Optimizing Reinforcement Learning: Fog and Edge Resource Management Through Bootstrapping and Reward Shaping

Hani Sami

A Thesis

in

The Concordia Institute

for

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Information and Systems Engineering) at

Concordia University

Montréal, Québec, Canada

November 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Hani Sami

Entitled: **Optimizing Reinforcement Learning: Fog and Edge Resource Management Through Bootstrapping and Reward Shaping**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Rodolfo Lima Coutinho

_____ External Examiner
Dr. Ala Al-Fuqaha

_____ Internal Program Examiner
Dr. Juergen Rilling

_____ Internal Examiner
Dr. Rachida Dssouli

_____ Internal Examiner
Dr. Roch Glitho

_____ Supervisor
Dr. Jamal Bentahar

_____ Co-supervisors
Dr. Hadi Otrok & Dr. Azzam Mourad

Approved by _____
Jun Yan, Graduate Program Direction

November 13, 2023

Date of Defense

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Optimizing Reinforcement Learning: Fog and Edge Resource Management Through Bootstrapping and Reward Shaping

Hani Sami, Ph.D.

Concordia University, 2023

The rapid and extensive use of technology is unprecedented. From small devices like sensors and mobile phones to large systems like servers and data centers, a wide range of computing setups exists to meet human needs. However, the increased demand has raised concerns about whether these setups can handle the load. Fog and edge computing are concepts that bring servers closer to users to improve response time and service quality. But the availability of these fog devices is limited, highlighting the need for systems to manage computing resources. These systems' main task is to efficiently distribute services across available resources and adapt to changing needs. Existing resource management solutions still possess challenges and limitations with regards to the quality of decisions due to their increasing complexity.

In this thesis, our main motivation is leveraging AI in addressing resource management, which stems from its ability to intelligently handle complex and dynamic scenarios, such as optimizing service placement, predicting demands, and adapting to changing environments. AI's capacity to learn from data and make informed decisions offers a promising approach to efficiently manage computing resources in a rapidly evolving technological landscape. This research is motivated by four main goals: (1) creating a strong computing

architecture that can meet diverse user needs across various applications managed by a resource management system; (2) using AI to develop resource management solutions that handle decisions like placement and scaling, as well as predict user demands and resource availability; (3) ensuring the AI solution is reliable despite potential errors by improving its performance or having a backup plan; (4) making the AI solution adaptable to sudden environmental changes to keep decisions effective.

The thesis aims to address these gaps by: (1) designing an effective networking and computing architecture in the context of on-demand fog and edge formation, while supporting an Intelligent Computing Resource Management solution (ICRM) for multi-types of applications through offline learning and bootstrapping; (2) using DRL to build the ICRM, driven by a Markov Decision Process (MDP) environment design that produces actions related to host selection and service placement while accounting for the change in user demands; (3) enhancing the proposed MDP by adding the support for predicting the change in both user demands and available computing resources, where the agent becomes capable for issues horizontal and vertical resource scaling decisions in multi-applications setting; (4) introducing the first solution to speed the learning speed of DRL agent by devising a Graph Convolutional Network solution as a potential-based reward shaping solution; (5) developing another reward shaping solution based on Convolutional Neural Network (CNN) carefully designed and inspired by the value iteration network (VIN), to speed learning. Besides these contributions, we present a set of experimental studies and simulations using real-world test cases for each of the contributions compared to state-of-the-art solutions.

In conclusion, this thesis identifies research gaps that warrant further exploration in the future.

Acknowledgments

I am grateful to God for blessing me with the health, capability, and patience to successfully complete this thesis.

Special appreciation goes to my Ph.D. supervisor, Dr. Jamal Bentahar, and my co-supervisors, Dr. Azzam Mourad, and Dr. Hadi Otrok for their continuous support and help throughout this fascinating journey.. Their professionalism, inspiration, and care have been indispensable, and I am acutely aware that my achievement would not have been possible without their guidance. Dr. Mourad's belief in me during my undergraduate studies laid the foundation for my academic journey, while Dr. Otrok's consistent advice and mentorship significantly shaped my career path. Dr. Jamal's inspirational leadership, spending considerable time supporting and guiding me in various projects, has left an enduring mark on my academic and personal growth.

I extend my gratitude to the esteemed members of my Ph.D. committee—Dr. Rachida Dssouli, Dr. Juergen Rilling, Dr. Roch Glitho, and Dr. Ala al-Fuqaha—for dedicating their valuable time and expertise to evaluate this thesis. Their insightful comments and profound knowledge have played a significant role in refining the thesis to its present form, providing a platform for the exploration of innovative and novel ideas.

Acknowledgement is also due to my colleagues at the research lab at Concordia University and the research team I am privileged to be a part of. Special thanks to Ahmad Hammoud, Mohamad Arafeh, Mohamad Wazzeh, Sarhad Arisdakessian, Mario Chahoud, and Osama Wehbi for fostering a collaborative and supportive work environment that feels

like a second family. I would like to also thank my friend Khaled Sarieddine with whom I shared precious and enjoyable moments during this journey.

This research was made possible through the generous financial support of the Fonds de recherche du Québec - Nature et technologies (FRQNT) and Concordia University. This support significantly alleviated my financial burdens, enabling me to focus on enhancing my research productivity.

Last but certainly not least, my deepest thanks go to my beloved wife, Wajiha, for her unwavering and boundless support throughout my Ph.D. journey. Without her endless encouragement, this accomplishment would not have been possible. I also express gratitude to my parents for surrounding me with unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an introduction to the context of our research work, outlines the problems addressed in this thesis, presents the corresponding research questions, and defines the objectives and contributions of our research.

1.1 Research Context and Motivation

Fog computing is utilized to work out the limitations of distant clouds affecting IoT devices in terms of networking, computation, and data storage. Fog can be any computing device located close to the user. The creation of these fogs can be static [3] or dynamic [1], depending on the environment it is serving. Effective computing resource management plays a pivotal role in today's technology-driven world. It involves the efficient allocation and utilization of the fog computing resources, such as processing power, memory, storage, and network bandwidth, to meet the diverse demands of various applications and tasks. The process of Intelligent Computing Resource Management (ICRM) encompasses resource optimization, scaling, load balancing, scheduling, and intelligent decision-making to ensure the most efficient and effective use of available resources. With the exponential growth of technology and the increasing complexity of computing systems, the significance

of ICRM cannot be overstated. In particular, ICRM is essential for various applications, including Autonomous Driving, energy management, smart cities, healthcare, financial services, manufacturing, industrial automation, video surveillance, and Natural Language Processing (NLP). These applications rely on real-time data processing, resource optimization, and intelligent decision-making for improved performance and efficiency.

To achieve a generalized environment setting for various applications, we explore the concept of on-demand computing [1, 4]. This mechanism utilizes Docker containers for hosting edge services orchestrated by Kubernetes clusters [5] on-demand. Docker containers offer a lightweight solution to offload and run containers on constraint computing devices. In this context, nodes are volunteers that possess variable capacities of CPU, Memory, and Disk resources. Despite the on-demand flexibility that offers lightweight service placement, there must be an effective ICRM for managing the selection of hosts and placement of services intelligently. Hence, the motivation of this thesis is to build an ICRM that considers the change in demands by users for various services in different locations and provide the required support proactively with the minimal impact on the Quality of Service (QoS). Such an action entails a careful consideration to the required service resources and the offerings on hosts. Furthermore, It is important to consider the time it takes to initialize or update an environment for serving these users. Besides, the ICRM should offer a flexibility of scaling computing resources dynamically and proactively, while attaining for a variety of objective functions that can required by the application.

1.2 Computing Resource Management: An Example

Users request a set of different services from the cloud using their smart devices. As shown in Fig. 1.1, these requests arrive from different locations with different volumes. Assuming that five services are initially running on the cloud $\{S1, S2, \ldots, S5\}$, each represented by a different color in figure 1.1, the cloud can become overloaded with tons of

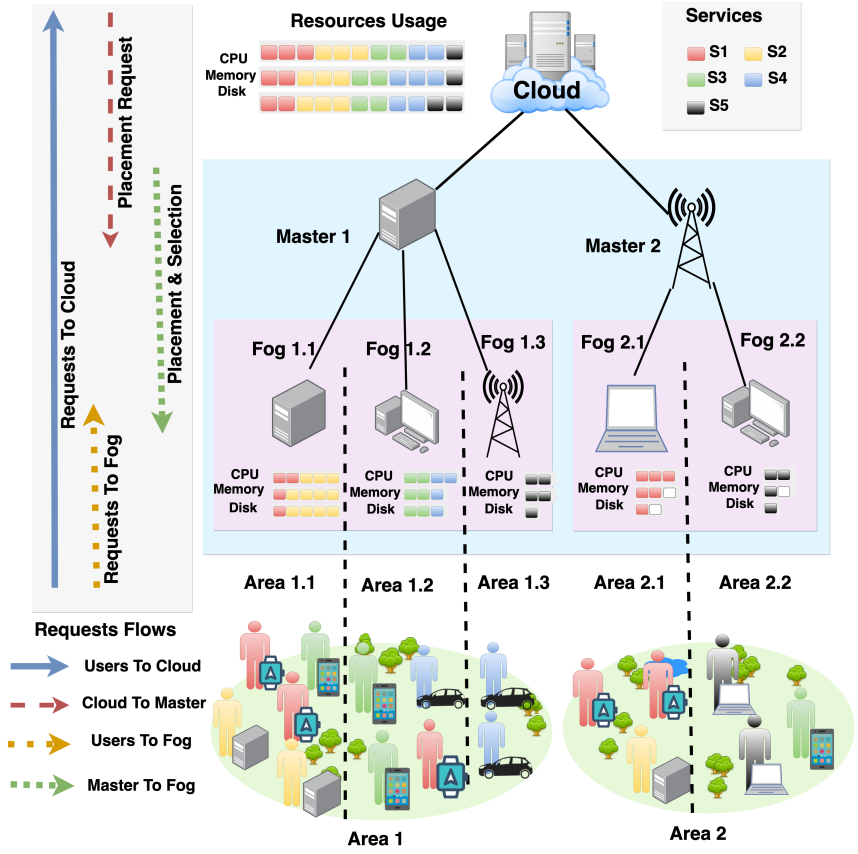


Figure 1.1: An Illustration of the Service Placement Problem on Fog Clusters

requests. Subsequently, users start complaining because of the embarrassing QoS caused by transmission delay or high load on the cloud servers. Thereafter, fogs are used to overcome these issues by running close to users with acceptable performance. In addition, edge also refers to computing machine that run closer to the users than fogs. In the remainder of this thesis, we are not making differences between fog and edge devices, so they are used interchangeably. As illustrated in the figure, the cloud makes scheduling decisions on a set of services that need to be placed near users. Following the approach in [6], the placement request is received by the master of the fog cluster. One of the master's tasks is to distribute these services on its fogs in a way that enhances the QoS level experienced by the users and preserves the fog requirements. For instance, these services should be as close as possible to users requesting them. Every fog has a coverage area users can reach. Therefore, fogs should serve users present within their range. In Fig. 1.1, Fog 1.1, serving Area 1.1, can only host $S1$ and $S2$ because of the limitation of its resources. In this subarea, a user is requesting service $S3$ (green); however, it's not hosted by Fog 1.1. Therefore, this user is still served by the cloud. Because of the fog servers resource limitation inside their Kubernetes cluster, solving the problem of fog selection and service distribution is of immense importance. In addition, the demands of services coming from users to fogs change over time. For instance, in Area 1.2, Fog 1.2 is hosting $S3$ and $S4$. If the demands for $S1$ gets higher than $S4$, the fog has to switch these services to start serving $S1$ to achieve a better QoS. The demands are only one objective that the master has to satisfy when adjusting the placements. Other objectives have to be considered. For example, minimizing the distance between the serving fog and the user, maximizing the number of services pushed having a higher priority, and minimizing the number of hosts for lighter orchestration. These objectives are essential to consider when making selection and placement decisions.

1.3 Problem Statement and Research Questions

Dynamic formation of on-demand fog clusters introduces the service placement problem. This problem is divided into fog selection and service assignment or placement. Fog selection entails choosing the best fog from a set of available ones, whereas service placement is the action of assigning services to selected fogs. The service placement and fog selection problem is NP-hard and requires at least a heuristic solution to solve it [1]. For instance, genetic algorithms use randomness to build populations, evaluate the solutions' fitness value, and evolve the pareto front. Thus, there is no single solution to the problem, especially when the input grows due to the hardness of reaching the optimal decision [1]. In other words, heuristic solutions are not always guaranteed to make acceptable decisions, affecting the quality of decisions made in critical situations. Furthermore, our problem requires studying users' demand of services so that services with high demand are prioritized for placement. Such a demand needs to be predicted in order for the model to react proactively by placing services before the demand occurs, and therefore eliminating the overhead of initializing fogs, migrating, and starting services. Proactive decision making is not feasible when using heuristics because of the lack of prediction model that can adapt to the stochastic change in demand for different services and from different locations.

Besides the limitation in the quality of decisions considering that an environment state is given, a change in the requirements or demands in this state hinders the adequacy of the existing placement. Traditional static resource allocation approaches [3] often lead to under-utilization or over-utilization of resources, resulting in performance bottlenecks, increased costs, and inefficient operations. Such static solutions, based on rule-based, predefined allocations, or heuristic-based methods, cannot adapt to changing user demands for different services, rendering them unsuitable for modern applications with complex requirement. This leads us to our first research question (RQ1):

- **RQ1: How to produce effective selection and placement decisions that account for a variety of objectives or application requirements as well as the change of user demands?**

Resource scaling decisions should combine horizontal and vertical scaling for more optimized resource usage [7]. In short, horizontal and vertical scaling imply adding/removing service instances to/from running hosts, and adjusting the amount of resources used by a running instance respectively. Unfortunately, existing auto-scaling solutions do not have robust models for predicting the change in demands for services [8], in addition to the change in computing resources. Moreover, hosts running the application instance offer a certain volume of resources that are subject to change depending on other running applications. Performing scaling on hosts with varying resource availability can cause resource overflow and application downtime. Furthermore, horizontal scaling creates new instances of services that should be placed on the correct host of a cluster, following a specific set of fog computing objectives [9]. Moreover, existing solutions are targeting a single application or service for scaling [10][9]. However, it is important to study multi-application scaling in the same cluster for achieving combined management of resources [11]. Finally, a large enterprise application may run a large cluster and many applications that require scaling, demanding a scalable solution. Henceforth, our second research question (RQ2) is:

- **RQ2: How to properly scale computing resources horizontally and vertically to better manage the continuously changing computing spaces on different hosts for multi-type and large scale applications?**

When it comes to production deployment and the dynamic management of computing resources, the ability to adjust these resources on the fly is a critical aspect. It requires intelligent decision-making that can effectively handle the challenges of large-scale, complex, and diverse applications. However, predicting user demands accurately and providing

optimal solutions can be a challenging task subject to potential errors.

Firstly, one common source of error is the misjudgment of demand for specific services. In the ever-changing landscape of technology and user preferences, anticipating the exact needs of users can be a daunting task. Such miscalculations can lead to either overprovisioning, wasting valuable resources, or underprovisioning, causing service disruptions and user dissatisfaction.

Secondly, the emergence of new conditions in the environment or applications can render existing solutions obsolete or unfeasible. For instance, an unexpected objective may arise that the placement solution wasn't designed to handle. This requires adaptability and agility in the decision-making process to accommodate novel requirements.

Moreover, the quality of the decision made by the ICRM system heavily relies on the amount of relevant information available about the environment and applications. A lack of sufficient data can limit the system's ability to make well-informed decisions, potentially leading to suboptimal resource allocation.

Therefore, to avoid errors in the decision making in such dynamic and complex environment, the ICRM solution should avoid errors by: (1) adapting to environment demands, (2) adapting to environment changes such as changes in application requirements, and (3) gathering the maximum amount of useful information to make decisions effectively.

- **RQ3: How to build a robust ICRM that avoids mistakes when placing services through offering high flexibility and adaptability to environment changes and diverse applications requirements?**

1.4 Research Objectives and Contributions

The ultimate goal of this thesis is to develop an effective, efficient, and flexible ICRM that is capable of producing host selection and service placement decisions. To achieve this

goal, we have outlined specific objectives as follows::

- **Objective 1**: Design a flexible architecture that facilitates on-demand service management through the ICRM, catering to diverse types of applications.
- **Objective 2**: Harness the power of Artificial Intelligence (AI) to predict user demands for various services, in addition to available volunteering resources, and generate multi-objective placement and resource scaling decisions accordingly.
- **Objective 3**: Implement an effective offline learning solution to enable the AI system to learn before live deployment, thereby reducing the likelihood of placement and scaling errors.
- **Objective 4**: Enhance the AI-based solution to adapt quickly and proactively to environmental changes and ensure a consistent and high level of Quality of Service (QoS).

Aware of the above limitations, described in the problem statement section, and cooping with 1) the Deep Reinforcement Learning (DRL) advancements in the resource management field [12], and 2) the potential of having an AI fostered environment supporting the fog and edge computing [13], exploring the use of DRL as a resource management solution is promising. A breakthrough in Reinforcement Learning (RL) has been witnessed after introducing the DRL algorithms. The DRL can achieve high degree non-linear function approximation capable of solving multi-objective optimization problems with intelligence while being able to adapt to environment changes. Subsequently, DRL is becoming the core solution for several resource management and networking-related problems, compelling intelligent decisions which usually require human intervention. Thus, there is a potential in using DQN in the context of fogs selection, service placement, and demand analysis. As part of our main contribution in this thesis, we argue that a DRL algorithm is the best fit to build an ICRM.

To achieve the stated objectives, this thesis makes the following significant contributions:

(1) **Contribution 1**: We conducted a comprehensive literature review about existing intelligence computing resource management frameworks. We also considered the efforts that tried to improve the speed of learning in Reinforcement Learning. Furthermore, we identified the main research gaps **(This contribution is discussed in Chapter 2)**.

(2) **Contribution 2**: We propose a promising on-demand based fog and edge formation architecture that supports hosting an ICRM in a multi-types of applications setting. Our architecture offers support for the RL solution through offline learning and bootstrapping. **This contribution is discussed in Chapter 3 and 4**.

(3) **Contribution 3**: We designed a Markov Decision Process (MDP) environment for the resource management that considers a multi-objective optimization decision and offers the capability to predict demands. This formulation is used to build a DRL algorithms. This work also offered means for avoiding decision errors made by the DRL during the first stages of learning. **This contribution is discussed in Chapter 3**.

(4) **Contribution 4**: We augmented the MDP of the previous contribution by adding additional support to proactive horizontal and vertical resource scaling that considers both the change in demand and avilable resources on volunteering hosts. Applied in the context of Mobile Edge Computing (MEC), the developed DRL proved high efficiency and intelligence. **This contribution is discussed in Chapter 4**.

(5) **Contribution 5**: We developed a reward shaping solution using Graph Convolutional Network (GCN) combined with Recurrent Neural Network (RNN) to form a GCRN

with the purpose of speeding learning in the context of DRL. **This contribution is discussed in Chapter 5**.

(6) **Contribution 6**: We devised another reward shaping solution using Convolutional Neural Network (CNN) which processes images from the environment and helps the DRL agent speed learning by converging and adapting faster to changes. **This contribution is discussed in Chapter 6**

1.5 Thesis Organization

Chapter 2 provides a detailed background on IoT, cloud, edge, and fog computing, along with computing resource management. It covers various technologies like micro-services, containers, virtual machines, Kubernetes, and network requirements. The chapter explores heuristic and AI solutions, including DRL, GCN, and CNN, and discusses reward shaping in DRL. It also includes a literature review on existing computing resource management solutions and related works using RL and AI in general.

In **Chapter 3**, we showcase the efforts to build a DRL solution, named Intelligent Fog and Service Placement (IFSP), to perform instantaneous placement decisions proactively combined with an offline learning mechanism. Following the MDP formulation and DRL algorithm development, we present a series of experiments to evaluate the performance of IFSP on real-life dataset of resource management. We also show the ability of IFSP to adapt to changes in the environment and improve the Quality of Service (QoS) compared to state-of-the-art-heuristic and DRL solutions.

In **Chapter 4**, we present IScaler, a DRL-based resource scaling and service placement solution combined with a suitable architecture for integration in clustering environments. IScaler is an extension of the work presented in Chapter 3, where horizontal scaling of a

single application using the SARSA RL algorithm is proposed. In the proposed architecture supporting IScaler, we devise a more advanced solution to the problem of learning errors by DRL by presenting an integration to a heuristic-based algorithm. Furthermore, we present in this chapter a series of experiments using real datasets to illustrate the ability of IScaler to perform optimal auto-scaling decisions in multi-application container-based clustering environments, while considering the change in demands and available resources.

Aware of the limitation in the convergence speed of DRL algorithms, we introduce in **Chapter 5** a novel reward shaping solution that improve the performance for most of the DRL solutions. In this chapter we present the novel Graph Convolutional Recurrent Network (GCRN) architecture that combines GCN and RNN. We also showcase the proposed training mechanism utilizing a message passing technique, augment Krylov, and the look ahead advice. The evaluations conducted on the Atari and MuJoCo games and compared to various baselines.

In **Chapter 6**, we further address the problem of slow learning and convergence speed of DRL algorithms by proposing the Value Iteration Network for Reward Shaping (VIN-RS). This Chapter elaborate on the architecture of VIN-RS based on CNN which helps DRL solutions improve planning over time. Furthermore, experiments are performed on tabular games, Atari 2600 and MuJoCo, for discrete and continuous action space to demonstrate the improvement achieved over state-of-the-art solutions.

