “A Comprehensive Discussion on Deep Reinforcement Learning,” in *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)*, Beijing, 2021.

[37] W. T. Scherer, S. Adams and P. A. Beling, “On the Practical Art of State Definitions for Markov Decision Process Construction,” *IEEE Access,* vol. 6, p. 4, 2018.

[38] M. NAEEM, S. T. H. R. and A. C. , “A Gentle Introduction to Reinforcement Learning and Its Application in Different Fields,” *IEEE Access,* vol. 8, p. 4, 2020.

[39] T. ALFAKIH, M. M. HASSAN, A. GUMAEI, C. SAVAGLIO and G. FORTINO, “Task Offloading and Resource Allocation for Mobile Edge Computing by Deep Reinforcement Learning based on SARSA,” *IEEE Access (SPECIAL SECTION ON CLOUD - FOG - EDGE COMPUTING IN CYBER-PHYSICAL-SOCIAL SYSTEMS),* vol. 8, 2020.

[40] S. Nath and J. Wu, “Deep reinforcement learning for dynamic computation offloading and resource allocation in cache-assisted mobile edge computing systems,” in *Intelligent and Converged Networks*, 2020.

[41] Microsoft, [Online]. Available: https://azure.microsoft.com.

[42] [Online]. Available: https://aws.amazon.com..

[43] S.Wang, X. Zhang, Y. Zhang, L.Wang, J. Yang and W.Wang, “A survey on mobile edge networks: Convergence of computing, caching and communications,” *IEEE Access,* vol. 5, p. 2, 2017.

[44] P. Mach and Z. Becvar, “Mobile edge computing: A survey on architecture and computation offloading,” *IEEE Communications Survey,* vol. 19, no. 3, p. 2, 2017.

[45] J. WANG, J. PAN, F. ESPOSITO, P. CALYAM, Z. YANG and P. MOHAPATRA, “Edge Cloud Offloading Algorithms: Issues, Methods, and Perspectives,” *ACM Compuing Survey,* vol. 1, no. 2, p. 5, 2019.

[46] S. Sardellitti, G. Scutari and S. Barbarossa, “Joint optimization of radio and computational resources for multicell mobile-edge computing,” *IEEE Transactions in Signal and Information Processing,* vol. 1, no. 2, p. 3, 2015.

[47] C. You, K. Huang, H. Chae and B.-H. Kim, “Energy-Efficient Resource Allocation for Mobile-Edge Computation Offloading,” *IEEE Transactions on Wireless communications,* p. 1, 2016.

[48] K. Habak, M. Ammar, K. A. Harras and E. Zegura, “Femto clouds: Leveraging mobile devices to provide cloud service at the edge,” in *Proceedings of IEEE International Cloud Computing Conference*, 2015.

[49] M. Deng, H. Tian and X. Lyu, “Adaptive sequential offloading game for multi-cell mobile edge computing,” in *IEEE Proceedings of 23rd Internation Telecommunications Conference*, 2016.

[50] Y. Mao, J. Zhang and K. B. Letaief, “Dynamic computation offloading for mobile-edge computing with energy harvesting devices,” *IEEE Selected Areas in Communication,* vol. 34, no. 12, pp. 3590-3605, 2016.

[51] Y. Wang, M. Sheng, X. Wang, L. Wang and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling," *IEEE Transactions in Communications,* vol. 64, no. 10, pp. 4268-4282, 2016.

[52] M. H. U. Rehman, C. Sun, T. Y. Wah, A. Iqbal and P. P. Jayaraman, "Opportunistic computation offloading in mobile edge cloud computing environments," in *Proceedings of 17th IEEE International Conference on Mobile Data Management*, 2016.

[53] A. R. Khan, M. Othman, S. A. Madani and S. U. Khan, "A survey of mobile cloud computing application models," *IEEE Communication Surveys,* vol. 16, no. 1, pp. 393-413, 2014.