Finally, we summarize the thesis contributions in **Chapter 7** and highlight on the existing research gap that require further consideration by the research community. Furthermore, we introduce as part of our future directions an ICRM solution that combines the

I-Scaler solution based on DRL with both the GCRN and CNN-based reward shaping techniques.

Chapter 2

Background and Literature Review

This chapter presents a detailed background required to learn about Internet of Things (IoT), cloud, edge, and fog computing, and the computing resource management problem. In particular, we cover the technological enablers related to building an environment fostering the resource management functionalities. Such technologies include, micro-services, containers, virtual machines, Kubernetes and orchestration tools, as well as network requirements. Furthermore, we discuss the use of heuristic solutions, as well as AI solutions that form the main building blocks behind our ICRM solution. Such solutions include: Heuristic algorithms (e.g. Evolutionary Genetic and Memetic algorithms), Deep Reinforcement Learning (DRL), Graph Convolutional Network (GCN), Recurrent Neural Network, and Convolutional Neural Network. As part of our objective and contribution to speed learning in DRL for improving ICRM solutions adaptability, we present the background required for using reward shaping solutions. These solution include potential-based and non-potential-based reward shaping techniques,.

This background is followed by a comprehensive literature review about existing computing resource management solutions, in addition to any intelligent form (ICRM) that uses RL or other forms of AI. Furthermore, we describe the related work that proposes reward shaping solutions to speed learning in the context of RL.

Following each part of our studied literature, we highlight the research gap and compare with our objectives and contributions in this thesis.

2.1 Background

2.1.1 On-Demand Fog and Resource Management

Fog computing was first proposed by Cisco back in 2013 [14] with an objective to reduce (1) the load on the network, (2) the cloud processing load, and (3) the transmission power and energy consumed by IoT devices. In addition, fog devices makes it easier to orchestrate IoT devices through a master-slave architecture [4]. It is also possible to have multi-level hierarchies of fog devices, where the closest are represented as edge nodes. Recently, the concept of on-demand fog computing has been proposed to consider deploying fog nodes on volunteering devices regardless of the place and time. The containerization technology is used to host services on the volunteering fog devices based on the demands of IoT nodes. The massive amount of containers produced to serve as fog services should be managed and controlled by an authority. Therefore, the Kubernetes tool is used to create a master-slave cluster of volunteering devices to host and remove services from worker (slave) nodes, as well as scale the resources. On-demand fog computing is proposed as part of our previous research efforts [1], and it is extended to serve the vehicular fog computing paradigm [15]. In this work, we utilize the on-demand fog computing technology to host containers in Kubernetes clusters for serving critical and time sensitive applications. The smart resource management solution composing the core of our proposal is deployed in the on-demand fog computing environment, powered by an advanced RL solution. The RL solution requires a careful design of an MDP environment to consider the user demands and manage the resources intelligently. In the following section, we describe the foundation of an MDP for decision making problems.

2.1.2 Markov Decision Process

An MDP is a mathematically designed framework for modeling decisions where part of it is random and the other is made by the decision maker. An MDP is a stochastic formulation of an environment that is composed of the following tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$. In this tuple, $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space. The taking action $a \in \mathcal{A}$ while on state $s \in \mathcal{S}$ results in a next state $s^{'}$. The probability of ending up in state $s^{'}$ following the old state and action is retrieved from the probability transition matrix $\mathcal{P}$. The action taken that results in the next state $s^{'}$ is called a transition. For each transition, a reward is given to check how well that action is relative to the problem. A reward function that takes a transition is denoted as $\mathcal{R}$. An MDP is a set of consecutive actions that are taken in the environment while transitioning in the environment to reach the final goal. Given that the aim is to reach the final or goal state through the minimum number of transitions possible, a total discounted reward should be computed. This discounted reward is maximized, resulting in a trajectory with the highest set of rewards to reach the final goal. In this regard, γ is the discounted factor used as part of the MDP environment design. For solving each problem, all these components should work together to model the problem environment. A solution resulting in the maximum total discounted reward for the problem is the optimal policy, which is denoted as π^*. A solution to compute the optimal ($\pi*$) or near optimal policy (π) for finite state and action spaces of an MDP is usually done using dynamic programming. For such solution, there is an assumption that the reward function and probability transition matrix are given, which is usually not the case. Solutions for MDP are categorized into value and policy iterations or updates.

2.1.3 Reinforcement Learning

Using an accurate MDP formulation of the environment with regards to the problem, an agent can be developed using RL to find an optimal solution or policy to the problem. Compared to other forms of Machine Learning (i.e. supervised and unsupervised), RL solutions do not learn from datasets. An RL agent learns from experiences or transitions taken in the environment based on the actions, which can be induced following a trade-off between exploration and exploitation. Such trade-off can be addressed by using the ϵ-greedy method which increases the exploration rate at the first stages of learning. The exploration rate then gets diminished as the agent continues learning in the environment. Exploration is selecting random actions in the environment no matter the cost, while exploitation is mainly evaluating the current value function and extracting the best possible action maximizing the notion of cumulative rewards. Compared to dynamic programming, RL methods are used to solve large MDPs where an explicit mathematical model is unknown. There exist multiple RL solutions including model-based and model-free. A model-based algorithm builds an explicit policy π, while model-free algorithms do not use any policy. A policy in model-free settings is inferred from a value function. In model-free methods, the policy is changed before the values are settled. Depending on the size of the state and action spaces, a tabular or deep learning version of RL can be used. There exist multiple types of Deep Reinforcement Learning (DRL) algorithms such as Deep Q-Network, Proximal Policy Optimization, and many versions of the Actor Critic (A2C) solution.

A basic RL agent interacts with the environment while at state s_t and considering action a_t. The agent tries to find the policy $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$ by minimizing a cost function $\mathcal{C}$. Action $a(t+1)$ in an MDP is incurred from the previous action and state, which tells the agent about the decisions to take. In value-iteration RL methods, the performance of $a(t+1)$ is represented using the state value function $V_\pi(s)$, which is calculated by observing the rewards during an infinite time horizon. In order to perform model-free control to

improve a policy, the agent should perceive $\mathcal{P}$ in order to configure its actions greedily towards this value function. However, in most cases, the dynamics of the environment are not given. An alternative is to use the state-action value function $Q(s, a)$, where considering greedy actions is possible through maximizing or minimizing $Q(s, a)$ of a given state. The ultimate goal of using an RL solution is to find the optimal policy π^*, which can minimize $\mathcal{C}_t$. π^* can be expressed as follows:

$$\pi^*(s) = \arg\min_a Q^*(s, a) \quad \forall s \in \mathcal{S} \tag{1}$$

Update the state-action value function is possible using the Bellman update, which is expressed as follows:

$$Q(S, A) \leftarrow Q(S, A) + \alpha(C(S, A', S') + \gamma Q(S', A') - Q(S, A)) \tag{2}$$

where α is the learning rate.

2.1.4 Hidden Markov Model

A Hidden Markov Model (HMM) is a Markov Process but with a set of hidden states. As part of the HMM definition, a hidden process is introduced, which is directly influenced by the original Markov Process. Through a solution to the observations, a solution to the Markov Process can be induced. Observations are denoted as $\mathcal{O}$. The goal is to compute the probability of the observation belonging to an optimal trajectory.

A common solution is to use the forward-backward algorithm. The probability inference view of RL can be used to build a reward shaping function [16]. To apply probability inference on an MDP structure, the binary optimality variable O is introduced, where $O_t = 1$ means the state $S_t \in \mathcal{S}$ is optimal, and $O_t = 0$ otherwise. The distribution over the optimality variable is written as: $p(O_t = 1|S_t, A_t) = f(r(S_t, A_t))$, where f is a function

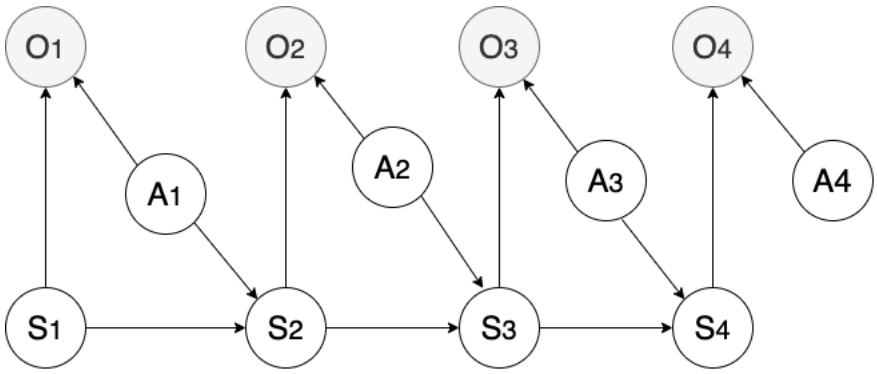


Figure 2.1: A Representation of the Hidden Markov Model

that maps rewards r to a probability value. This structure is presented in Figure 2.1, and is analogous to HMMs. Thus, the VFA is obtained using the backward message passing function of the form: $\beta(S_t, A_t) = p(O_{t:T}|S_t, A_t)$ [17, 18]. In this equation, $O_{t:T}$ represents the observation variables from time t until the end of the episode or the final state at T. The measure is proven to represent the probability inference view of RL, which is a generalization of the optimal control problem [19]. We use this quantity as a signal to speed the learning in the form of a potential function.

On the other hand, forward messages (α) are used to look back at the trajectory from $t = 0$ to $t - 1$. Thus, combining forward and backward messages allows the agent to view the outcome of the whole trajectory and induce its optimality, which is relevant for forming an effective reward shaping function. The forward message has the following form: $\alpha(S_t, A_t) = p(O_{t_0:t-1})|S_t, A_t)p(S_t, A_t)$. Thus, the combined messages is expressed as:

$$p(O_t|S_t, A_t) \simeq \alpha(S_t, A_t)\beta(S_t, A_t) \tag{3}$$

Based on [16], the potential function is expressed as $\phi_{\alpha,\beta} = \alpha(S_t, A_t) \times \beta(S_t, A_t)$

Forward and backward messaging is only possible for some MDP problems where the state space and the size of trajectories are small. To make the message passing scalable, GCN is proposed as a function of reward shaping in [16]. GCN is first used to perform

semi-supervised learning benefiting from its recursive nature and information propagation of labeled data to neighbors [20]. Transforming an MDP to a graph structure and applying GCN to perform message passing for reward shaping is possible. A basic form of two layers GCN is written as:

$$\phi_{GCN}(X) = Softmax(\mathrm{T}\ ReLu(\mathrm{T}XW_1)W_0) \tag{4}$$

where $ReLu$ and $Softmax$ are the activation functions, T is the GCN filter used to define the graph connections, X is the input matrix, and W_0, W_1 are the weights of each layer. As shown in Equation 4, T is used in the GCN calculations for propagating information from the neighbors. In the context of reward shaping, T is an approximation of the transition matrix. In [16], the graph Laplacian is used as an approximate of the transition matrix. This is motivated by the proto-value function framework in the RL literature, which uses the graph Laplacian as a surrogate of the transition matrix due to its smooth approximate to the value function of an MDP following the spectral graph theory [21, 22].

As argued in [23], the graph Laplacian is guaranteed to form an approximate estimate of the value function when assuming that the latter is smooth over the induced MDP graph. In order to resolve the issue of smoothness for the value function, diffusion wavelets can be used. However, diffusion wavelets require a matrix inverse operation, which is very expensive to compute. In addition, the work in [23] derived that using the actual transition matrix instead of the adjacency matrix leads to a smaller margin of error for forming the VFA. Henceforth, the use of the Krylov basis instead of the graph Laplacian for approximating the value function is a better option. The Krylov basis is formed using the augmented Krylov algorithm. In this thesis, we argue in Chapter 5 that the Krylov basis can be used for approximating the transition matrix for the GCN propagation. To predict the future reward shaping value, we propose the use of GCRN that combines GCN and Bi-GRUs to study the spatio-temporal dependencies between the nodes of the graph. In the next subsection, we

provide more details about the augmented Krylov algorithm and the Krylov basis.

2.1.5 Reward Shaping

The idea behind reward shaping is to apply expert knowledge for directing the reward function. This is done by appending scaler values, from the shaping function F, to the reward function so that it takes the following form:

$$R(S_t, A_t, S_{t+1}) = r(S_t, A_t) + F(S_t, S_{t+1}) \tag{5}$$

In this equation, $F(S_t, S_{t+1})$ is the shaping function that takes as input the states at t and $t+1$ from $\mathcal{S}$. Because it is important to preserve the policy when applying reward shaping, F is written as [24]:

$$F(S_t, S_{t+1}) = \gamma\phi(S_{t+1}) - \phi(S_t) \tag{6}$$

where ϕ is a potential-shaping function that outputs a scaler value. Existing solutions that form shaping functions either rely on human intervention [25], or cannot scale well in complex environments [26]. On the other hand, in [16], the authors propose the use of GCN as the shaping function, which resolves the issue of scalability. However, [16] does not consider the drawback of using the graph Laplacian as the GCN filter and neglects the importance of studying the temporal dependencies between states of the graph. Moreover, it is important to consider the option of adding the look-ahead feature when computing the reward shaping values using the potential function [24].

2.1.6 Look Ahead Advice

The look-ahead advice is a reward shaping technique proposed in [24] without altering the optimal policy. The shaping function usually takes the form of Equation 6, allowing

the agent to send feedback after observing the rewards of the states only. A more rigorous advice is given by building the reward shaping on both states and actions. Following the look-ahead advice proposed in [24], the shaping function takes the following form:

$$F(S_t, A_t, S_{t+1}, A_{t+1}) = \gamma\phi(S_{t+1}, A_{t+1}) - \phi(S_t, A_t) \tag{7}$$

where ϕ is a function of the state from $\mathcal{S}$ and action from $\mathcal{A}$ that produces a scaler value. Hence, the reward shaping function is expressed as follows:

$$R(S_t, A_t, S_{t+1}, A_{t+1}) = r(S_t, A_t) + F(S_t, A_t, S_{t+1}, A_{t+1}) \tag{8}$$

The look-ahead advice in the reward shaping decisions could further speed the learning process by augmenting the action values and affecting the action selection [24].

2.1.7 Augmented Krylov

In [23], the authors analyze the performance of the graph Laplacian for computing the function approximation. A normalized graph Laplacian is widely used and expressed as $L_c = I - \mathfrak{D}^{\frac{-1}{2}}\mathfrak{A}\mathfrak{D}^{\frac{-1}{2}}$, where $\mathfrak{D}$ and $\mathfrak{A}$ are the degree matrix and the adjacency matrix respectively, and I is the identity matrix. Besides, L_c is adapted in the original GCN work to resemble a filter connecting the nodes of the graph. Furthermore, GCN with L_c is used to generate the potential function for reward shaping in RL, where the adjacency matrix forming L_c is extracted from the graph of states [16]. In the same context, we argue in this work that the vectors extracted from the augmented Krylov algorithm, which we refer to as Krylov basis, outperforms the use of L_c as the GCN filter for forming the potential function. The results of the analysis in [23] show that the graph Laplacian forms an effective approximation when assuming that the adjacency matrix is symmetric (i.e., the value

function is smooth over the graph of states). Such an approximation can be defined by considering the bottom eigenvectors of the graph Laplacian to approximate smooth functions with a low Sobolev norm. The final step is to aggregate these vectors to form the VFA.

According to [23], using the adjacency matrix to approximate the value function is not well motivated. In fact, using the actual transition matrix P^{π} leads to a smaller margin of errors when approximating the function compared to using the graph Laplacian. Using the spectral approximation of VFA, we can compute the top eigenvectors of P^{π}, which are considered as the approximation vectors. The algorithm for computing these vectors is referred to as the *weighted spectral method*. There is also another motivated base choice, which is using m vectors from the Neumann series, where m is the degree of the minimal polynomial of $(I-\gamma P)$ [27]. These extracted vectors form the Krylov space to approximate the VFA, denoted as $\mathcal{K}(P, r)$.

A practical implementation of the *weighted spectral method* faces issues in regards to the complexity of computing the eigenvectors. In addition, Jordan-decomposition should be calculated when dealing with a non-diagonalizable transition matrix, which is time consuming, and there might be complex numbers in the computed eigenvalues and eigenvectors. In order to overcome these problems, the augmented Krylov algorithm is used, which combines the benefits of both approximations. In particular, the Krylov space captures the short-term behavior, while the top eigenvectors of the transition matrix capture the long-term behavior [23]. These features are well suited for our proposed GCRN for propagating the short and long term impacts of the messages in GCN. In augmented Krylov, the Krylov basis is built using the top eigenvectors of P followed by m vectors of the Neumann series.

2.1.8 Bi-Directional Gated Recurrent Units

The purpose of reward shaping is to speed the learning. Thus, predicting the reward shaping value of the next state can further improve the effectiveness of any potential function. To do this, we propose in Chapter 5 adding a bi-directional RNN that processes the reward shaping values from $t = 0$ to T to predict the shaping value at $T+1$. In other words, we use the bi-directional RNN to learn the temporal dependencies between the states and transform it to a shaping value. The loss function in this network relies on using the message passing to compute the actual labels. In the sequel, we describe the mechanism behing the Bi-GRUs.

LSTM and GRUs are RNN variants that use the gated mechanism to expand the memory capabilities. Both LSTM and GRU excel in different domains. LSTMs are more complex and require more training time; however, GRUs have simpler structures with less number of parameters and less time to train. For these reasons, GRUs are more useful in the context of reward shaping. Besides, Bi-GRUs contain hidden layers for studying the sequences in forward and backward directions, which is better suited for our problem.

2.1.9 Convolutional Neural Network

The CNN network is proven to be successful in computer vision and natural language processing applications. Furthermore, CNN layers extract features from the images as it goes deeper into the network. The input layer in a CNN takes as input raw image pixels which are three-dimensional. A CNN is composed of convolutional layer, non-linearity layer, max pooling layer, and fully connected layer. At each convolutional (conv) layer, there is a kernel matrix or filter with a certain size. Setting the filter size is considered a hyperparameter. A filter for the current layer is applied on the input matrix, where matrix multiplication takes place. In addition, a sliding window or stride on the input matrix can be applied so that more information about the image can be concluded. Adding more layers

with various filters results, in most cases, in more feature extraction from the input matrix. Furthermore, padding can be added to the images to not lose any information present at the frame or edges of the matrix.

2.1.10 Value Iteration Network

A VIN incorporates planning inside a policy of the original MDP $\mathcal{M}$ by performing value iteration using a CNN [28]. It is assumed that VIN tries to learn and solve another MDP $\bar{\mathcal{M}}$. Similar to any MDP, $\bar{\mathcal{M}}$ has states, actions, reward and transition functions denoted as $\bar{s} \in \bar{\mathcal{S}}, \bar{a} \in \bar{\mathcal{A}}, \bar{\mathcal{R}}(\bar{s}, \bar{a})$, and $\bar{\mathcal{P}}(\bar{s'}|\bar{s}, \bar{a})$ respectively. The state and action spaces in $\bar{\mathcal{M}}$ are similar to $\mathcal{M}$. The reward and transition matrices of $\bar{\mathcal{M}}$ depend on the observations of $\mathcal{M}$, i.e. $\bar{\mathcal{R}} = f_r(\phi(s))$ and $\bar{\mathcal{P}} = f_p(\phi(s))$. The functions of the reward and transition (f_r and f_p) are learned during the policy training of $\bar{\mathcal{M}}$ using the CNN. The learned policy of $\bar{\mathcal{M}}$ is connected to obtaining the optimal policy of $\mathcal{M}$, even though the reward and transition functions are not the same. The input to the VIN model is a list of images extracted from the environment. The first layer in the CNN of VIN is a convolution (Conv) layer that processes raw pixels and pass them to the second layer. The first step in value iteration at the second conv layer is to convert the input of the previous layer into a reward matrix using the reward function f_r. The filter/kernel of this second conv layer is considered as the probability transition matrix of $\bar{\mathcal{M}}$. In addition, the third conv layer contains the $\mathcal{Q}$ value or state-action value function over the channels of this layer for $\bar{\mathcal{M}}$. Finally, this last layer is max-pooled to produce the next value iteration $\bar{\mathcal{V}}$.

The output of VIN is only for a subset of states. Therefore, the output is passed to an attention module that helps reduce the number of parameters to train or the actions to focus on. Furthermore, the output of the attention model is passed to the policy update of $\mathcal{M}$ to guide the model in selecting better actions. The CNN is trained using the standard backpropagation to support RL or IL decisions.

2.2 Litearture Review

In this section, we overview the existing literature work for (1) intelligent resource management, (2) bootstrapping in RL, and (3) reward shaping.

2.2.1 Computing Resource Management

Computing Resource management includes problems related to services distribution, cashing, and computation offloading. Resource management is needed at any of the well-known layers of cloud, fog, and edge computing. Table 2.1 overviews some of the latest research on resource management from multiple disciplines in computing. In this table, we compare the existing literature work based on nine different metrics. Horizontal and Vertical scaling indicate if the work considers scaling actions. Availability is to indicate if proactive actions are taken in order to account for the demands for services, tasks, or cached content.

There exist literature work that considers horizontal and vertical scaling while predicting the change of demands for certain services, tasks, or content. Furthermore, intelligent solutions are used to adapt to environmental changes using Machine Learning (including RL). Furthermore, some of the existing work considers service placement as a problem. All these features are provided by [9, 10, 45, 46]; however, none of them considers proactive service placement to account for availability, offers prediction to resource changes, or performs offline learning through bootstrapping. Moreover, some research efforts propose solutions for horizontal resource scaling [41, 42], with demands prediction [43, 44] or service placement [47]. Other solutions do not consider intelligent decision-making but still consider proposing methods for solving the service placement problem [33, 34, 35, 36].