[54] Z. Sanaei, S. Abolfazli, A. Gani and R. Buyya, "Heterogeneity in mobile cloud computing: Taxonomy and open challenges," *IEEE Communication Surveys,* vol. 16, no. 1, pp. 369-392, 2014.

[55] R. Roman, J. Lopez and M. Mambo, "Mobile edge computing Fog et al .: A survey and analysis of security threats and challenges," *Future Generation Computer Systems.*

[56] N. C. Luong, P. Wang, D. Niyato, Y. Wen and Z. Han, "Resource management in cloud networking using economic analysis and pricing models: A survey," *IEEE Communication Surveys.*

[57] M. D. Hossain, T. Sultana, M. A. Hossain and E.-N. Huh, "Edge Orchestration Based Computation Peer Offloading in MEC Enable Networks: A Fuzzy Logic

Approach," in *2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, 2021.

[58] N. CHA, C. WU, T. YOSHINAGA, Y. JI and K.-L. A. YAU, "Virtual Edge: Exploring Computation Offloading in Collaborative Vehicular Edge Computing," *IEEE ACCESS,* vol. 9, 2021.

[59] Y. Li and W. Wang, "Can mobile cloudlets support mobile applications?," in *IEEE Conference Computing Communications (INFOCOM)*, 2014.

[60] X. Gu and G. Zhang, "Energy-efficient computation offloading for vehicular edge computing networks," *ELSEVIER Computer Communications,* 2020.

[61] Z. WANG, S. ZHENG, Q. GE and K. LI, "Online Offloading Scheduling and Resource Allocation Algorithms for Vehicular Edge Computing System," *IEEE Access (SPECIAL SECTION ON COMMUNICATION AND FOG/EDGE COMPUTING TOWARDS INTELLIGENT CONNECTED VEHICLES (ICVS),* vol. 8, 2020.

[62] Z. Chen and Z. Zhou, "Dynamic Task Caching and Computation Offloading for Mobile Edge Computing," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020.

[63] Y. Dai, K. Zhang, S. Maharjan and Y. Zhang, "Edge Intelligence for Energy-Efficient Computation Offloading and Resource Allocation in 5G Beyond," *IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY,* vol. 69, no. 10, 2020.

[64] X. Li, L. Zhao, K. Yu, M. Aloqaily and Y. Jararweh, “A cooperative resource allocation model for IoT applications in mobile edge computing,” *Computer Communications,* 2021.

[65] Z. LI, C. DU and S. CHEN, “HIQCO: A Hierarchical Optimization Method for Computation Offloading and Resource Optimization in Multi-Cell Mobile-Edge Computing Systems,” *IEEE Access,* vol. 8, 2020.

[66] 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,* vol. 8, no. 5, 2021.

[67] Q. Wang, Z. Li, K. Nai, Y. Chen and M. Wen, “Dynamic resource allocation for jointing vehicle-edge deep neural network inference,” *ELSEVIER Journal of Systems Architecture,* 2021.

[68] S. Tuli, S. Ilager, K. Ramamohanarao and R. Buyya, “Dynamic Scheduling for Stochastic Edge-Cloud Computing Environments using A3C learning and Residual Recurrent Neural Networks,” *IEEE TRANSACTION ON MOBILE COMPUTING,* p. 3, 2020.

[69] H. Wu, Z. Zhang, C. Guan, K. Wolter and M. Xu, “Collaborate Edge and Cloud Computing With Distributed Deep Learning for Smart City Internet of Things,” *IEEE INTERNET OF THINGS JOURNAL,* vol. 7, no. 9, 2020.

[70] J. Chen, H. Xing, X. Lin and S. Bi, “Joint Cache Placement and Bandwidth Allocation for FDMA-based Mobile Edge Computing Systems,” in *IEEE International Conference on Communications (ICC)*, Dublin, Ireland, 2020.

[71] Y. Liu, Q. He, D. Zheng, X. Xia, F. Chen and B. Zhang, “Data Caching Optimization in the Edge Computing Environment,” *IEEE TRANSACTIONS ON SERVICES COMPUTING,* 2020.

[72] C. Li, M. Song, S. Du, X. Wang, M. Zhang and Y. Luo, “Adaptive priority-based cache replacement and prediction-based cache prefetching in edge computing environment,” *Journal of Network and Computer Applications,* no. 165, p. 4, 2020.

[73] J. WANG, J. PAN, F. ESPOSITO, P. CALYAM, Z. YANG and P. MOHAPATRA, “Edge Cloud Offloading Algorithms: Issues, Methods, and Perspectives,” *ACM Computing Surveys,* vol. 52, no. 1, p. 3, 2019.

[74] L. Chunlin, T. Jianhang, H. Tang and Y. Luo, “Collaborative cache allocation and task scheduling for data-intensive applications in edge computing environment,” *Future Generation Computer Systems,* 2019.

[75] J. Singh, Y. Bello, A. R. Hussein, A. Erbad and A. Mohamed, “Hierarchical Security Paradigm for IoT Multiaccess Edge Computing,” *IEEE Internet of Things Journal ,* vol. 8, no. 7, 2021.

[76] [Online]. Available: https://www.vmware.com/topics/glossary/content/virtual-machine.html.

[77] H. Mao, M. Alizadeh, I. Menache and S. Kandula, “Resource Management with Deep Reinforcement Learning,” *HotNets '16: Proceedings of the 15th ACM Workshop on Hot Topics in Networks,* pp. 50-56, 09 November 2016.

[78] Z. Yang, Y. Xie and Z. Wang, “A Theoretical Analysis of Deep Q-Learning,” in *ICLR 2020 Conference Blind Submission*, 2019.

[79] M. C. S. Filho, R. L. Oliveira, C. C. Monteiro, P. R. M. Inácio and M. M. Freire, “CloudSim Plus: A cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness,” in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, Lisbon, 2017.

[80] C. Mechalikh, H. Taktak and F. Moussa, “PureEdgeSim: A Simulation Framework for Performance Evaluation of Cloud, Edge and Mist Computing Environments,” in *The 2019 International Conference on High Performance Computing & Simulation*, Dublin, 2019.

[81] C. Sonmez, A. Ozgovde and C. Ersoy, “Fuzzy Workload Orchestration for Edge Computing,” *IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT,* vol. 16, no. 2, p. 8, 2019.

[82] C. Li, M. Song, S. Du, X. Wang, M. Zhang and Y. Luo, “Adaptive priority-based cache replacement and prediction-based cache prefetching in edge computing environment,” *Journal of Network and Computer Applications,* vol. 165, p. 4, 2020.

[83] T. M. Ho and K.-K. Nguyen, “Joint Server Selection, Cooperative Offloading and Handover in Multi-access Edge Computing Wireless Network: A Deep

Reinforcement Learning Approach," *IEEE Transactions on Mobile Computing,* no. Early Access, 2020.

[84] K. Lee, B. N. Silva and K. Han, "Algorithmic implementation of deep learning layer assignment in edge computing based smart city environment," *Computers and Electrical Engineering,* vol. 89, 2020.

[85] K.-J. Hsu, K. Bhardwaj and A. Gavrilovska, "Couper: DNN Model Slicing for Visual Analytics Containers at the Edge," *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing,* vol. 19, pp. 179-194, 2019.

[86] M. Houmer, M. L. Hasnaoui and A. Elfergougui, "Security Analysis of Vehicular Ad-hoc Networks based on Attack Tree," in *International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT)* , 2018.

[87] W. Rafique, L. Qi, I. Yaqoob, M. Imran, R. U. Rasool and W. Dou, "Complementing IoT Services Through Software Defined Networking and Edge Computing: A Comprehensive Survey," *IEEE Communications Surveys & Tutorials,* vol. 22, no. 3, 2020.

[88] A. Sharma, "Towards Datascience," 01 10 2018. [Online]. Available: https://towardsdatascience.com/restricted-boltzmann-machines-simplified-eab1e5878976. [Accessed 19 01 2022].