Table 2.1: Table of Comparison Between Latest Service Placement and Scaling Solutions

Solution	RL-Based	Heuristic-Based	Proactive	On-Demand	Scalable	Supports Multi-Apps	Horizontal Scaling	Vertical Scaling	Resources Prediction	Model Bootstrapping
[29]	✓	-	✓	✓	-	✓	-	-	-	-
[30]	✓	-	-	-	✓	✓	-	-	-	-
[1]	-	✓	-	✓	-	✓	-	-	-	-
[15]	-	✓	-	✓	-	✓	-	-	-	-
[31], [32]	-	✓	-	-	-	-	-	-	-	-
[33, 34]	-	✓	-	-	-	✓	-	-	-	-
[35, 36]	-	✓	-	-	-	✓	-	-	-	-
[37]	✓	-	-	-	-	-	-	-	-	-
[38]	-	✓	-	-	-	-	-	-	-	-
[39, 40]	✓	-	-	-	✓	-	-	-	-	-
[41, 42]	-	-	-	-	-	-	✓	-	-	-
[43, 44]	-	-	✓	-	-	-	✓	-	-	-
[9, 10, 45, 46]	✓	-	✓	-	-	-	✓	✓	-	-
AWS EKS	-	-	✓	✓	✓	✓	-	✓	-	-
AWS Auto Scaling	-	-	✓	-	✓	✓	✓	-	-	-
Azure AKS	-	-	-	✓	✓	✓	✓	-	-	-
Azure AutoScale	-	-	✓	-	✓	✓	✓	-	-	-
Google GKE	-	-	-	✓	✓	✓	✓	✓	-	-
Our Solution	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Classical Solutions

Classical solutions do not employ intelligent or machine-learning-based solutions. For instance, the authors in [42] deploys a space-search pruning algorithm to find the best edge server for migration and scaling. Despite that the complexity of the search algorithm can grow exponentially in the worst case, the solution has to wait for the demands to occur to make a decision. On the other side, the authors in [41] measure the system state and classify its workload into low, medium, and high based on predefined thresholds. Similar to [42], downtime or degradation of QoS can happen while scaling resources. In addition, the work in [47] proposes the use of a heuristics search algorithm to perform the resource scaling in a cloud environment. The limitation in [47] is the use of a heuristic-based solution to perform the scaling after the increase in demand occurs, which directly affects the QoS. Besides, a heuristic solution does not guarantee optimal solutions.

Machine Learning Solutions

There are many recent proposals that utilize machine learning to solve wireless and resource management problems. In this section, we focus on the RL-based solutions, which outperform classical machine learning solutions due to their ability to perform linear and non-linear approximations for the state-action value function, adapt to environmental changes, and learn without prior knowledge. The main RL applications in the context of resource provisioning include network resource management, computational resource scaling, wireless network security, and content caching [48]. DRL solutions exist for each application under different fields, such as the internet of vehicles, unmanned aerial vehicles, cloud, edge computing, and cellular technologies (5G & 6G) [48]. With regard to network management, DRL is used for solving the problem of resource management for network slicing [49]. Besides, DRL is also exploited in [50] for securing the wireless communication at the physical layer by adjusting the agent's reflecting elements with a base station. In

[37], RL-based linear function approximation is used for content caching on base stations in the context of 5G based on the change of users' demand. In the same context, the authors in [9] builds a Markov Decision Process (MDP) by defining the states, actions, reward function, and retrieving the probability transition function. In their work, a Model-Based RL solution is proposed to take the scaling decision while knowing the probability transition function through period updates. As shown in Table 2.1, the main limitation of [9] is the risk of application downtime. Downtime can happen in two cases. First, the probability transition matrix extracted might not be representative enough of the dynamics of the environments. Therefore, wrong scaling decisions are possible to be made. Second, in case the state space grows, it becomes impossible to estimate the probability transition function because this will require huge memory on the processing machine.

More recently, several approaches, such as [43] and [44], are working on improving the prediction of workload forecasting, which leads to accurate scaling decisions if successful. However, these time-series forecasting approaches still look for seasonality and pattern in the data studied, which drains its accuracy in case new patterns are encountered.

Industry-Based Solutions

Dynamic resource scaling is mandatory in any clustering environment that hosts services or executes computing tasks. The leading industry companies that offer cloud platforms offer the service scaling feature. Examples of these cloud solutions are Google Cloud Platform, Microsoft Azure, and Amazon Web Services (AWS). These solutions integrate the Kubernetes clustering tool to benefit from the orchestration and embedded scaling features, thus offering new environments titled Google GKE [1], Azure AKS [2], and AWS EKS [3]. As shown in Table 2.1, these environments offer vertical, horizontal, or both scaling, as

[1] https://cloud.google.com/kubernetes-engine
[2] https://azure.microsoft.com/en-us/services/kubernetes-service/
[3] https://aws.amazon.com/eks/

well as availability because of the multi-zone hosting of services inside Kubernetes clusters. The main limitation of these environments is that the demands of services, such as resource load or response time experienced by the user, are not predicted. These solutions either rely on thresholds or manual configurations similar to Azure AutoScale, which runs per application instance [4]. Such problems are resolved by a platform-native solution for AWS titled Auto Scaling, which is independent of Kubernetes [5]. In AWS Auto Scaling, time series prediction takes place to be able to scale the application instance before actual demands occur. As mentioned earlier, this method is not reliable because the time-series model might not be able to capture seasonality or signature in the demands.

Resource load is not an accurate measure of the scaling decision in cluster-based environments; however, most of the recent research and industry approaches are utilizing this metric at the service instance and cluster levels. Moreover, as shown in Table 2.1, some solutions do not consider service placement because either they perform vertical scaling only or they run the horizontal scaling inside the same hosting machine. Additionally, the resources available on the servers may vary because of hosting several other independent applications. However, resource prediction is not considered by the aforementioned state of the art solutions.

2.2.2 Bootstrapping for Reinforcement Learning

Bootstrapping is an existing field of research that gained attention recently with the increase of RL use to solve challenging problems. Challenging problems are usually sensitive to high costs and require bootstrapping solutions to avoid high exploration costs. Using a dataset from storage, offline learning is performed to boost the performance of RL at the first stages of model building. The literature contributions vary concerning the learning solution used offline on the stored history of experiences. The authors in [51] study the

[4] https://azure.microsoft.com/en-ca/features/autoscale/
[5] https://aws.amazon.com/autoscaling/

difficulties of performing offline learning of a model using any history of experiences, then fine-tune the policy online. The authors propose an actor-critic bootstrap solution for a smoother transition between offline and online policy tuning by introducing constraint actor updates. Moreover, the work in [52] contributes by advising the use of transformers to generate more offline data to further boost the performance of the offline solution. The main problem addressed by [52] is the limited representation or distribution of the offline data, which necessitates the generation of more data with enhanced distribution. Some of the limitations of these methods are preventing the value function from generalizing and the misleading characterization of actions that are out of distribution [53]. In this regard, the authors in [53] propose an uncertainty estimation method for penalizing the value function in addition to a novel offline sampling method. The problem with data distribution of offline datasets could severely affect the bootstrapping criteria, which continues to be addressed in [54]. The authors in [54] propose a prioritization method of online experiences while training multiple Q-functions on various experiences to select the closest policy to the online settings.

The bootstrapping criteria proposed in the literature focus on the data distribution of the history of experiences, which we agree severely affects the model updates. In this proposal, we propose using heuristic-based methods to replace RL at the exploration stages. Using our approach, less burden is placed on the quality of historical data, and better decisions are produced using heuristics during exploration. Data and actions extracted using heuristics are utilized by the RL model to speed learning during exploration. We believe that our approach is the first in the literature to propose a heuristic-based bootstrapping solution for supporting RL.

2.2.3 Reward Shaping

Reward shaping has gained more attention in the past years due to the importance of speeding the learning process in Deep RL [55], especially in applications requiring real-time feedback. In this section, we first describe VIN and its applications. Second, we discuss existing reward shaping solutions and their limitations. Third, we discuss and present the limitations of recent literature solutions that utilize deep learning to build the potential function.

Value Iteration Networks and Applications

Solving numerical problems to obtain an optimized solution is a machine learning task [56], even when genetic algorithms are used [57]. In Chapter 6, we consider the use of VIN as the optimization solution to obtain the optimal policy for an RL solution using CNN. CNN is a type of neural network commonly used in image and video recognition tasks. Therefore CNN can be used to extract image features to be passed for RL solutions such as Deep Q-Network, Actor Critic, or PPO. Thus, CNN can be a part of the approximated function to compute the policy or value networks in RL. On the other hand, VIN is a specific neural network architecture designed for solving MDPs and forming the policy of the RL solution. In specific, VIN uses the CNN layers to compose the different components forming the RL policy update mechanism. In other words, VIN is composed of a set of convolutional and pooling layers that encode the information from the MDP, which can estimate the value function. In [28], a VIN module is proposed to perform planning on a new MDP extracted from images of the environment. The motivation behind VIN is to support planning in NNs by integrating a new value iteration method inspired by the policy updates in RL. The value of the work comes from using the convolution operators to perform value iteration on an unknown MDP that outputs an optimal trajectory. The states and actions of this MDP are the same as the original MDP, while the reward and transition

functions are any differentiable functions that can be learned from the model while training. One of the main limitations of VIN is that it can be applied using imitation learning which requires a lot of ground truth labels or RL that provides poor performance. Furthermore, VIN supports low-dimensional environments or MDPs only. Due to the importance of the planning feature in NNs, various works extended form [28] and proposed new differentiable reward and transition functions that can stabilize learning in the network. For example, the authors in [58] proposes a novel convolution operator to learn and plan on spatial and irregular graphs.

Various applications use VIN to combine planning with a supervised learning task to improve the quality of the decisions. For instance, the work in [59] utilizes VIN to perform UAV planning and adapt to novel physical locations. Furthermore, the work in [60] uses VIN to learn urban navigation planning. Besides, the authors in [61] offer a solution for traffic control between vehicles using VIN, which takes as input a graph of the traffic.

In this work, we propose a new variation of VIN, named VIN-RS, for the first time in the context of potential-based reward shaping. VIN-RS encodes a new training mode with message passing as part of the loss function. Our VIN-RS can effectively plan trajectories to speed learning in the original policy. VIN-RS takes images of the environment as input to output shaping values. The shaping values are used in the form of potential-based reward shaping by appending the original reward function.

Existing Reward Shaping Methods

There exist various reward shaping solutions that (1) alter the optimal policy, (2) target action exploration instead of reward signal, or (3) requires human intervention. For instance, Learning Intrinsic Rewards for Policy Gradients (LIRPG) proposes an optimal reward framework and does not guarantee invariance of the optimal policy [62]. The Random Network Distillation (RND) approach provides an action exploration method to accelerate

learning [63]. RND uses an exploration bonus, which is calculated using the error of a NN that predicts the observation features. The proposed RND solution provides Superior performance in the challenging Montezuma Revenge game. In [64], authors propose the intrinsic curiosity module (ICM) to speed learning using action exploration. In ICM, the authors formulate exploration by the error of the agent when trying to predict the consequences of its actions [64]. In contrast to RND and ICM, our proposed VIN-RS offers reward shaping based on the reward signal and not the exploration bonus. In [16], the authors proposed a potential-based reward shaping solution that performs message passing using GCN. More details about the GCN solution are provided in the next subsection.

In LIRPG [62], the mean relative improvement over A2C baseline is 23% with a standard deviation of 12%, indicating significant variation in performance across different Atari games. The sensitivity of the optimal step size (β) and intrinsic reward scaling parameter (λ) impacts performance, showing variation with different β values.

For ICM [64], it achieves a 66% success rate in "very-sparse" reward settings, while A3C and ICM-pixels fail. The performance difference between ICM and ICM-pixels is attributed to the difficulty of learning pixel-prediction models with increasing textures. However, no specific quantitative analysis is provided for the superiority of ICM over ICM-pixels.

In GCN-based reward shaping [16], the impact of environment graph sampling on efficiency lacks specific statistics. A comprehensive comparison of different sampling methods would aid in understanding the approach's effectiveness. Additionally, there is no statistical analysis of the sensitivity of crucial hyperparameters α and β, warranting further investigation to assess their impact and robustness accurately.

Using GCN for Reward Shaping

Existing reward shaping solutions requires a neural network to become scalable and accommodate for dynamicity in the environment with large state or action spaces. For instance, the work in [26] suffers from scalability issues, while [25] demands human intervention to update the reward function with feedbacks. Therefore, we study the related work proposing to build the potential function through deep learning to overcome the mentioned problems. In specific, the most suitable deep learning models belong to the family of Graph Neural Network (GNN), such as GCN. Hence, we discuss the two most related work that uses GCN to build the value function [16, 65]. The GCN is capable of recursively propagate messages among neighboring nodes in the graph, determining its relation and importance. When using message passing of HMM, those messages reveal more information about the state or trajectory of state optimality, thus enlightening the original reward function about useful information among the selected path. In [16], an improvement of learning speed and reward achieved are presented for the first time when using GCN as a reward shaping function. Despite its performance, the presented mechanism suffers from various limitations including the representation of the sampled MDP as a sub-graph of sampled transitions, and the approximation of the transition matrix using graph Laplacian, which results in a margin of error affecting its performance.

The image-based approach allows VIN-RS to capture more comprehensive and detailed information about the entire environment, providing a broader perspective and a larger set of states compared to GCN's limited graph-based representation. Furthermore, using GCN to compute the message passing requires an approximation of the transition matrix. In this case, the graph Laplacian is utilized [16]. Due to many drawbacks of using graph Laplacian as analyzed in [23], reward shaping using GCN affects the performance. Several methods were proposed to construct bases for Value Function Approximation (VFA), such as using the graph Laplacian. It is proven that the graph Laplacian can only produce effective VFA

when assuming that the latter is smooth over the induced MDP graph. Therefore, using the graph Laplacian to approximate the transition matrix cannot generalize to all MDPs [23]. In our previous work [65], the graph Laplacian is replaced by a Krylov subspace computed using the augmented Krylov [23], as an attempt to overcome the graph Laplacian limitations. However, this method still cannot guarantee an improvement over GCN in some cases. Therefore, we replace in this work GCN by a CNN. In the proposed VIN-RS, the probability transition matrix within the convolution layers is learned when training the CNN network, thus avoiding the burden of approximating this matrix.

Chapter 3

Demand-Driven Deep Reinforcement Learning for Scalable Fog and Service Placement

The increasing number of Internet of Things (IoT) devices necessitates the need for a more substantial fog computing infrastructure to support the users' demand for services. In this context, the placement problem consists of selecting fog resources and mapping services to these resources. This problem is particularly challenging due to the dynamic changes in both users' demand and available fog resources. Existing solutions utilize on-demand fog formation and periodic container placement using heuristics due to the NP-hardness of the problem. Unfortunately, constant updates of services are time consuming in terms of environment setup, especially when required services and available fog nodes are changing. Therefore, due to the need for fast and proactive service updates to meet users' demand, and the complexity of the container placement problem, we present in this chapter a Deep Reinforcement Learning (DRL) solution, named Intelligent Fog and Service Placement (IFSP), to perform instantaneous placement decisions proactively. By

proactively, we mean producing placement decisions before demands occur. The DRL-based IFSP is developed through a scalable Markov Decision Process (MDP) design. To address the long learning time for DRL to converge, and the high volume of errors needed to explore, we also devise a novel end-to-end architecture utilizing a service scheduler and a bootstrapper on the cloud. Our scheduler and bootstrapper perform offline learning on users' demand recorded in server logs. Through experiments and simulations performed on the NASA server logs and Google Cluster Trace datasets, we explore the ability of IFSP to perform efficient placement and overcome the above mentioned DRL limitations. We also show the ability of IFSP to adapt to changes in the environment and improve the Quality of Service (QoS) compared to state-of-the-art-heuristic and DRL solutions.

3.1 Motivational Use Case

We consider the use case of a road with groups of autonomous vehicles performing self-driving, in addition to unmanned aerial vehicles (UAVs). Drivers and passengers are requesting different types of services using existing network protocols [66]. In particular, some drivers are requesting a service to retrieve real-time traffic information [67]. A smart vehicle is requesting services to collect more sensed data from vehicles around to improve its driving decisions [68]. Furthermore, the UAVs are requesting network management services [69, 70]. The passengers from their sides are interested in infotainment-related services such as video streaming and playing video games. Due to the limited computing resources on On-Boarding Units of the vehicles, on-demand fog computing is employed to improve the QoS experienced by the requesters. In this case, the fog computing cluster is initialized on volunteering edge servers, such as the Road Side Units (RSUs). In the on-demand fog context, services are not always lightweight. For example, downloading a guest operating system for hosting the traffic measurement service takes time. Besides,

initializing the Kubernetes cluster and downloading the required modules are also time-consuming. In addition, the moving vehicles and UAVs can leave the range of the serving RSU, leading to networking delays and reachability issues. In order to overcome this problem, proactive placement is required. Proactive placement can be performed by predicting the pattern of demands on each road by the RSUs. Furthermore, vehicles send different volumes of requests for services. Hence, a dynamic placement of services on the nearest RSU is required to meet the users' demand. Besides, the number of available resources running on the RSUs is changing due to other independent applications. Hence, the demands and available resources should be predicted to perform optimal placement.

3.2 Proposed Architecture Realizing DRL

In this section, we elaborate on a proposed architecture tailored to enable the use of DRL to solve our service placement and fog selection problem. This architecture ensures a complete end-to-end intelligent and automated solution for service placement and fog selection to improve QoS. A presentation of the architecture and components interactions is depicted in Fig. 3.1. The two core layers of our architecture are the cloud and fog layers. In this architecture, users can be any IoT devices. As shown in Fig. 3.1, users start by requesting services from the cloud. Applications on the cloud are hosted inside computing engines that are dedicated to hosting the back-end logic of the applications. Every application has a logging logic that keeps track of connected users, the source IP, and the requested APIs or services. These logs keep on updating, which means that new demands will always be taken into consideration while our models are learning. Our proposed architecture will then fully rely on these logs as a source of data to learn and make decisions. These data are then utilized by IFSS (Intelligent Fog Service Scheduler) and Bootsrapper, which are two intelligent components running two different reinforcement learning algorithms.

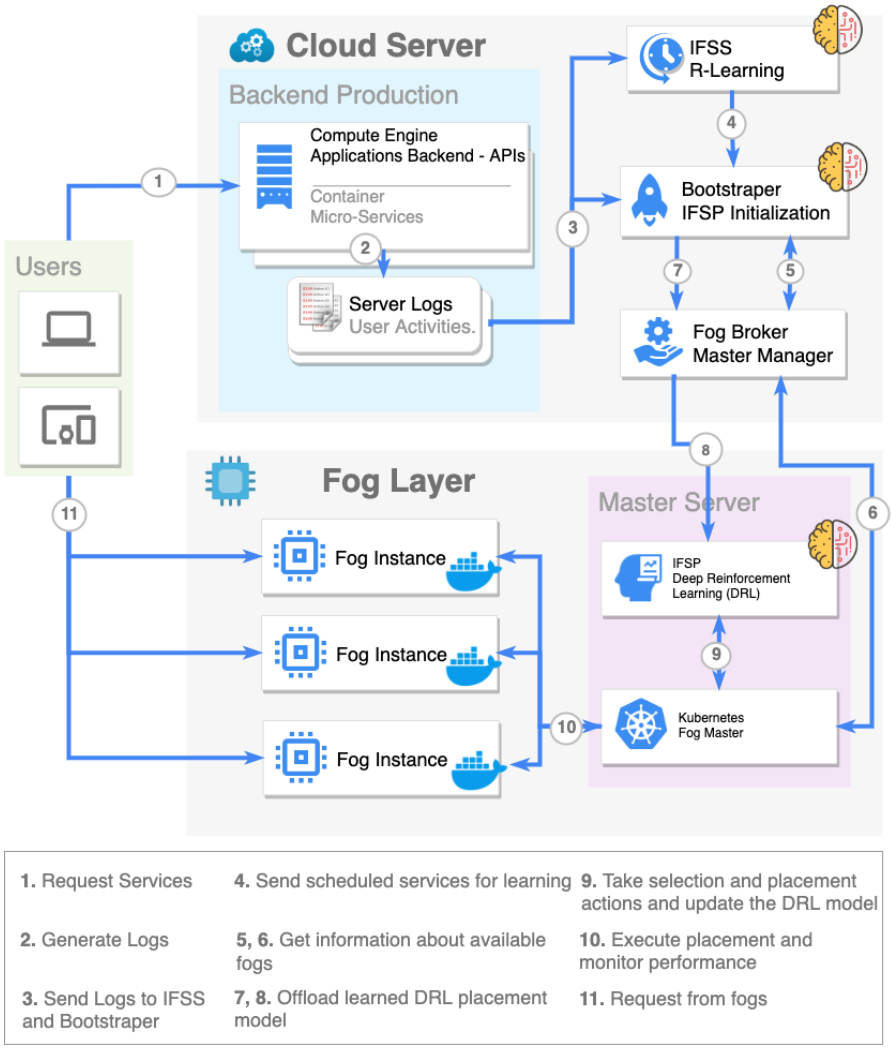


Figure 3.1: Proposed Architecture Realizing DRL

The IFSS agent runs an R-Learning algorithm that uses an MDP formulation for deciding about the best time and place for pushing services to fogs. IFSS was modeled and implemented in our previous work [6]. IFSS learns from the server logs the different demands of users divided by location and based on the time of the day. The decision of the IFSS scheduler is proactive, which means a decision for placing a service is taken before the actual demands occur. The bootstrapper then receives the scheduling decision by the IFSS agent, which contains the list of locations, each having a list of services to place for the given time. The Bootstrapper on the cloud runs a DRL algorithm that takes as input a

state built using the services to be placed, the fogs available in the target location, and the changing demands of users over time retrieved from the server logs. The MDP for the agent is discussed in Section 3.2. In contrast, the DRL algorithm is presented in Section **??**. The primary purpose of this model is to perform offline learning on all the demands captured in the log files using the DRL algorithm. This model is then considered as a bootstrapping for the IFSP (Intelligent Fog Service Placement) running on the master, which performs online learning. IFSP will receive a mature model and avoid the long learning time and errors the model yields at the first stages of learning.

The Fog Broker is responsible for managing all the communication between the cloud and the set of master nodes available anywhere. The Fog Broker is the gateway of the cloud to the Internet. It is horizontally scalable, keeps track of all available master nodes, and ensures reliable connections. The broker also reaches periodically to all the master nodes of the different clusters about their available fog nodes, the resources capacity, and the geographical locations of each. The Bootstrapper then utilizes this information as a requirement for the model to start offline learning. Once the Bootstrapper finishes modeling the environment and builds the DRL agent by achieving convergence, it forwards this model using the Fog Broker to the master node.

The fog layer consists of the fog clusters, each having a master node responsible for orchestrating fogs and managing the running services on each one. Each master node contains an IFSP and the Kubernetes required model for creating and maintaining a fog cluster which relies on the containerization technology. The master expects the broker's requests, which contain the new IFSP model to be adapted in its cluster. The received model is mature, and the master can rely on to make selection and placement decisions. Because user demands are stochastic, the master will also run an online update for each decision made. Using the Kubernetes model, the master has knowledge of the demands for services on each fog and how it changes over time. This information is fed for the IFSP

for the online updates. Kubernetes is also used to take the actions generated by decisions from the IFSP including placing and updating containers on available fogs. Fog nodes will be able to run services that are of best use for users and therefore improving the QoS of the applications. Thus, users will migrate their requests from the cloud to available fogs.

3.3 IFSP Modeling

Following our proposed model, IFSP can be used in fog clusters with confidence that the deployed model online will act maturely and keep on improving itself as new demands are measured. In this section, we present a novel MDP model that is capable of taking the selection and placement decisions proactively, takes into account the changing demands per service, considers four contradicting objectives, and is scalable, which means it can model hundreds of fogs and containers as input.

3.3.1 Background

MDP is a mathematical framework for modeling sequential decision making in stochastic environments. An MDP is characterized by the following tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{C}, \gamma)$. $\mathcal{S}$ is the set of states that the agent can be at, where $\mathcal{S} = \{s_1, s_2, \dots\}$. $\mathcal{A} = \{a_1, a_2, \dots\}$ is the action space or the set of possible actions that can be taken by the agent at each step. $\mathcal{P}r$ is the probability transition matrix or the probability distribution over the successor states s' after taking an action in $\mathcal{A}$. In case $\mathcal{P}r$ is known, the agent then has a model of the environment. In most cases, $\mathcal{P}r$ is not given and the use of model-free RL is essential to obtain a model of the environment. $\mathcal{C}$ is the cost function designed to measure how well the agent is doing after taking an action from $\mathcal{A}$ at a state s and moving to a state s'. Finally, γ is a decimal value from $[0, 1]$. The value of γ is usually close to one. γ is used to help the model converge by discounting over the rewards of the next states. In other words, γ

tells how much the agent cares about the reward of the future states. In order to model our problem as MDP, we need to define $\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{C}$, and γ.

3.3.2 States and Actions Modeling

The decision taken by the IFSP agent takes place at different time moments t, where $t = 1, 2, \ldots$. Let $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$ be the list of m fogs, where each fog has a list of available CPU, memory, disk, and geographical distance from the user. A fog F_i in the cluster can be represented as a vector $F_i = [F_{i_{cpu}}, F_{i_{mem}}, F_{i_{disk}}, F_{i_d}]$, where F_{i_d} is the mean of distances between the location of each user in a target area and the fog location. In other words, F_{i_d} measures the proximity of F_i to requesting users in the area. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be the set of n containers to place. Each service P_j has resources requirements and a priority value indicating that this service should be prioritized for placement to maintain a certain level of QoS. A service P_j has the following requirements for deployment $P_j = [P_{j_{cpu}}, P_{j_{mem}}, P_{j_{disk}}, P_{j_k}]$, where P_{j_k} is the priority of service P_j having a value of either zero for low priority or one for high priority.

Let $q(t)$ be the vector of normalized number of requests for every service in $\mathcal{P}$. An element in $q(t)$ for service P_j is denoted by $[q(t)]_{P_j}$ and is calculated as follows:

$$[q(t)]_{P_j} = \frac{\text{Number of requests for service } P_j \text{ at time } t}{\text{Total number of requests for all services in } \mathcal{P} \text{ at } t} \tag{9}$$

The placement decision is taken sequentially for each container per state at a time. The combined placement decision denoted by $k(t)$ at t is a binary matrix of size $m \times n$, where $k(t)_{ij} = 1$ means that P_j is placed on F_i, and 0 otherwise. Furthermore, a counter u is used to indicate the current container P_u the agent is taking the decision for, such that $u \in \{1, \ldots, n\}$. After making n decisions, the counter is reset to one. Henceforth, the state

of our model is:

$$s(t,u) = [q(t), k(t), u] \tag{10}$$

Selecting an action from the set of possible actions $\mathcal{A}$ in our MDP allows the agent to take a placement decision for the current container P_u. The possibilities in $\mathcal{A}$ are (1) selecting a fog from $\mathcal{F}$; (2) selecting a container from $\mathcal{P}$; or (3) doing nothing. In case a fog is selected, the action performed is to place P_u on this fog. On the other hand, in case a container is selected, this container is removed from it's current running fog and replaced by P_u. Mathematically, $\mathcal{A} = \{0, f_1, f_2, \ldots, f_m, p_1, p_2, \ldots, p_n\}$, where zero means that container P_u is not assigned to any fog, f_i means the fog F_i is selected for placement, and p_j means an already placed service P_j is unplaced from its fog and replaced by the service P_u. We also denote by $a(t)$ a typical element of $\mathcal{A}$ at t. It is important to emphasize that the list of services for replacement in the action space are the ones that were placed in previous time-steps. This list of services can be extracted from $k(t)$. Therefore, there are some infeasible actions which are discarded from the actions list based on the current state. The main motivation behind considering one container to place at a time is to make the MDP design capable of handling large inputs, in contrast to [37] where the action space can grow exponentially. Moreover, placing one service at a time guarantees more availability as the other services will keep running.

3.3.3 States Transition and Model Dynamics

Using Fig. 3.2, we elaborate in this section on the evolution of the key quantities used to evaluate the cost function and build the next states for the agent. In our formulation, every episode is divided into time-steps where the agent takes the placement decision for a single container. Consequently, during every episode, the agent takes separate placement decisions for each container in $\mathcal{P}$. The action taken at state $s(t,u)$ is $a(t+1)$, which is a preparation for the coming request at state $s(t+1, u^+)$ where $u^+ = u+1$ if $u \leq n-1$ and

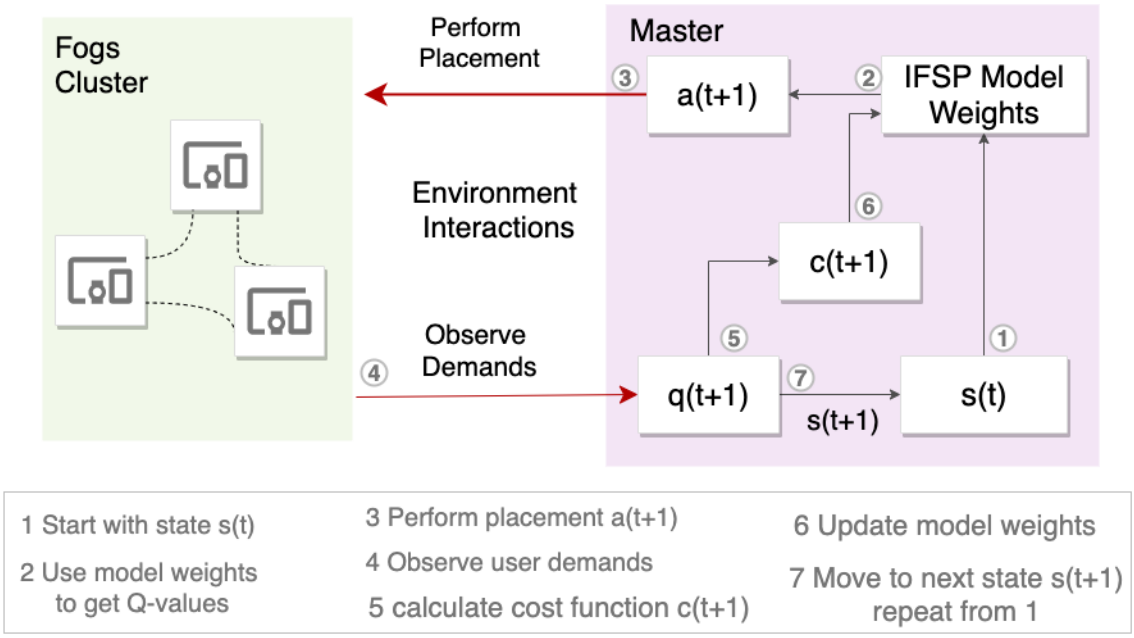


Figure 3.2: The Evolution of the Main Quantities Used for Cost Calculation over Time

$u^+ = 1$ if $u = n$. Thus, u is incremented by 1 and reset to 1 in case $u = n$. After taking the action $a(t+1)$, $k(t+1)$ is extracted by updating $k(t)$ following the action taken. The agent then waits to observe $q(t+1)$ to be able to calculate the cost function $\mathcal{C}$ at the next state. The process of calculating $\mathcal{C}$ is elaborated in the next subsection. It is important to mention that the time difference between the time-steps is short so that the agent will be able to update the placement for all containers and learn different patterns of demands. The state space grows to cover all possible combinations of demands, resulting in more robust placement decisions.

Knowing that the loads for users' requests on services is unpredictable because of its stochastic nature, the design of our formulation helps the agent predict the common sequence of loads by considering the load of the next episode to calculate the cost of the combined decision for the current episode. Because of the dynamic change in demands, we use a model-free approach, which does not require an explicit modeling of the environment.

In case the agent is at state $s(t, u)$ and takes action $a(t+1)$, the model has to represent

the cost incurred by every previous state action in order to find the optimal policy for the new state at the next step. The calculation of the cost incurred and the formulation of the different objectives are presented in the forthcoming sub-section.

3.3.4 Cost Function

For an agent at a certain state, taking an action and moving to the next state is evaluated by considering four contradicting objectives. In this section, we elaborate on a mathematical formulation for each objective. The objectives are (1) minimizing unserved requests measured using unsatisfied demands per service; (2) minimizing the number of fogs selected for placement; (3) minimizing the number of not placed services with high priority; and (4) minimizing the distance of selected fogs from requesting users. A cost in our formulation is represented as $\mathcal{C}(s(t-1,u^-),a(t)|q(t))$, where $u^- = u-1$ if $u > 1$ and $u^- = n$ if $u = 1$. In the sequel, we present the mathematical formulation of the cost function which is a superposition of four sub-costs.

Throughout the cost function formulation, we denote by $g(t)$ a 1-dimensional binary list of size n, where $g_j(t) = 1$ means that the j^{th} service is placed on a fog in $\mathcal{F}$, and 0 otherwise. In addition, we denote by $r(t)$ a 1-dimensional binary list of size m, where $r_i(t) = 1$ indicates that the fog F_i is hosting at least one container from $\mathcal{P}$. $g(t)$ and $r(t)$ are extracted from $k(t)$ for each step in an episode. For each cost related to an objective, we assign a weight $\lambda_l \in [0,1]$ to it such that $\sum_{l=1}^{4} \lambda_l = 1$. These weights are adjustable depending on the service provider preferences, given that the higher the weight, the more the objective has impact.

For calculating the cost of $s(t-1,u^-)$, the first cost c_1 considers the cost of not placing requested services at the next state. The purpose of this cost is to motivate the agent to place services that will be demanded in the next time-step. Following this approach, the agent learns the pattern of how the demand of services changes over time. This objective ensures

maximizing the number of satisfied requests served by the fog cluster; hence, leading to a lower response time, higher throughput and thus better overall QoS. This cost can be mathematically expressed as follows:

$$c_1 = \lambda_1(\mathbb{1} - g(t))^\intercal q(t) \tag{11}$$

Following Equation 11, $\mathbb{1} - g(t)$ results in a binary vector with 1 indicating that the container is not placed on any fog. Therefore, Equation 11 sums the loads ($q(t)$) for all unplaced services. This cost motivates the agent to satisfy as much demands as possible in order to minimize the cost.

The second cost, c_2, ensures that services with higher priority are considered in the placement decision in order to maintain an acceptable QoS level for all high priority services. High priority services do not necessarily have high demand. c_2 is then calculated using the below equation:

$$c_2 = \lambda_2 \mathcal{P}_k^\intercal [\mathbb{1} - g(t)] \tag{12}$$

where $\mathcal{P}_k$ is the vector of priority levels (0 or 1) for all services in $\mathcal{P}$. In Equation 12, we take services that are not placed by the decision at t, and some those with high priority. The aim for the agent is to minimize the total sum resulted by c_2.

In the third cost, c_3, we aim at minimizing the distance from selected fogs for placement and users requesting services to be placed. This preserves a very important feature for the fog layer, which is bringing services as close as possible to users. Respecting this objective leads to a lower response time experienced by users, therefore a better QoS. This cost is calculated using the following formula:

$$c_3 = \lambda_3 \mathcal{F}_d^\intercal N(t) \tag{13}$$

where $N(t)$ is a vector of size m indicating the total count of containers placed at each fog in $\mathcal{F}$ and $\mathcal{F}_d$ is the vector of mean distances of each fog in $\mathcal{F}$ to the users. Thus, c_3 computes the total sum of mean distances for all fogs used times the number of containers hosted on each. The end goal is to minimize this sum to ensure that running services on selected fogs are as close as possible to users.

The objective of the fourth cost, c_4, is to minimize the number of fogs used for a placement. This helps minimize the cluster complexity and the load on the orchestrator, which is responsible of managing all services running and the health of every fog. This objective leads to faster learning of changing fog resources and optimal placement, therefore improving the QoS experienced by the users. This cost is calculated as follows:

$$c_4 = \lambda_4 r(t)^\intercal \mathbb{1} \tag{14}$$

In Equation 14, we sum the number of fogs that are used for placement, by summing the 1's in $r(t)$.

Finally, the agent is allowed to make placement decisions that are not feasible, however, it's prompted to learn from it's mistakes through a punishment technique. For instance, an action is deemed infeasible if the agent overloads a fog by utilizing more than its available capacity. Thus, we calculate the CPU punishment score for the agent using the below equation:

$$p\_score_{cpu} = \sum_{i=1}^{m} \max(\sum_{j=1}^{n} (P_{j_{cpu}} k(t)_{ij}) - F_{i_{cpu}}, 0) \tag{15}$$

In Equation 15, $\sum_{j=1}^{n} P_{j_{cpu}} k(t)_{ij}$ is equal to the total CPU required by the containers and asked to be hosted on F_i. Our p_score_{cpu} calculates the excess of CPU utilization on F_i to be added to the total cost. Similar equations apply for calculating p_score_{mem} and p_score_{disk}, which are the punishment scores for memory and disk excess, by simply replacing the index cpu by mem and $disk$ respectively. Therefore, the punishment score is

the sum of the three scores following Equation 16.

$$p\_score = p\_score_{cpu} + p\_score_{mem} + p\_score_{disk} \quad (16)$$

Following the calculation of the four sub-costs and the punishment score, our cost function for evaluating the agent action is expressed as follows:

$$\mathcal{C}(s(t-1,u^{-}),a(t)|q(t)) = \sum_{l=1}^{4} c_l + p\_score \quad (17)$$

This cost is a combination of different measures that are mainly used to evaluate the QoS level of the user in the fog environment. In our case, minimizing the function $\mathcal{C}$ implies optimizing the QoS. Therefore, we use the cost function as a measurement of the QoS level experienced by the users in our experiments.

3.4 IFSP Using Deep Reinforcement Learning

The IFSP agent interacts with the environment for evaluating the placement action taken for each container. The agent executes actions for every state encountered and builds a strategy that adapts to the stochastic changing demands of users requesting services. The end goal of the agent is to learn the transition probability distribution from a state to all next states and find the optimal policy π^{*}, which takes as input a state and outputs the action that minimizes the future cost. In other words, π^{*} is a strategy or a set of actions the agent takes to minimize the cost. The future costs are discounted by γ, which controls the effect of future actions on past and current states and helps achieve the agent's mathematical convergence. By letting $\mathbb{C}(s(t,u),\pi)$ be the cost implied by choosing policy π from t that indicates the following actions $a(t^{'})$, such that $t \leq t^{'} \leq \mathcal{T}$ where $\mathcal{T}$ is the final time-step of

the episodes, the future discounted cost is represented as:

$$\mathbb{C}(s(t,u),\pi) = \sum_{t'=t}^{\mathcal{T}} \gamma^{t'-t} \mathcal{C}(s(t',u'),a(t')|q(t')) \tag{18}$$

where u' is updated to u^+ at each t'. We denote by $Q^*(s,a)$ the optimal action value function which minimizes the average expected cost for any selected strategy. It can be expressed as:

$$Q^*(s,a) = \min_{\pi} \mathbb{E}[\mathbb{C}(s(t-1,u^-),\pi)|s(t-1,u^-) = s, \\ a(t) = a, \pi] \tag{19}$$

The optimal Q-function selects the action of the next state that minimizes the action value function following the below equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}[\mathcal{C} + \gamma \min_{a'} Q(s',a')] \tag{20}$$

where $\mathcal{C}$ is the immediate cost from Equation 17 and $\mathcal{E}$ is the state at $\mathcal{T}$. The basic form of RL is to find the optimal action value function using iterative updates following the Bellman equation. This update can be expressed as:

$$Q(s,a) := Q(s,a) + \alpha[\mathcal{C} + \gamma \min_{a'} Q(s',a')] \tag{21}$$

where α is the learning rate. In Equation 21, the update of the Q-function happens following the Q-learning algorithm [71]. All Q-values are stored in a table structure containing the list of states and actions. An exploration-exploitation trade-off aids the agent into interacting with the environment by covering the maximum number of possibilities, observing the cost signal, and updating the Q-values using Equation 21.

However, the use of tabular RL is not practical in our problem, where we have a large state space. The state-space can grow with an increase in the number of containers and hosts to place. Thus, handling the whole table in memory, trying to cover all possible actions for every state, and updating the Q-values for all of them is computationally very expensive. Such an implementation is time consuming and makes any tabular RL agent diverge [72]. As a solution, learning the optimal Q-values can be retrieved from some adjustable weights denoted as θ. These weights get updated using gradient descent to update the weights downwards towards the direction of the gradient for minimizing the error of the calculated Q-values for every iteration. The common form of approximation is the linear function approximation which generalizes the environment through its weight, where the Q-function becomes close to the optimal Q^* having $Q^*(s, a) \approx Q(s, a, \theta)$.

Given the advantages of a linear approximation to overcome the tabular learning limitations, these models will not be able to generalize well when the model complexity and state spaces increase. Here comes the advantage of using non-linear approximations such as Deep Neural Network (DNN) to approximate the environment, giving the agent the power of Deep Learning (DL) to update its weights, where training can be customized [73]. The Deep Q-Network (DQN) algorithm has the advantage of merging the concepts of RL and DL [74]. Henceforth and after experimenting with the different linear approximation approaches for building our IFSP agent, including Temporal Difference TD(0) and TD(λ) [75], DQN outperforms the other approximation methods. Algorithm 1 provides a pseudo-code of our IFSP learning algorithm, which benefits from the advancement achieved in DQN.

DQN benefits from the DL power in the supervised learning paradigm of machine learning. This is made possible by introducing a replay buffer that performs mini-batch sampling and stores the weights in a target network. In the sequel, we go over our implementation of the DQN algorithm for building the IFSP agent.

Algorithm 1: IFSP Algorithm Using DQN

Build a Multi-Layer Perceptron as source model to calculate Q and randomly initialize its weights θ;
Build a target model for Q with weights θ^{-} which are a copy of θ;
Initialize replay buffer D to capacity G;
while *episode* X **do**
    Initialize a random state $s(t,u)$;
    Reset t;
    while $t < \mathcal{T}$ **do**
        /* following ϵ-greedy policy */
        if *Random Selection* **then**
            select $a(t+1)$ randomly from feasible actions;
        else
            $a(t+1) = max_a Q(s(t,u), a, \theta)$;
        end
        Update $k(t+1)$, observe $q(t+1)$;
        Calculate $\mathcal{C}(s(t,u), a(t+1)|q(t+1))$ using Equation 17;
        Update u to u^{+};
        Build $s(t+1, u^{+})$;
        Store $[s(t,u), a(t+1), \mathcal{C}(s(t,u), a(t+1)|q(t+1)), s(t+1,u^{+})]$ in D;
        Select random mini-batch transition $(s_i, a_i, \mathcal{C}_i, s_{i+1})$) of size Y from D;
        for *j in length(mini-batch)* **do**
            $y_i = \mathcal{C}_i + \gamma \min_{a'} Q(s_{i+1}, a^{'}, \theta^{-})$;
        end
        Update θ using gradient descent towards minimizing the loss: $(y_i - Q(s_i, a_i, \theta))^2$ for every transition;
        if *length(D)* $> G$ **then**
            Pop out the oldest element in D;
        end
        Every Z steps, copy θ into θ^{-};
        Update the current state to $s(t+1, u^{+})$;
        Increment t;
    end
    Increment Episode;
end

As illustrated in Algorithm 1, we start by creating a multi-layer perceptron for the source model used for calculating the state action-value function Q using its weights θ. The input to the model is a transition sample, and the output is a single neuron with linear activation. A target multi-layer perceptron is created, which is a copy of the source model. We denote by θ^- the weights of the target model, which are a copy of θ in the initialization phase (line 1). We then initialize a replay buffer D of size $G = 1000$ which stores the transition containing the current state, the action taken, the cost retrieved, and the next state observed (line 2).

The learning starts by initializing a random state $s(t)$ at the beginning of every episode (line 5). X episodes are played for learning. X varies depending on the input size for the test case. Each episode is bounded by $\mathcal{T}$ learning steps. Every step starts by deciding on the action taken for the current state. We implement this decision by following the ϵ-greedy policy, which is essential for achieving a trade-off between exploration and exploitation. In ϵ-greedy, we set ϵ to be a variable that decays over time. For instance, $\epsilon = \frac{B1}{B2+Numberofiteration}$ decreases as the number of iteration increases, where $B1$ and $B2$ are two constants such that $B1 < B2$. We then generate a random value of w between zero and one. If $1 - \epsilon > w$, we select an action randomly from the action space (lines 8-9). This is known as an exploration iteration for the agent. Otherwise, the action having the maximum Q-value in the source model is selected (lines 10-11). This is known as the exploitation iteration.

After taking the action, the agent observes the service demands after the service placement is updated. This then allows the agent to calculate the cost $\mathcal{C}(s(t,u), a(t+1)|q(t+1))$ using Equation 17. After forming the next state $s(t+1, u^+)$, a transition is stored in the replay buffer (lines 13-17). Because updating the model online as data come causes instability, data are stored in the replay buffer. Samples from these data, of size $Y = 50$, are extracted randomly and uniformly to form the mini-batch dataset for the model to train and

break the problem of correlation between sequences of actions (line 18). As mentioned previously, the source weights are stored in the target model. This is vital to improve the source model learning stability. The source model adjusts θ of Q-function by using the predicted Q-values of the target model as labels (lines 19-21). This, in turn, builds a supervised learning context with a fixed dataset and labels on which to train. In our implementation of our IFSP-based DQN, loss functions are inferred and calculated for every iteration using the mean squared error loss (line 22). This loss is back-propagated to the neurons using the gradient-descent towards minimizing the loss to get a better estimate of Q (line 22). To preserve the RL concept for allowing the model to keep on improving the Q-function as new data come, the replay buffer D keeps on updating slowly by removing the oldest transitions at every iteration when the buffer is full (lines 23-25). On the other hand, the weights for the target model θ^{-} keeps on updating after $Z = 500$ (line 26).

3.5 Experimental Study

In this section, we experiment with our proposed IFSP solution based on DQN by studying the following:

- The convergence behavior of IFSP for small, medium, and large scale clusters, where the objective is to minimize our cost function.
- The ability of the IFSP agent to adapt to changes in the environment, including unexpected changes in the users' demand patterns for requested services, and for unexpected changes in the cost function parameters.
- The ability of the IFSP bootstrapping technique on the cloud to avoid the high rate of exploration errors, make the learning faster, and scale for large inputs.
- The ability of IFSP to outperform existing heuristic-based approaches in (1) quality

of the decision and (2) execution time to take the decision.

Our data utilized throughout the experiments are extracted from the Google Cluster Trace 2011-2 dataset (GCT) [76] and Nasa Server Logs (NSL) [77]. GCT provides real-life deployment scenarios of services on available servers. Thus, it provides a set of hosts with available resources and a set of services having resources requirements. In order to measure changing demands of requested services in real scenarios, we can utilize any logs present on any server, which point to the source IP of the user, the service requested, and the timestamp. These fields are enough for our IFSP agent to conduct the bootstrapping on the cloud and update itself when running on the orchestrator. In NSL, we considered source IP having the same subnet mask as a single geographical entity requesting services. Requesting a specific endpoint on the NASA web server is considered as calling a single service. Thus, for each service extracted from GCT, we assign a list of changing demands aggregated during a specified period of one hour.

Our simulation was performed on a Core i7-8700 (12 CPUs), 32GB RAM, and a Graphic card for GPU computation of type NVIDIA Quadro P620. We implemented Algorithm 1 using Python and the Tensorflow library. Using Tensorflow, we built the source and target networks. Our networks have four layers of neurons with 32, 16, 8, and 1 neurons respectively. The activation function used on each layer is ReLU, except for the output layer, where linear activation is used to predict the Q-Value. The input of the source and target neural networks is a combination of the state and action taken at that state. The first layer has an input size of $n + (m \times n) + 2$, where n is the number of services, and m is the number of hosts. We also use the RMS optimizer with a learning rate of 0.001. Furthermore, we compare our approach with two heuristic-based solutions and a DRL service placement solution. These solutions are implemented from scratch based on the objectives specified by each paper. In some of our experiments, we rely on evaluating the cost function which is directly related to the QoS experienced by the users. Furthermore, for each experiment,

$\mathcal{F}$	$\mathcal{F}_{cpu}$	$\mathcal{F}_{mem}$	$\mathcal{F}_{disk}$	$\mathcal{F}_{d}$
F_1	0.5	0.24	0.4	500
F_2	0.25	0.4	0.4	50
F_3	1.0	1.0	1.0	20

Table 3.1: Fogs Configurations for Scenario 1 (s1)

we run a different number of iterations depending on the objectives. For instance, we run 10^3 iterations to study the convergence and stability of IFSP. Meanwhile, 30 iterations are enough to illustrate the bootstrapper advantage.

3.5.1 IFSP Convergence and Scalability

In order to study the performance of IFSP following our MDP design, we simulate two different sizes of clusters with a different number of fogs and services having varying demands. Similar sizes are expected to be used in real-life settings. In Scenario 1 (s1), we use 3 fogs and 6 containers which are shown in Tables 3.1 and 3.2 and extracted from the GCT dataset. The purpose of Tables 3.1 and 3.2 is to show a snapshot of the data we have. For Scenario 2 (s2), we also use the GCT dataset to simulate a larger cluster composed of 15 fogs and 40 containers for validating the scalability of our MDP design. For both scenarios, demands are assigned to each service randomly from the NSL dataset. To compare our solution with the optimal decision, we utilize a greedy search for small inputs and pass it the demands after their occurrence. This greedy search generates all possible solutions for a given input, and yields the best placement for it, considering the same weights and cost function.

We define the weights for the current simulation in both scenarios as $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 0.25$. Fig. 3.3 illustrates the convergence results towards minimizing the cost function while the number of iterations increases. The experiment was executed for 10^3 iterations in order to illustrate the convergence and stability of IFSP. The results shown in this figure

$\mathcal{P}$	$\mathcal{P}_{cpu}$	$\mathcal{P}_{mem}$	$\mathcal{P}_{disk}$	$\mathcal{P}_k$
P_1	0.12	0.2	0.2	0
P_2	0.24	0.23	0.1	1
P_3	0.18	0.2	0.32	0
P_4	0.25	0.42	0.2	1
P_5	0.31	0.15	0.24	1
P_6	0.375	0.175	0.08	0

Table 3.2: Containers Configurations for Scenario 1 (s1)

are an average of 50 iterations for every observation. We can observe in (s1) that IFSP is capable of converging to make optimal decisions by approaching the optimal line. The optimal line is extracted using the greedy search method applied at each iteration for every encountered load. In (s2), the the IFSP agent also converges, validating its ability to handle large inputs. Due to the large input and high paste of changing demands, using greedy search to obtain the optimal decision is impossible. Therefore, we elaborate later in this section on the optimality of the decisions taken by comparing with a heuristic approach. We can also observe from the results that (s1) achieves faster convergence than (s2) due to the larger input experienced by the agent in (s2). Furthermore, it is important to mention that because all objective costs are given equal weights, the agent is expected to take longer because of a more complicated policy required to learn. In case one or two objectives are given a weight of zero, the convergence will be faster due to less complex policies to learn, which we demonstrate in the next subsection.

3.5.2 IFSP Adapting to Environment Changes

In order to elaborate further on the ability of IFSP to adapt to changing demands, we simulate Scenario 3 (s3) containing 8 fogs and 20 containers. We aim to study the combined and normalized demands of users that are not met after doing the IFSP placement for all services. In (s3), we aim to change the pattern of demands four times intentionally for

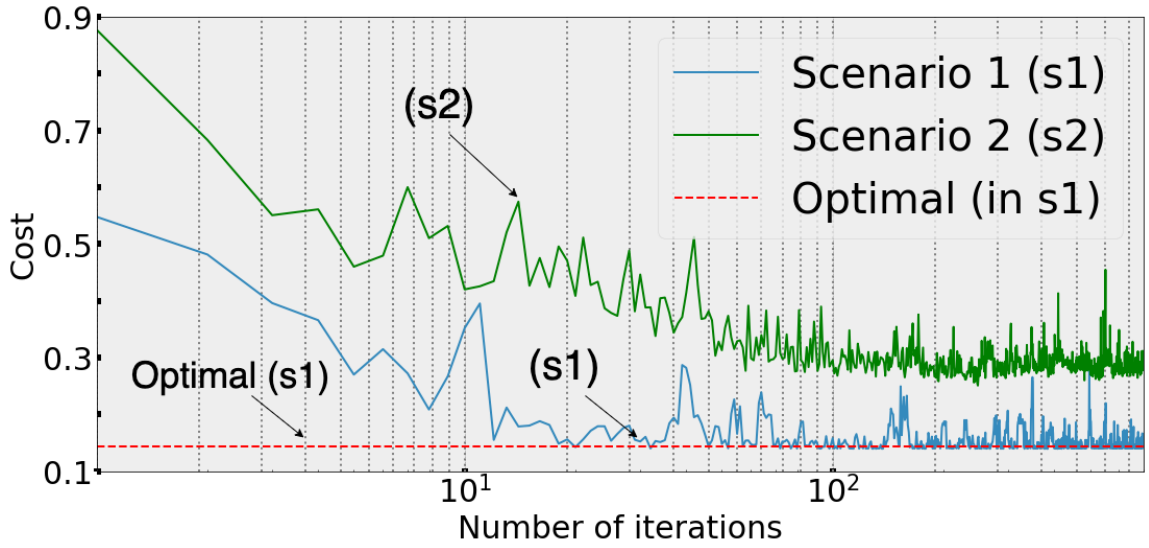


Figure 3.3: Convergence Performance of IFSP for Small and Large Clusters

each container to simulate how IFSP reacts to such a situation. The change in demands is provoked every 40 iterations, leading to 160 iterations during the evaluation. The results are shown in Fig. 3.4. As shown in the figure, IFSP is able to always converge into learning a pattern for the new incoming demands. We can see four jumps in the numerical cost for the four changes done to the demands because of the new states encountered. We are also able to notice through this simulation that the convergence incurred after the second change in demands is slightly faster than the starting stages of learning. This reduction in convergence speed is due to the tuned model that we have from the first cycle of demands. As a conclusion from (s3), whenever the pattern of demands for a service changes over time, IFSP adapts to this change by learning new patterns and meeting new demands when possible. In some cases, the change in demands makes placing services impractical, and therefore IFSP refrains from placing it. This is illustrated in the first and fourth cycles of learning where the error is larger compared to the second and third cycles.

The non-served requests studied in this experiment are served by the cloud. A decrease in this amount implies that more requests are served by the fog, therefore the user experiences a lower response time and a better QoS. Therefore, we show through this experiment that

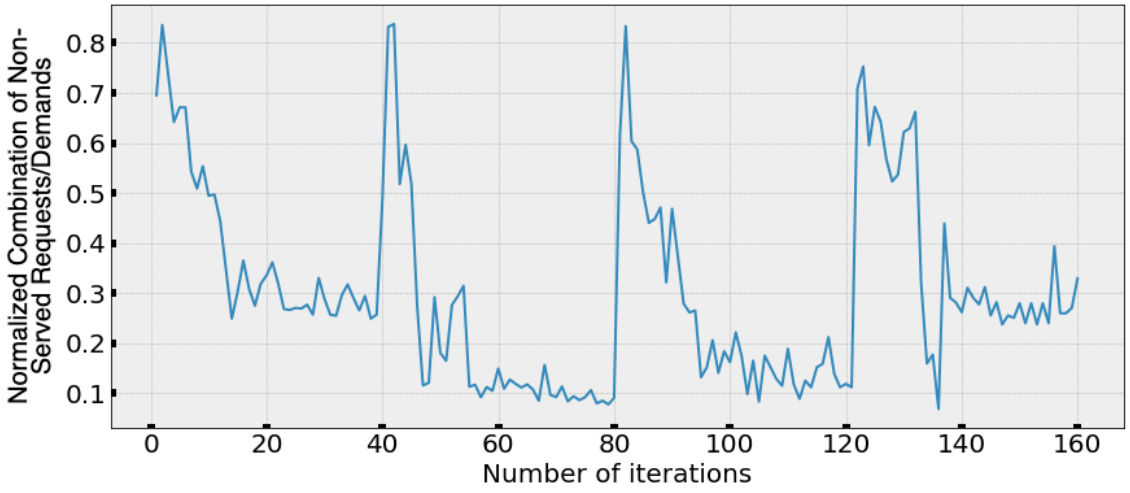
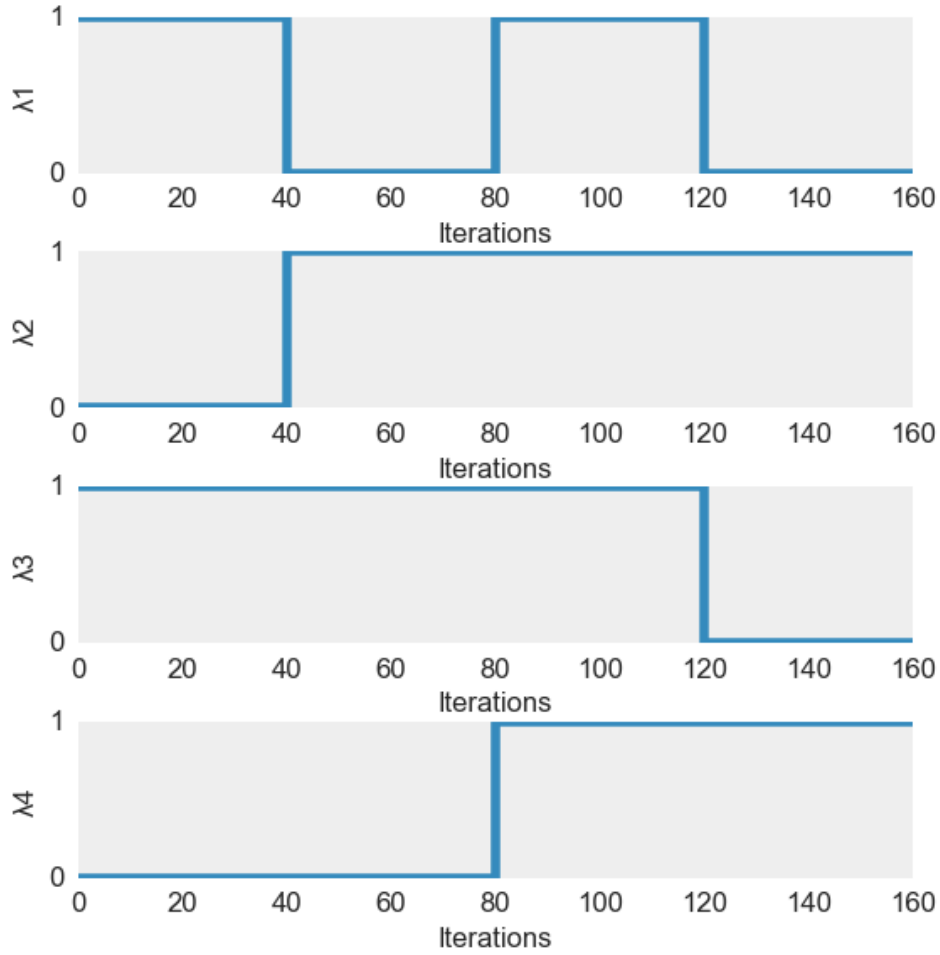


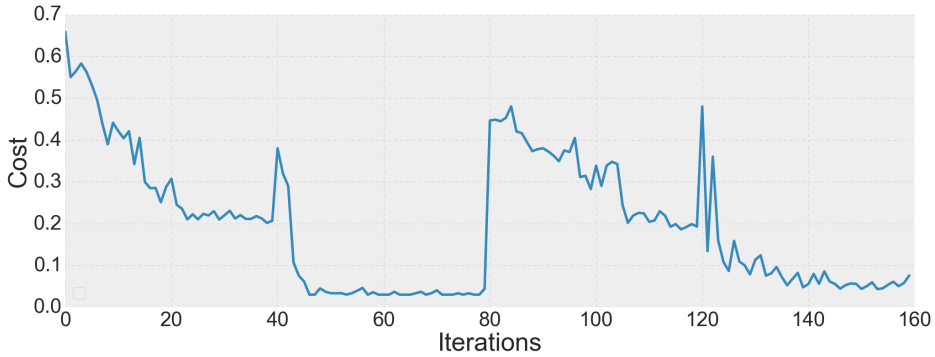
Figure 3.4: IFSP Performance Evaluation while Changing Demands

IFSP is able to improve the QoS experienced by the users even-though new patterns of demand are encountered.

As mentioned in our cost function formulation, four weights are assigned to each objective cost, which can be adjusted based on the service provider's assessment of the environment's needs. Such changes in the cost function introduce changes to the cost calculated by the agent. In order to verify the robustness of the agent in such a situation, we simulate Scenario 4 (s4), which confirms the intelligence of IFSP for adapting to the changes in the cost function. In (s4), we copy the cluster configuration of (s3) and update the weights after hitting a predefined number of iterations by the agent. The strategy for updating the weights is illustrated in Fig. 3.5a. In this figure, a signal equal to one means that the weight is in use for the current cycle. If more than one weight is used, the total weight is divided equally among them. In Fig. 3.5b, we show the performance of IFSP while considering four consecutive changes in weights. Every 100 iterations in (s4) are averaged to obtain the results shown in the figure. The IFSP performance is measured using the normalized cost function. The cost function starts converging at the beginning until a change to the weights occurs. A change in the weights causes a peak in the cost, as shown in the results. After every change in weights, IFSP is able to converge again to optimal solutions.



(a) Weights Variation



(b) Model Convergence

Figure 3.5: IFSP Performance Evaluation while Changing Weights

3.5.3 Comparison with DRL and Heuristic Approaches

Following the large input in (s2), we study in this section the performance of our IFSP agent compared to existing heuristic-based solutions [1, 2]. Besides, we compare with a DRL solution for service placement at the edge [30] and show the importance of using the IFSP bootstrapper. The large input caused scalability issues by overloading the memory when generating the possible actions in [30], which is not the case when using IFSP. We also present the limitations of the existing heuristic solutions in terms of execution time when the input to the problem grows, thus requiring more iterations to try finding the near optimal solution. This in turn results in increasing the execution time to make the placement decision, henceforth delaying the update of services or ignoring them completely as demands can change more often. We are also able to highlight the importance of proactive placement, which is not possible when using a heuristic solution. In this context, we build Scenario 5 (s5), which is a copy of (s2) input. However, in (s5), the IFSP model has passed the bootstrapping on the cloud discussed in Section 3.2, hence the model is a continuation of the learning that happened in (s2). Knowing that heuristic approaches rely on randomness to generate solutions, the range of possibilities is considerable when the input size is large, making it hard for the algorithm to hit the optimal solution. For instance, in (s5), and following our MDP design, the agent has 825 possibilities of placement to be taken for each observed demand, which is an acceptable number for such a large input. If we consider the service placement formulation in [1] or in [2] for cloud/fog placement, the number of possibilities is the different binary combinations of a matrix of size 15×40. The implemented heuristic solution embeds a local search to minimize the chances of halting in local optimal, and speeds reaching a feasible solution. The same cost function formulated in Equation 17 is implemented for fitness evaluation in MA. We used 200 generations and 100 individuals per generation that evolves to find a feasible solution. More details about the implemented MA can be found in [1].

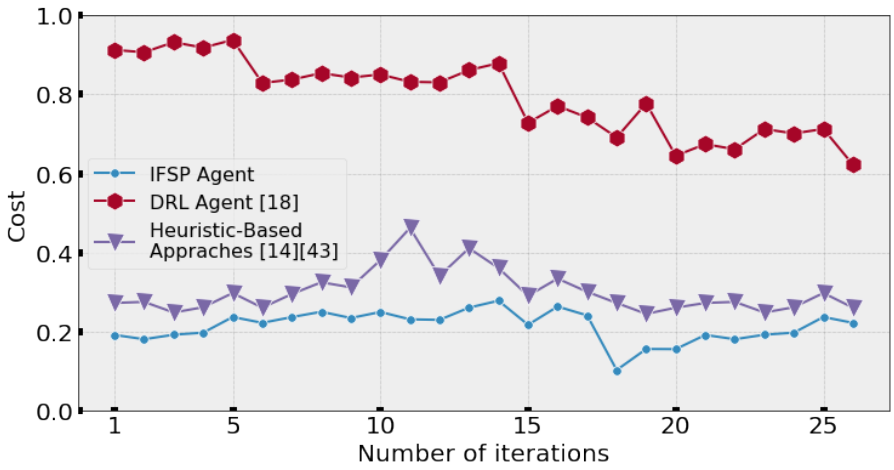


Figure 3.6: Our IFSP Agent Performance v.s. Heuristic-Based Approaches In Scenario 5 (s5)

After completing the bootstrapping phase in (s5), we utilize the ready model to make decisions, benefiting from the mature model. On the other hand, we run the DRL agent of [30] without bootstrapping, causing the agent to start learning from scratch on the current environment. Afterward, a snapshot of the decisions made by IFSP and the DRL agent of [30] is taken. The lists of fogs and services are passed to the heuristic solution to make the placement decisions. In the existing heuristic solutions [1][2], the change in demands for services is not studied. In order to measure the heuristic decision's performance, we pass the input to MA 10 times and record the average cost of the placement decisions. A snapshot of the decisions made by IFSP compared to heuristic-based approaches following (s5) is illustrated in Fig. 3.6. The evaluation is performed for 30 iterations, which are enough to show the importance of using a bootstrapper compared to a traditional DRL solution. Besides, each environment state is passed to heuristic for evaluation, which is time-consuming. This also explains the choice of 30 iterations for evaluation.

Following the performance of IFSP compared to DRL and heuristic-based decisions, we can observe that the normalized costs produced by IFSP are always less than or equal to those produced by both solutions. The IFSP and heuristic-based results are not only for the snapshot of the sample, but also always valid after IFSP converges. From Fig. 3.6, we can

observe the large difference in the cost results for all decisions made. In the case of DRL [30], the agent starts at the beginning by exploring the environment and taking random actions, which causes a high cost (i.e. low QoS) at the first stages of learning compared. In contrast, the IFSP solution has a pretrained model using the bootstrapper. This comparison highlights the importance of using a bootstrapper to overcome existing DRL limitations. Due to the limited capabilities of the heuristic solutions, proactive placement of services is not possible. Because we are executing the heuristic algorithm periodically, the placement of the current timestamp is outdated because it does not meet the actual demand. Moreover, due to the large input size, the execution time of the heuristic solution increases exponentially as illustrated in the next experiments. In this context, initializing the environment and migrating the containers consume more time, rendering the heuristic algorithms infeasible in time-sensitive applications. Noting that the results of Fig. 7 do not consider the time to setup the environment based on the new placement decision. On the other hand, IFSP is capable of taking decisions on the fly with a negligible processing time.

In heuristic-based algorithms, the processing time increases due to the increase in the input size. Every Pareto set, or list of best solutions is generated every time in heuristic by looping for specific number of generations and manipulating a set of individuals. The higher the numbers of generations and individuals are, the more likely the heuristic algorithm will generate better solutions. In (s5), because the input is large, we varied the numbers of generations/individuals and studied the run time for making the placement decision. The results of this experiment are revealed in Fig. 3.7. The execution time of the heuristic solution increases exponentially as the numbers of Generation/Individual increase. Because our IFSP solution is based on DRL, the execution time to get the placement decision is negligible because the agent takes a forward pass on the deep network for each action to select the best.

In order to elaborate further on the execution time drawback of using the heursitic

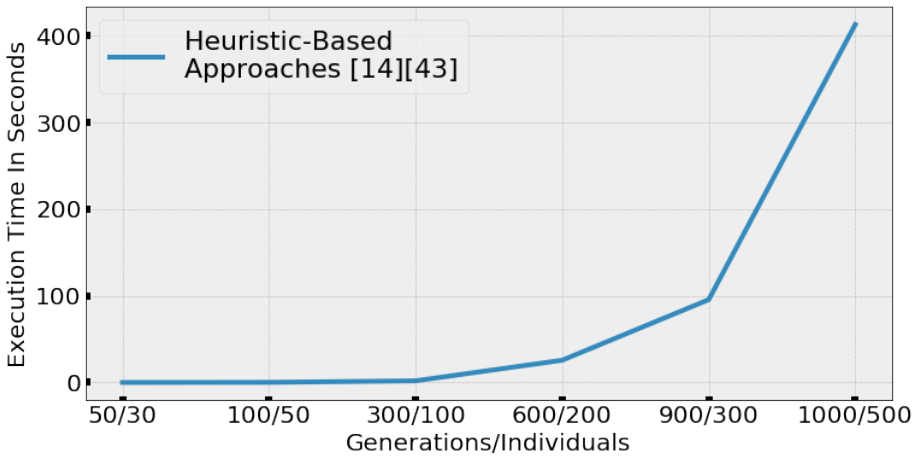
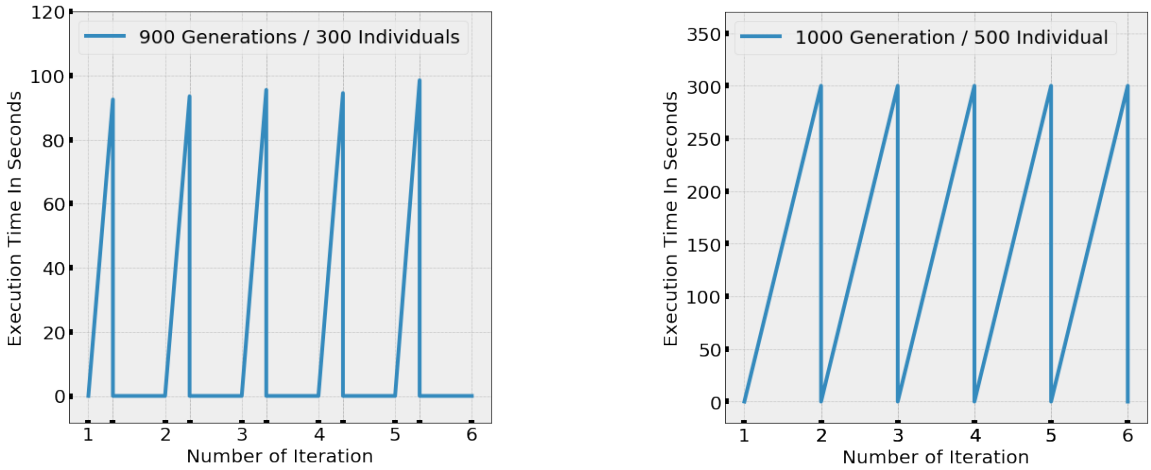


Figure 3.7: Execution Time While Changing The Number of Generations and Individuals In Scenario 5 (s5)

solutions to solve our problem, we consider applying the heuristics to our placement environment by feeding it the changing demands in (s5). The time between one iteration and another when the demands change is 5 minutes. The purpose of the experiment is to show the time needed for heuristic-based solutions to update services in the environment. We also varied the numbers of generations/individuals for each trial. The results are shown in Fig. 3.8a for 900/300 and in Fig. 3.8b for 1000/500.

As shown in Fig. 3.8a, the heuristic solution takes around 1.5 minutes to generate every solution. This is shown in every peak in execution time at the beginning of the iteration. These peaks are the time taken to update services, whereas our IFSP agent updates the services proactively before demands occur. Increasing the number of generations/individuals to 1000/500, we observe in Fig. 3.8b that the execution time of the heuristic solution is not terminating at every iteration (never updating the services), because the time it takes to produce the solution is more than 5 minutes (the duration of observing new demands).

Therefore, heuristic-based solutions are not suitable for time-sensitive placement problems and can be replaced by our solution. IFSP is capable of producing more efficient results in minimal execution time. We also benefit from the IFSP Bootsrapper running on



(a) Execution Time With 900 Generations and 300 Individuals

(b) Execution Time With 1000 Generations and 500 Individuals

Figure 3.8: The execution time of heuristic-based approaches [1, 2] using generations/individuals of 900/300 v.s. 1000/500 and changing the demands every 5 minutes for each iteration in scenario (s5)

the cloud to prepare the orchestrator. Furthermore, because IFSP is (1) scalable, (2) capable of adapting to changes in the environment, and (3) making proactive decisions before demands occur, it can completely replace the state-of-the-art heuristic solutions.

3.6 Conclusion

Fog and service placement is a challenging problem in demands-driven context entailing the need for effective decisions while adequately adapting to environmental changes. The use of heuristic solutions to perform the placement is not feasible due to the changing demands and the possibility of heuristics to diverge from optimal solutions. Empowered by the breakthroughs in the AI field, we exploited in this chapter the use of DRL as an intelligent solution for fog selection and container placement. Despite the errors made by the agent at the exploration stage and the long time required to learn, we are able to build an IFSP agent based on DQN capable of making efficient decisions in no time. This is possible by incorporating an intelligent IFSS scheduler and a bootstrapper for preparing the

IFSP model before being used. We then formulated an MDP design used for developing the IFSP agent based on the DQN algorithm. Our MDP formulation allows the agent to take proactive decisions, to study the change in user demands, and to consider fulfilling multiple objectives for serving the fog computing context. Through experimental studies, we used real-life datasets and demonstrated our IFSP agent's ability to generate efficient solutions for small and large cluster sizes. We were also able to validate the ability of IFSP to adapt to changes in the environment, including demand changes and preferences adjustments for calculating the cost function. In addition to these advancements, we were able to exhibit the power of our intelligent solution for generating better solutions compared to the state of the art heuristic solutions in large scale clusters.

Chapter 4

AI-based Resource Provisioning of IoE Services in 6G: A Deep Reinforcement Learning Approach

Currently, researchers have motivated a vision of 6G for empowering the new generation of the Internet of Everything (IoE) services that are not supported by 5G. In the context of 6G, more computing resources are required, a problem that is dealt with by Mobile Edge Computing (MEC). However, due to the dynamic change of service demands from various locations, the limitation of available computing resources of MEC, and the increase in the number and complexity of IoE services, intelligent resource provisioning for multiple applications is vital. To address this challenging issue, we present in this chapter IScaler, a novel intelligent and proactive IoE resource scaling and service placement solution. IScaler is tailored for MEC and benefits from the new advancements in Deep Reinforcement Learning (DRL). Multiple requirements are considered in the design of IScaler's Markov Decision Process. These requirements include the prediction of the resource usage of scaled applications, the prediction of available resources by hosting servers, performing combined

horizontal and vertical scaling, as well as making service placement decisions. The use of DRL to solve this problem raises several challenges that prevent the realization of IScaler's full potential, including exploration errors and long learning time. These challenges are tackled by devising an architecture that embeds an Intelligent Scaling and Placement module (ISP). ISP utilizes IScaler and an optimizer based on heuristics as a bootstrapper and backup. Finally, we use the Google Cluster Usage Trace dataset to perform real-life simulations and illustrate the effectiveness of IScaler's multi-application autonomous resource provisioning.

4.1 What is Resource Scaling?

Resource scaling is necessary for dynamic resource management of clustered computing environments. In the literature, resource scaling methods can either be vertical or horizontal for one micro-service. Horizontal scaling scales an instance in and out. Scale-out means creating and placing copies of a micro-service in the cluster, while scale-in means removing placed instances. Horizontal scaling requires a service placement decision, which assigns or remove services to and from the servers based on preferred objectives. Besides, vertical scaling is composed of Scale-up and Scale-down methods, which adjust the CPU and Memory for a micro-service. Scaling in and down are very important for improving energy and resource consumption and offering more resources to be utilized by other applications running in the cluster. A better visualization of resource scaling and service placement is depicted in Fig. 4.1.

In this work, we develop a multi-application scaling and placement solution, which can be integrated into any service-based clustering environment running containers and orchestration technologies, such as MEC. In reality, resource scaling is accompanied by many

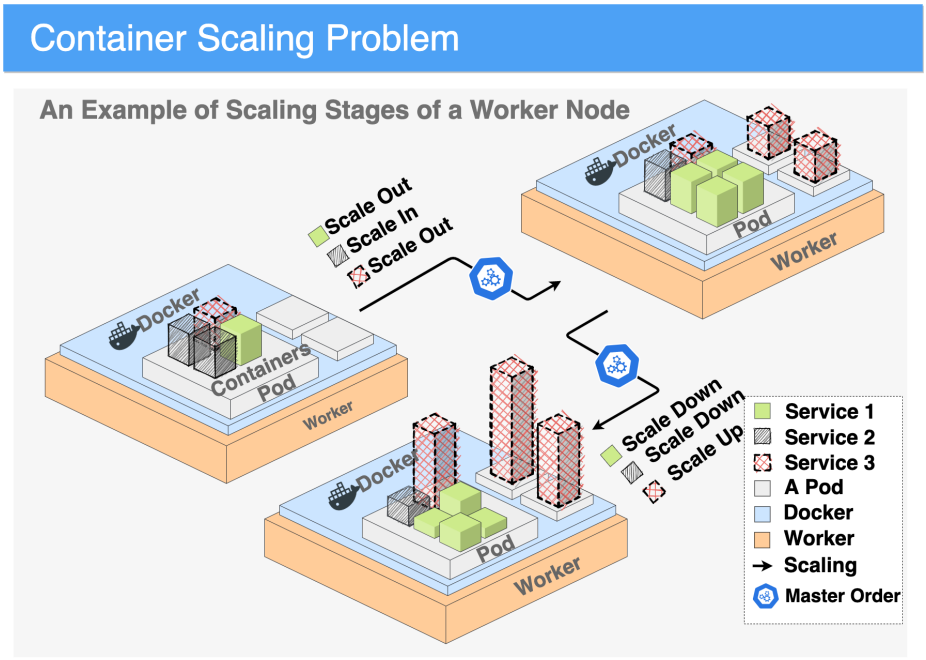


Figure 4.1: Visual Representation of the Horizontal and Vertical Resource Scaling Problem

requirements and sub-problems, which render the decision challenging. For instance, multiple applications could run on the same cluster; therefore, multi-application resource horizontal and vertical scaling is essential to find a balance between the different applications. Besides, in case scaling is not proactive, the QoS and QoE are affected. For instance, in case of scaling is performed as demands occur, the user might encounter a service delay, which can be measured from the starting time of demands to the time when scaling is performed. Hence, proactive scaling of resources is mandatory. Knowing that available resources of worker nodes can change, a prediction of availability is necessary for proactive scaling. Aside from executing the scaling decision, the target hosting worker should be identified to remove or add instances. This is known as the service placement problem, where decisions are made based on many objectives that can be configured depending on the cluster's situation. In other words, the cluster might be situated at the edge; therefore, placing a service closer to the user is required.

4.2 Architecture for Resource Provisioning in MEC Clusters

In this section, we present our architecture for integrating the IScaler technology in MEC clusters serving a 6G environment. This includes container-based clustering architectures managed using an orchestration technology. In this architecture, we apply changes to existing master nodes at the orchestration layer. These changes include the novel ISP module, which is responsible for performing intelligent scaling using IScaler and avoiding the challenges of using DRL.

4.2.1 Architecture Overview

For simplicity, we assume in this work that Kubernetes clusters are used for scaling, which is dedicated to managing Docker containers. Kubernetes also offers a suitable environment for managing and scaling resources, as well as load balancing the tasks on running instances. As shown in Fig. 4.2, this architecture covers the common cases of running an MEC cluster that contains orchestrators and worker nodes. In the MEC layer, the orchestrator performs scaling through IScaler, and the MEC servers host services and execute scaling decision. Finally, the user layer tha generates requests.

The cluster manager node runs the necessary Kubernetes components for cluster and connection management. The master is responsible for adding and removing worker nodes from the cluster. Moreover, installing, removing, and performing physical scaling are done through the master controller. A connection to all workers is checked for ensuring healthy running services. Failure to reaching services results in rebooting or migration to other workers. Besides, the master node receives information about the loads of each worker to perform optimal load balancing. The information is stored in log files, which are used for IScaler learning. Utilizing these standard functionalities of the master node, we propose

the integration of the ISP module described in Section 4.2.3. Besides, the communications that happen between the master and worker nodes at the MEC layer of the 6G environment go through the 6G network.

The worker nodes in our architecture are running at the MEC layer. The nodes can resemble any computing device, ranging from a mobile phone to a powerful server or a base station compute engine. These devices run the required Kubernetes components to be able to join the cluster and communicate with the master. Worker nodes receive orders from the master to host services. These services run in the form of containers hosted inside pods. A worker is also responsible for sending periodic updates about the current status and the time of availability to the master. More importantly, the worker nodes host services for supporting the requests coming from the user layer. These requests arrive with different levels of demands that change over time and have to be supported. These demands are reflected on a load of worker nodes. Hence, the master's job is to load balance these requests and use IScaler to scale the available resources proactively using AI.

In our architecture, users requesting edge services through the 6G network can be any computing device that initiates requests and can range from small IoT devices to powerful computing servers. Using IScaler and in case the edge servers have enough available resources, users are guaranteed highly available applications and a satisfactory QoE with a high response rate and negligible delay.

4.2.2 Architecture Components

Resource scaling does not tolerate mistakes or sub-optimal decisions that directly affect the hosted applications by causing downtime and disrupting the QoS and QoE. IScaler utilizes model-free DRL; therefore, the agent learns the environment from scratch through interaction and trial and error. Henceforth, by using DRL, IScaler is subject to producing wrong decisions at the starting stages of learning, or when unseen patterns or new states are

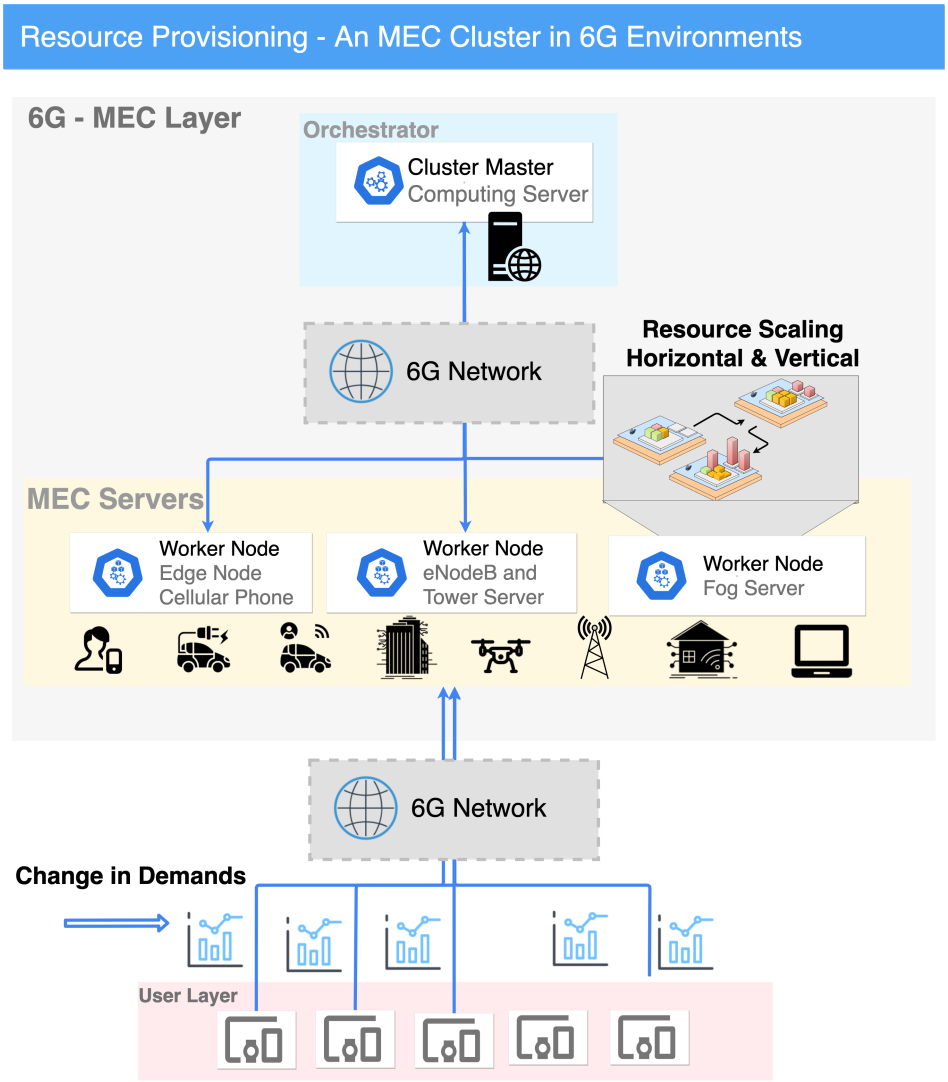


Figure 4.2: Resource Provisioning Architecture for IoE Services Hosted by an MEC Cluster in a 6G Environment

encountered and a model update is needed. To solve this issue, we propose in this section a novel architecture utilizing an optimizer combined with IScaler to cover possible errors during the learning phases. Also, our architecture offers a suitable environment for IScaler to learn using the collected application logs. A description of the orchestration layer's components is presented in Fig. 4.3 and described in the sequel.

4.2.3 Cloud Layer

The cloud layer is composed of (1) the actual application logic which gets offloaded to edge servers, and (2) the service management of connected clusters through orchestrators. The application logic is divided in two sublayers, the cloud native and containerization middleware. Additionally, the CaaS (Container as a Service) layer manages the underlying orchestrators of edge clusters. Initially, service requests generated from users are served by the application hosted on the cloud in a form of connected micro-services. These micro-services can be hosted in different clusters and locations, and they are launched and managed by the app server for correct distribution of tasks and collection of results. The role of the database in our architecture is to store the application logs which are exploited by IScaler for learning. IScaler, heuristics, and the solution switch are hosted by the cloud for learning. These three components are as well employed by the orchestrator for intelligent scaling and placement of services. More details about these scaling components is provided in the next subsection describing the orchestrator layer.

CaaS Module

The Container as a Service (CaaS) module presents the different Kubernetes components that should be running on the master node. The cluster orchestrator is the manager for its cluster. Workers' initialization, management, and configuration happen through the cluster orchestration entity. Moreover, this entity is responsible for updating the logs that represent the worker nodes' status and load. Besides, the cluster controller component is the direct connection with the worker node, which distributes, scales, and organizes services following the instructions of the cluster orchestration entity [6].

The Intelligent Scaling and Placement (ISP)

The intelligent scaling and placement sublayer is composed of IScaler, the Optimizer, and the Solution Switch. These components can be integrated into existing cluster orchestrators for performing resource scaling. IScaler is the DRL-based resource scaling solution that is responsible for proactively scaling computing resources and placing newly formed services on available servers. DRL solutions require time to learn by interacting with the environment in two cases: (1) learning the environment from scratch through trial and error, and (2) facing unseen patterns of services' demand or available resources in the studied data. These two factors cause the agent to make mistakes while performing the scaling decisions. To overcome this issue, we host a heuristic solution on the orchestrator as an alternate solution to replace IScaler while learning, and that confirms IScaler's correctness when making decisions on newly observed states or patterns. In other words, the Optimizer component is the bootstrapping tool for IScaler. The Solution Switch unit is utilized to check if IScaler's learning converges, to find the right time to switch between the Optimizer and IScaler solutions. This unit takes the output of IScaler and compares it with the one issued by the Optimizer. For simplicity, we use a threshold-based approach that counts and evaluates the correctness of the IScaler decision compared to heuristics. In case this count exceeds a predefined threshold, for instance, one hundred consecutive correct decisions, the orchestrator switches to using IScaler. It is also important to note that a heuristic solution cannot replace IScaler because: (1) a heuristic solution has to wait for demands to occur because it cannot take proactive decisions, and (2) heuristics cannot always guarantee a good solution.

Learning Data From Logs

Data utilized by the Optimizer and IScaler to learn and make decisions are provided by the Solution Switch module. This module keeps track of the current loads of each edge

server and monitors the demands of hosted micro-services at the edge. The service mesh component is utilized by the Optimizer and IScaler to respect the connection between microservices. Despite that these data are used for learning, they can be used by the Solution Switch to monitor IScaler's performance to take further actions.

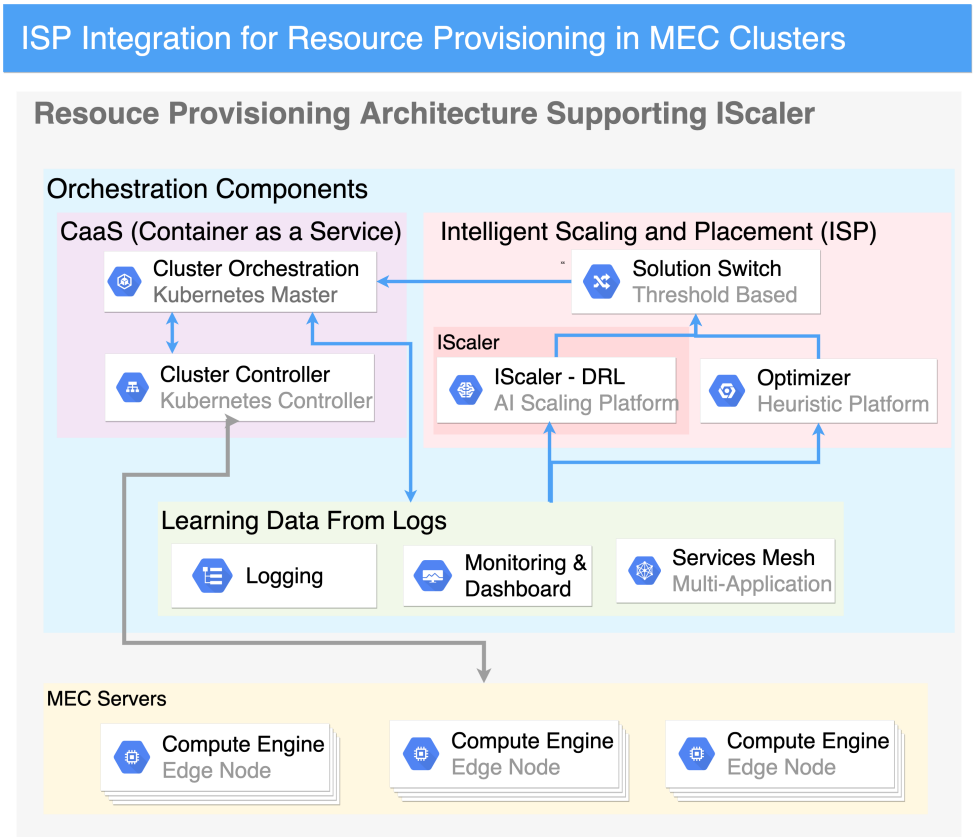


Figure 4.3: MEC Architecture Components Embedding ISP for Enabling IScaler

4.3 MDP Formulation for IScaler

IScaler agent utilizes a model-free DRL algorithm for learning. DRL takes as input a representation of the environment MDP model and tries to learn its dynamics or state transitions following some actions. In this section, we present a novel MDP formulation for performing proactive resource scaling and service placement while considering the change

in users' demand and available resources. Our MDP design guarantees scalability by handling large inputs. In other words, IScaler is still able to perform fast learning in case of a large input, in addition to consuming less memory while learning.

4.3.1 Background

An MDP is a mathematical formulation for modeling sequential decisions in a stochastic environment and is the main framework applied to problems solved using RL. An MDP is characterized by the following tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{C}, \mathcal{Y})$. This tuple is a design choice that can affect the RL solution scalability and speed of convergence. $\mathcal{S} = \{s_1, s_2, \dots\}$ is the state space, i.e., the set of all states of the environment. Problems can have a finite or infinite number of states. $\mathcal{A} = \{a_1, a_2, \dots, a_l\}$ is the action space, i.e., the set of possible actions the agent can take at any given state. $\mathcal{P}r$ is the probability transition matrix which outputs for each state s the probability distribution for going to the next state s' when performing action a. When $\mathcal{P}r$ is given, model-based RL is used. However, in most real-life applications, $\mathcal{P}r$ is not known. In this case, model-free RL techniques are used to estimate it. $\mathcal{C}$ is the cost function which reflects the objectives of the agent. $\mathcal{C}$ takes the current state, action, and next state, and outputs a value to be minimized. Finally, $\mathcal{Y}$ is the discount factor, which is a decimal number $\in [0, 1]$ and is usually close to one. The main use of $\mathcal{Y}$ is to speed the convergence by discounting over the reward of the next states. In order to build a DRL solution for IScaler, we need to define these elements of the MDP tuple. In the sequel, we provide a novel design of each element.

4.3.2 State and Action Spaces

IScaler is a multi-application scaling solution, where every application has a set of services. Hence, we denote by $\mathcal{G} = \{G_1, G_2, \dots, G_g\}$ the set of applications of size g, which are represented by services. In addition, We denote by $\mathcal{E} = \{E_1, E_2, \dots, E_n\}$ the set

of services of size n. A service $E_i \in \mathcal{E}$ is represented as: $E_i = [E_i^{cpu}, E_i^{mem}, E_i^{pri}, k]$, where $1 \leq i \leq n$ and E_i^{cpu}, E_i^{mem} are the CPU and memory requirements respectively. Moreover, E_i^{pri} is an integer representing the priority level over other services. When E_i^{pri} is high, E_i has a high priority to be considered for scaling and placement before other services with lower priority. Finally, k is the application index implying that $E_i \in G_k$. On the other hand, we denote $\mathcal{H} = \{H_1, H_2, \ldots, H_m\}$ the set of available hosts of size m that are running the services in $\mathcal{E}$. Every host H_j is represented as $H_j = [H_j^{cpu}, H_j^{mem}, H_j^{dis}]$, where $1 \leq j \leq m$ and $H_j^{cpu}, H_j^{mem}, H_j^{dis}$ are the CPU and memory available and the distance of this host from the group of requesting users respectively. As highlighted in Section 4.2, the hosting cluster can run at the MEC layer. In this case, there are specific requirements for hosting services. For instance, service placement should consider minimizing the hosts distances from the area of the users. Consequently, the host distance feature is considered later as one of the objectives in the MDP cost function described in Section 4.3.

In our state space, we represent the change in users demand and resource availability of each host in the cluster at different timestamp t. In each state, $q(t)$ represents the change in demand for different services, where $q(t)_i$ is a matrix of size $m \times 2$, which contains the average resource usage of CPU and memory of service E_i at t for every host in $\mathcal{H}$. The values in $q(t)$ are normalized to the total resource available on the hosts. Furthermore, we denote by $r(t)$ a matrix of size $m \times 2$ representing the normalized available resources of all hosts at t. $r(t)_j$ denotes the line j of $r(t)$ that represents the average resources for host H_j, i.e., $r(t)_j^{cpu}$ for CPU and $r(t)_j^{mem}$ for memory. Besides, available resources at a given state can be bounded by how much resources can service E_i use to scale. This boundary is set by a system administrator in order to leave space for other applications to scale in case the system is overloaded. To keep track of the latest scaling decisions, we denote by $p(t)$ the matrix of size $m \times n$ that stores these decisions taken at each host for each service. Each element $p(t)_{i,j}$ contains the CPU and memory allocations represented as $p(t)_{i,j}^{cpu}$ and

$p(t)^{mem}_{i,j}$ respectively. Thereafter, we represent a state s at t in our state space as follows:

$$s(t, i, j) = (q(t), r(t), p(t), i, j) \tag{22}$$

Following Equation 22, i and j are part of the state representation to denote the current service and host the agent is performing the decision for. The full $p(t), q(t), r(t)$ matrices are required in each state representation to make a combined decision while considering all services requirements and hosts availabilities. During t, the agent passes over all services and makes scaling decisions for each one separately. This reduces the action space and makes our MDP design scalable no matter the input size.

The action space in our MDP has a constant size which is very important for the model scalability. Every action a is composed of a list of two elements that hold the scaling decision of the CPU and memory for a given state. A scaling decision for CPU for instance is denoted as $a[0]$ and belongs to $\{-u, -1, 0, 1, u\}$. In this set, ± 1 denotes horizontal scaling, $\pm u$ is a decimal value that denotes vertical scaling, and 0 means no action is taken. It is important to emphasize that some actions are not feasible; however, if taken, the agent is punished using our cost function described in Section 4.3.4.

4.3.3 States Transition and Model Dynamics

IScaler state design presented in Section 4.3.2 relies on $q(t)$ and $r(t)$, which are the applications' resource requirements and the available resource at the next timestep t respectively. The values of these lists are unknown and are hard to predict. In other words, $q(t)$ and $r(t)$ have a stochastic behavior which is based on the demand change of the user for an application and the change in resource usage of hosting servers in the cluster. Because these values are unknown, $\mathcal{P}r$ of our MDP is unknown. Henceforth, the RL algorithm that should be used for these environments is model-free. On the other hand, the state design

entails the ability to perform scaling decisions for large clusters hosting several applications. To avoid blowing the action space, each state within a given timestep is divided into several steps.

Assuming that the current state is at t, the state representation is $(q(t), r(t), p(t), i^+, j^+)$. For instance, if the timestep is t, there are two loops of iterations defining the next states. The first loop considers fixing an application service and increasing j by one until passing over all the hosts and choosing the proper scaling action from $\mathcal{A}$. Once $j = m$, $j+$ becomes zero and i is increased by 1, which is denoted as i^+. Hence, $j^+ = j < m \ : j + 1 \ ? \ 0$. In addition, $i^+ = j = m \ : i + 1 \ ? \ i$, which means that $i+$ increases i by 1 in case $j = m$ and does not change i otherwise. Moreover, $p(t)$ at a state is updated by every scaling decision for the given i and j. It is important to note that for these internal iterations within a timestep, q and r are fixed until the agent moves to the next timestep. In this case, i, j, and $p(t)$ reset to zero, and a new q and r are observed by the agent.

4.3.4 Cost Function

Given the current state, the action taken, and the next state the agent results in, the cost function is calculated. When navigating in the state space, the goal of IScaler is to select the best action of the current state that results in the minimum cost. In other words, the correct selection of the action using the cost function C helps in forming an optimal policy for IScaler. In this section, we present four different objectives composing the cost function. These objectives are: (1) minimize the application load, (2) minimize the overload of the available resources, (3) minimize the containers priority cost, and (4) minimize the cost of other customizable objectives, such as minimizing the distance cost from the serving edge workers to actors. Taking an action moves the agent to the next timestep, i.e. from $t-1$ to t. A cost is represented as $C(s(t-1), a(t)|s(t))$. In the sequel, we present the mathematical formulation of each objective in the proposed cost function.

Minimize Application Load

The purpose of this objective function is to meet the load of different applications at the next timestep. Considering that the applications' loads are predicted, C_1 evaluates the scaling decision and compares the allocated resources to the ones required by each application. If the scaling decision underestimates the load, the cost returned is the difference between the actually required resources and the scaled ones. In case the demands are met for an application, the resource cost returned is zero. For this objective, we consider the cost of meeting the applications' resource requirements for both CPU and memory. Mathematically, C_1 of the CPU cost is represented in Equation 23.

$$C_1^{cpu}(t) = \frac{\sum_{i=1}^{n}(q(t)_i^{cpu} - \sum_{j=1}^{m} p(t)_{j,i}^{cpu} \times E_i^{cpu})}{\sum_{i=1}^{n} q(t)_i^{cpu}} \tag{23}$$

$$\text{such that } \forall i, \quad \sum_{j=1}^{m} p(t)_{j,i}^{cpu} \times E_i^{cpu} < q(t)_i^{cpu}$$

where $q(t)_i^{cpu}$ is the CPU usage of service i and E_i^{cpu} is its CPU requirement. Otherwise, if $\exists i \; s.t. \; q(t)_i^{cpu} \leq \sum_{j=1}^{m} p(t)_{j,i}^{cpu} \times E_i^{cpu}$, the cost of this service is zero because the resource requirements for the application are met. We also divide the cost by $\sum_{i=1}^{n} q(t)_i^{cpu}$ for normalization. It is important to note that Equation 23 refers to CPU calculation, which is the same for memory calculation; however, we use $q(t)_i^{mem}$ and E_i^{mem} instead of $q(t)_i^{cpu}$ and E_i^{cpu}. Finally, $C_1(t) = C_1^{cpu}(t) + C_1^{mem}(t)$.

Minimize Available Resources Overload

In this objective function, the agent is punished for exceeding the use of available resources using the proactive scaling decision made. We denote C_2 as the cost of this objective. C_2 represents the resource overload cost by each application for the CPU and memory on each host. The CPU cost for this objective is represented mathematically in Equation

24.

$$C_2^{cpu}(t) = \frac{\sum_{j=1}^{m}\sum_{i=1}^{n}(p(t)_{j,i}^{cpu} \times E_i^{cpu}) - q(t)_i^{cpu}}{\sum_{j=1}^{m} r_j^{cpu}} \tag{24}$$

$$\text{such that } \forall j, \ \sum_{i=1}^{n}(p(t)_{j,i}^{cpu} \times E_i^{cpu}) > q(t)_i^{cpu}$$

In case the sum of scaled resources on at least one host underestimates available resource (i.e. $\exists j, \ \sum_{i=1}^{n}(p(t)_{j,i}^{cpu} < 0$), the agent is punished for the action taken and a cost of k is returned for this objective, where $k > 1$. The same punishment applies when the scaling decision surpasses the available resources of any host (i.e. $\exists j, \ \sum_{i=1}^{n}(p(t)_{j,i}^{cpu} \times E_i^{cpu}) > r(t)_j^{cpu}$). Equation 24 applies if the usage surpasses the required resource load identified. Hence, the sum of the difference between the scaled resources of each application and the actual required resource load is returned. Otherwise, the cost is zero. In addition, A normalization factor of $\sum_{j=1}^{m} r_j^{cpu}$ is considered. Finally, Similar to C_1, $C_2(t) = C_2^{cpu}(t) + C_2^{mem}(t)$.

Priority Cost

A priority level is assigned to each service description. This value prioritizes the scaling of a service over others, which is important in the multi-application context and helps IScaler make efficient decisions. The cost of this objective is denoted as C_3. C_3^{cpu} is mathematically formulated in Equation 25 showing the CPU cost.

$$C_3^{cpu}(t) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(q(t)_i^{cpu} - p(t)_{j,i}^{cpu} \times E_i^{cpu}) \times E_i^{pri}}{\sum_{i=1}^{n} q(t)_i^{cpu} \times E_i^{cpu}} \tag{25}$$

$$\text{such that } \forall i, \ \sum_{j=1}^{m} p(t)_{j,i}^{cpu} \times E_i^{cpu} < q(t)_i^{cpu}$$

In case the resource loads of the application at the next timestep are met, or the service priority is zero, the cost of this service is zero. Otherwise, Equation 25 is applied to calculate

the remaining amount of resources needed to meet the resource load requirements of that service. Finally, $C_3(t) = C_3^{cpu}(t) + C_3^{mem}(t)$.

Minimize Distance Cost

As highlighted earlier, the infrastructure admin can add custom objectives to our IScaler cost function. Using this custom objective, IScaler can adapt and produce the desired scaling actions following specific preferences related to the hosting environment. Supposing that the cluster where IScaler is deployed is hosted at the edge, one of the possible objectives to consider is minimizing the distance between the edge worker and the group of requesting users. Therefore, we present in Equation 26 a mathematical representation of C_4 for minimizing the total distance cost.

$$C_4(t) = \frac{\sum_{j=1}^{m} v(t)_j \times H_j^{dis}}{\sum_{j=1}^{m} H_j^{dis}} \tag{26}$$

where H_j^{dis} is the distance cost of host H_j, and $v(t)$ is a vector of size m and is calculated as follows: $\forall j$, $v(t)_j = 1$ if $\sum_{i=1}^{n} p(t)_{i,j} > 0$ and 0 otherwise. A normalization factor of $\sum_{j=1}^{m} H_j^{dis}$ is added.

Therefore, our cost function becomes:

$$\begin{aligned} \mathcal{C}((s(t-1), a(t))|s(t)) = \lambda_1 \times C_1(t) + \lambda_2 \times C_2(t) + \\ \lambda_3 \times C_3(t) + \lambda_4 \times C_4(t) \end{aligned} \tag{27}$$

where $\lambda \in [0, 1]$ is a weight corresponding to each cost function given $\sum_{i=1}^{4} \lambda_i = 1$. These weights are tuned depending on the applications requirements and the nature of the cluster to give some cost functions more importance over the others, where the aim is to minimize $\mathcal{C}((s(t-1), a(t))|s(t))$.

4.4 Intelligent Scaling and Placement (ISP)

4.4.1 IScaler using Deep Reinforcement Learning

The IScaler agent interacts with the environment for evaluating the placement action taken for each container. The agent executes actions for every state encountered and builds a strategy that adapts to the stochastic demands of users requesting services, as well as the change in available resources on worker nodes. The end goal of the agent is to learn the transition probability distribution from a state to all next states and find the optimal policy π^*, which takes as input a state and outputs the action that minimizes the future cost. In other words, π^* is a strategy or a set of actions the agent takes to minimize the cost. The future costs are discounted by γ, which controls the effect of future actions on past and current states, and helps the agent achieve faster convergence. let $\mathbb{C}(s(t-1), \pi|s(t))$ be the future discounted cost implied by choosing policy π at t that indicates selecting an action $a(t^{'})$, such that $t \leq t^{'} \leq \mathcal{T}$ where $\mathcal{T}$ is the final timestep of the episodes. $\mathbb{C}(s(t-1), \pi|s(t))$ is computed as follows:

$$\mathbb{C}(s(t-1), \pi) = \sum_{t^{'}=t}^{\mathcal{T}} \gamma^{t^{'}-t} \mathcal{C}(s(t^{'}-1), a(t^{'})|s(t^{'})) \tag{28}$$

We denote by $Q^*(s, a)$ the optimal action value function which minimizes the average expected cost for any selected strategy. It is expressed as follows:

$$Q^*(s, a) = \min_{\pi} \mathbb{E}[\mathbb{C}(s(t-1), \pi)] \tag{29}$$

$$\text{where} \;\; s(t-1) = s, a(t) = a$$

Let $[s..\mathcal{L}]$ be the chain of states from s to $\mathcal{L}$ linked by transitions using $\mathcal{P}r$. The optimal Q-function selects the action of the next state that minimizes the action value function

following Equation 30:

$$Q^*(s,a) = \mathbb{E}_{s' \in [s..\mathcal{L}]}[\mathcal{C} + \gamma \min_{a'} Q(s', a')] \tag{30}$$

where $\mathcal{C}$ is the immediate cost from Equation 27, and $\mathbb{E}_{s' \in [s..\mathcal{L}]}$ is the expected value from the current state s to the last state $\mathcal{L}$ at $\mathcal{T}$. The basic form of RL is to find the optimal action value function using iterative updates following the Bellman equation. This update can be expressed as:

$$Q(s,a) := Q(s,a) + \alpha[\mathcal{C} + \gamma \min_{a'} Q(s', a')] \tag{31}$$

where α is the learning rate. In Equation 31, the update of the Q-function happens following the Q-learning algorithm [71]. All Q-values are stored in a table structure containing the list of states and actions. An exploration-exploitation trade-off aids the agent into interacting with the environment by covering the maximum number of possibilities, observing the cost signal, and updating the Q-values using Equation 31.

However, the use of tabular RL is not practical in our problem, where we have a large state space. The state-space can grow with an increase in the number of containers and hosts to place. Thus, handling the whole table in memory, trying to cover all possible actions for every state, and updating the Q-values for all of them is computationally very expensive. Such an implementation is time-consuming and makes any tabular RL agent diverges [72]. As a solution, learning the optimal Q-values can be retrieved from some adjustable weights denoted as θ. These weights get updated using gradient descent to update the weights downwards towards the direction of the gradient for minimizing the error of the calculated Q-values for every iteration. The common form of approximation is the linear function approximation, which generalizes the environment through its weight, where the Q-function becomes close to the optimal Q^* having $Q^*(s,a) \approx Q(s,a,\theta)$.

Given the advantages of a linear approximation to overcome the tabular learning limitations, these models will not be able to generalize well when the model complexity and state spaces increase. Here comes the advantage of using non-linear approximations such as Deep Neural Network (DNN) to approximate the environment, giving the agent the power of Deep Learning (DL) to update its weights, where learning can be customized [73]. The Deep Q-Network (DQN) algorithm has the advantage of merging the concepts of RL and DL. Henceforth and after experimenting with the different linear approximation approaches for building our IScaler agent, including Temporal Difference TD(0) and TD(λ) [75], DQN outperforms the other approximation methods. Algorithm 2 provides a pseudo-code of our IScaler learning algorithm, which benefits from the advancement in DQN.

As illustrated in Algorithm 2, we start by creating a multi-layer perceptron for the source model used for calculating the state action-value function Q using its weights θ. The input to the model is a transition sample, and the output is a single neuron with linear activation. A target multi-layer perceptron is created, which is a copy of the source model. We denote by θ^{-} the weights of the target model, which are a copy of θ in the initialization phase (line 1). We then initialize a replay buffer D of size $G = 1000$, which stores the transition containing the current state, the action taken, the cost retrieved, and the next state-observed (line 2).

The learning starts by initializing a random state $s(t)$ at the beginning of every episode (line 5). X episodes are played for learning. X varies depending on the input size for the test case. Each episode is bounded by $\mathcal{T}$ learning steps. Every step starts by deciding on the action taken for the current state. We implement this decision by following the ϵ-greedy policy, which is essential for achieving a trade-off between exploration and exploitation. In ϵ-greedy, we set ϵ to be a variable that decays over time. For instance, $\epsilon = \frac{B1}{B2+NI}$ decreases as the number of iterations NI increases, where $B1$ and $B2$ are two constants such that $B1 < B2$. We then generate a random value of w between zero and one. If

Algorithm 2: IScaler Algorithm Using DQN

Build a Multi-Layer Perceptron as source model to calculate Q and randomly initialize its weights θ;
Build a target model for Q with weights θ^{-} which are a copy of θ;
Initialize replay buffer D to capacity G;
while *episode* X **do**
    Initialize a random state $s(t)$;
    Reset t;
    while $t < \mathcal{T}$ **do**
        /* following ϵ-greedy policy */
        if *Random Selection* **then**
            select $a(t+1)$ randomly from feasible actions;
        else
            $a(t+1) = max_a Q(s(t), a, \theta)$;
        end
        Update $p(t+1)$, observe $q(t+1)$ and $r(t+1)$;
        Update j to j^{+}; // if applicable
        Update i to i^{+}; // if applicable
        Build $s(t+1)$;
        Calculate $\mathcal{C}(s(t), a(t+1)|s(t+1))$ using Equation 27;
        Store $[s(t); a(t+1); \mathcal{C}(s(t), a(t+1)|s(t+1)); s(t+1)]$ in D;
        Select random mini-batch transition of size Y from D;
        for *k in length(mini-batch)* **do**
            $y_k = \mathcal{C}_k + \gamma \min_{a'} Q(s_{k+1}, a^{'}, \theta^{-})$;
        end
        Update θ using gradient descent towards minimizing the loss: $(y - Q(s, a, \theta))^2$ for every transition;
        if *length(D)* $> G$ **then**
            Pop out the oldest element in D;
        end
        Every Z steps, copy θ into θ^{-};
        Update the current state to $s(t+1)$;
        Increment t;
    end
    Increment Episode;
end

$1 - \epsilon > w$, we select an action randomly from the action space (lines 8-9). This is known as an exploration iteration for the agent. Otherwise, the action having the maximum of Q-value in the source model is selected (lines 10-11). This is known as the exploitation iteration.

After taking the action, the agent observes the service demands and available resources after the service placement is updated. This then allows the agent to calculate the cost $\mathcal{C}(s(t), a(t+1)|s(t+1))$ using Equation 27. After forming the next state $s(t+1)$, a transition is stored in the replay buffer (lines 13-18). Because updating the model online as data comes causes instability, data are stored in the replay buffer. Samples from these data, of size $Y = 50$, are extracted randomly and uniformly to form the mini-batch dataset for the model to train and break the problem of correlation between sequences of actions (line 19). As mentioned previously, the source weights are stored in the target model. This is vital to improve the source model learning stability. The source model adjusts θ of Q-function by using the predicted Q-values of the target model as labels (lines 20-22). This, in turn, builds a supervised learning context with a fixed dataset and labels on which to train. In our implementation of IScaler-based DQN, loss functions are inferred and calculated for every iteration using the mean squared error loss (line 23). This loss is back-propagated to the neurons using the gradient-descent towards minimizing the loss to get a better estimate of Q (line 23). To preserve the RL concept for allowing the model to keep on improving the Q-function as new data come, the replay buffers D keeps on updating slowly by removing the oldest transitions at every iteration when the buffer is full (lines 24-26). On the other hand, the weights for the target model θ^{-} keeps on updating after $Z = 500$ (line 27).

4.4.2 Optimizer

An Evolutionary Memetic Algorithm (MA) is an extension of the genetic algorithm that includes a local search. The local search allows the heuristic solution to minimize the

chance of halting in a local optimal and offers faster convergence to a near-optimal solution. The MA is used in our previous research work for resource management [1], [15].

In the context of resource scaling and management, we utilize the Optimizer to support IScaler in producing efficient decisions. The implementation of the Optimizer is based on the MA solution. In terms of the formulation of the MA, the inputs to the algorithm are: the set of services $\mathcal{E}$, the set of hosts $\mathcal{H}$, the set of resource usage of each application q after the occurrence of the demands, and the set of currently available resources r. The output of the MA is a two-dimensional matrix of size $m \times n$ representing the scaling value of each container on every host. An element in the output matrix is a value in $[0, \mathcal{G}_{max}]$ where $\mathcal{G}_{max}$ denotes the maximum value a container can be scaled to. Moreover, obviously, our MA solution for the Optimizer uses the same cost function presented in Equation 27. This allows the Solution Switch to be able to compare the output of the Optimizer and IScaler. More details about the MA implementation can be found in [1].

4.5 Experiments and Evaluations

In this section, we describe the experimental setup and elaborate on the different experiments conducted to show the efficiency of the proposed IScaler in different contexts, the advantage of utilizing the Optimizer within ISP, and finally a comparison with a recent existing scaling solution. In brief, the objectives are:

- Study IScaler DRL model convergence in a multi-application context, in addition to studying the efficiency of the decisions made towards resource provisioning.
- Highlight the advantage of using our Optimizer on the orchestrator during the learning phase of IScaler.
- Compare the performance of IScaler to a model-based RL algorithm for edge computing environments [9].

4.5.1 Experiment Setup

To meet the objectives of the experiment, we implemented a DRL algorithm based on the proposed MDP design for building IScaler, a model-based RL algorithm called Dyna-Q [9] for a comparative study with IScaler, the MA for simulating the behavior of the Optimizer, and finally the Solution Switch. The experiments were executed on a Windows 10 machine having a Core-i7 (12 CPUs), 32GB of RAM, and an Nvidia Quadro P620 graphic card for GPU training. The programming language used is Python V3.7, and we relied on the Tensorflow library for the implementations of the source and target deep learning models of IScaler [78]. The source and target networks for the DRL implementation are deep neural networks that consist of four layers having 32, 16, 8, and 1 neuron, respectively. The activation function on the hidden layers is ReLU, while we are using a linear activation on the output. The networks are configured to use the RMS optimizer, the Huber loss, a learning rate of 0.001, and a batch size of 60. The source code for our MDP and DQN implementation is accessible on Github[1].

The conducted experiments are based on simulations on the Google Cluster Usage Traces v3 2019 dataset (GCT) [79]. In this dataset, google physical machines are used and grouped into cells (clusters) having different resources allocated and available. Moreover, jobs refer to users' requests to execute a certain task on the cluster. GCT provides the data describing each machine in the cell, the resources allocated and available, the jobs to be executed on each machine, and the required resources in terms of CPU and memory. In our work, we map the cell to a Kubernetes cluster where IScaler, the Optimizer, and the Solution Switch are running. Machines in the cells resemble the worker nodes of our clusters. Furthermore, the jobs correspond to the containers to be scheduled in the cluster and scaled using our solution.

[1] https://github.com/hanisami/IScaler-DRL

Data pre-processing, cleaning and visualization are performed on the dataset. For simplicity, three jobs/containers and three machines/worker nodes are selected from the dataset and visualized in Figs. 4.4 and 4.5. In Fig. 4.4, we show the average change in CPU demands by three different services over multiple samples from the dataset. In this figure, the curve drops imply a decrease in resource demands and therefore fewer requests from users arrive. In contrast, the curve reaches higher values when the demands of users increase. On the other hand, Fig. 4.5 illustrates the change of the average available resources for three different hosts sampled from the dataset. These figures demonstrate a real-life scenario for the change in demands and offered resources in the cluster. We benefit from this data in our experiment to show the ability of IScaler to adapt to these changes and perform efficient scaling and service placement decisions. For Simplicity, we display the CPU usage in the results instead of both CPU and Memory.

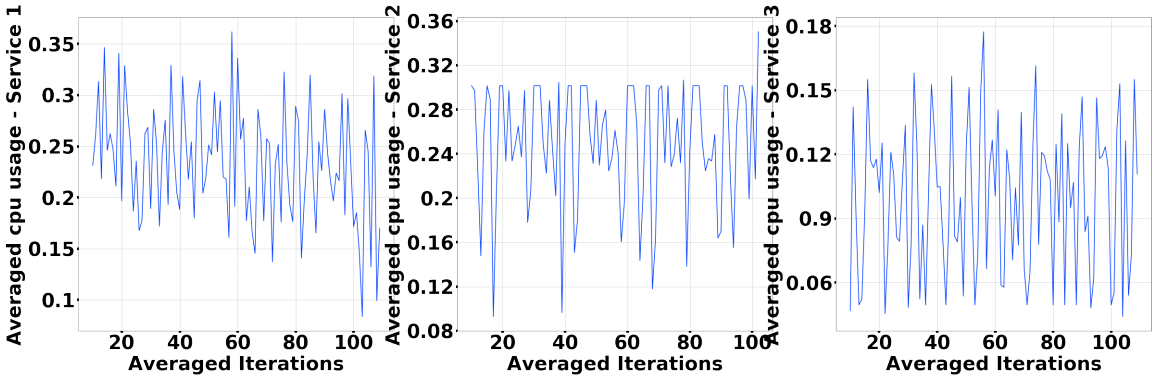


Figure 4.4: GCT Services Resource Demands

4.5.2 Multi-Application Model Convergence

In this part, we experiment with the performance of IScaler in a multi-application setting using the GCT dataset. In particular, we use the three samples of services and hosts described in Section 4.5.1. Services are presented by $\{E_1, E_2, E_3\}$, and hosts by $\{H_1, H_2, H_3\}$. Following the objectives of our cost function described in Section 4.3.4,

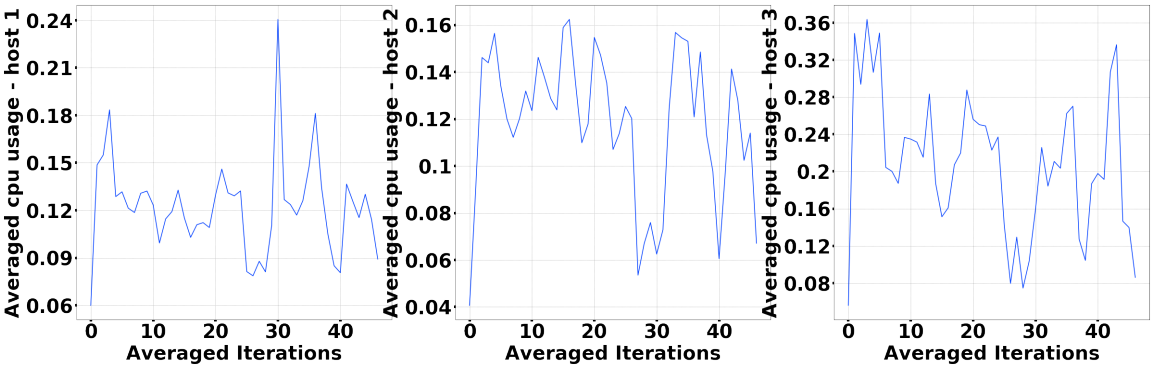


Figure 4.5: GCT Hosts Available Resources

we assign to each service a priority level, and for each host a distance value. The distance value represents the distance between the host location (longitude and latitude), and the central point between a group of users. The priority levels are assigned as follows: $E_1^{pri} = 1,\ E_2^{pri} = 3,\ E_3^{pri} = 2$. Moreover, the distance value assigned to each host are: $H_1^{dis} = 10\ H_2^{dis} = 20\ H_3^{dis} = 30$. Besides, we assign different weights to each objective of the cost described in Section 4.3.4. The weights assigned are: $\lambda_1 = 0.2\ , \lambda_2 = 0.4\ , \lambda_3 = 0.2\ , \lambda_4 = 0.2$. Consequently, the objective of minimizing the resource load of the application has more influence on the decision of the agent.

Following two million iterations of learning for IScaler using data of demands and resource availability fed from the GCT dataset, the model can converge with respect to the cost value produced for every decision. The long time of convergence is interpreted by the stochastic nature of demands and available resources on the GCT dataset as shown in Figs. 4.4 and 4.5. In Fig. 4.6, we show the convergence of our proposed DRL solution. In this figure, we plot the variation of the average cost value with respect to the average number of iterations, which are considered epochs. This graph is displayed on a logarithmic scale for better visualization of the agent performance.

In addition, we study the efficiency of decisions made during the learning and after the convergence with respect to every objective of our cost function. In Fig. 4.7, we show the amount of CPU resource load that is proactively prepared for each service after scaling.

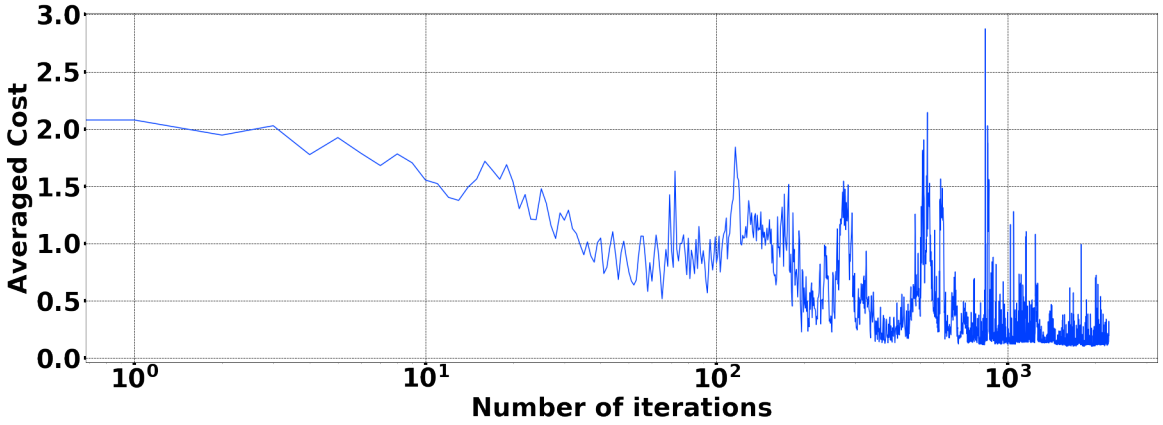
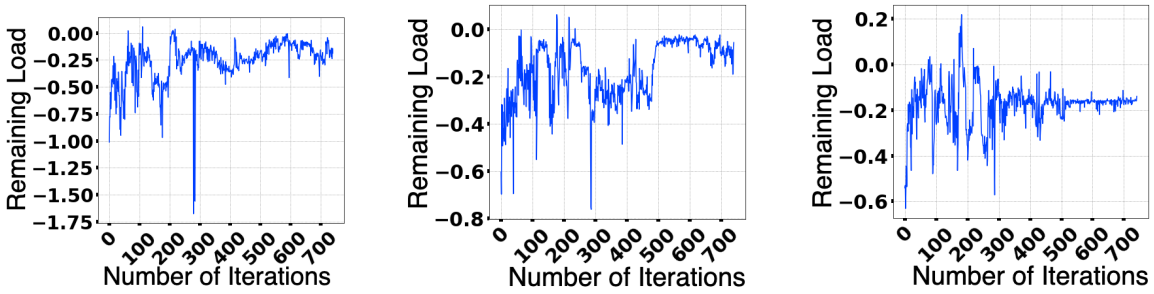


Figure 4.6: IScaler Convergence

In each graph, we plot the averaged difference between the actually required resource and the offered resources in the cluster with respect to the averaged number of iterations. This means that a closer value to zero means exactly the required demands are offered. On the other hand, a negative value indicates that the amount of offered resources exceeds the actual requirements of each application.



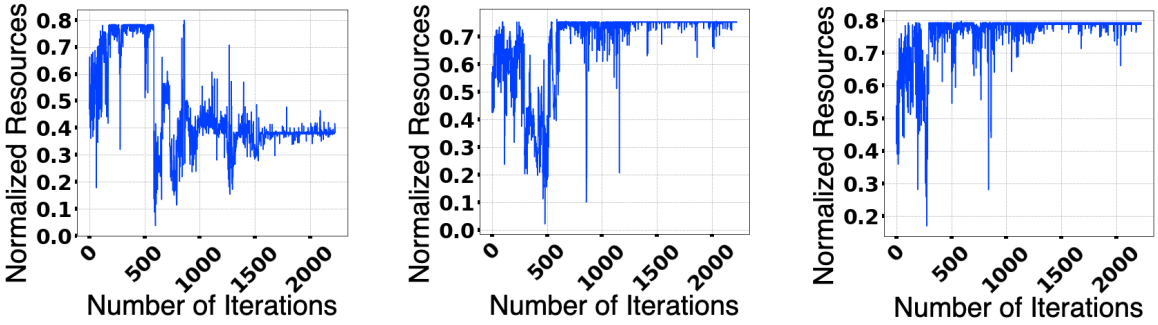
(a) E1 **(b)** E2 **(c)** E3

Figure 4.7: The Difference Between Actual Demands and Offered Resources for Each Service

As shown in the results of each figure, the amount of offered resources is most of the time larger than the required ones at the beginning of learning. This is because $\lambda_2 = 0.4 > \lambda_1 = 0.2$. Thus, meeting the amount of required resources has more impact compared to using available resources. Besides, the amount of utilized resources is approaching zero in each graph as the agent converges. In the end, the agent is able to learn the optimal resource allocation decisions for each service. More importantly, the resource of E_2 is exactly met

at each iteration due to the high priority score.

Available resources change over time, thus it is important to check if IScaler is utilizing more than the available resources on each host, which should be avoided. In Fig. 4.8, we show the averaged difference between the utilized resources by IScaler and the available resources on each host. A value of zero means that IScaler is proactively using the exact amount of available resource on this host, while a positive value indicates the amount of remaining available resources IScaler can use.



(a) H1 **(b)** H2 **(c)** H3

Figure 4.8: Remaining Available Resources of Each Host

As shown in Fig. 4.8, IScaler is able to converge while respecting the change of available resources on each host. It is important to note that the available resources on H_1 are more utilized compared to other hosts because H_1 has the shortest distance to the user. Moreover, as shown in the results of each figure, IScaler is not utilizing the full available resources on each host, therefore respecting the first objective in our cost function to minimize the amount of utilized resources.

In summary, IScaler is capable of performing efficient scaling decisions by meeting the load requirements for each application, more importantly, the ones with high priority, and respecting the amount of available resources for each host.

4.5.3 ISP Performance

As shown in the results of Figs. 4.6, 4.7, and 4.8, the agent is performing decisions that result in high costs on the environment. The high cost is reflected in shortening the applications' availability through unbalanced resource utilization and incorrect scaling of containers. This behavior of a DRL agent can occur in two cases. First, the agent is in the first stages of learning. Second, the agent is facing an unprecedented change in the environment that requires a model update. In both cases, the Optimizer can intervene to perform the scaling decision until IScaler develops a better model.

In order to experiment with the advantage of using the Optimizer, we simulate the behavior of combining the Solution Switch, the Optimizer, and the IScaler. While IScaler starts learning from scratch, we use the same setting and input of the previous experiment. The results of the averaged cost function using ISP with respect to an averaged number of iterations are shown in Fig. 4.9. As shown in this figure, the cost of the decisions made

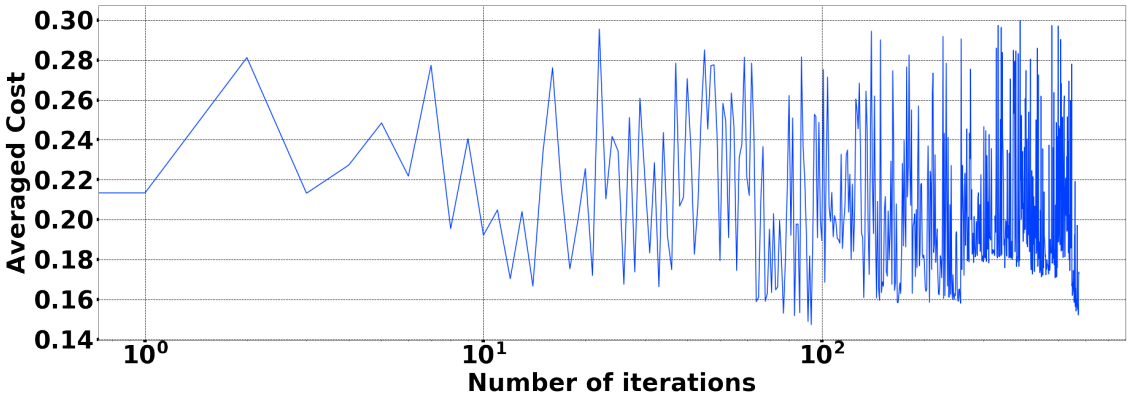
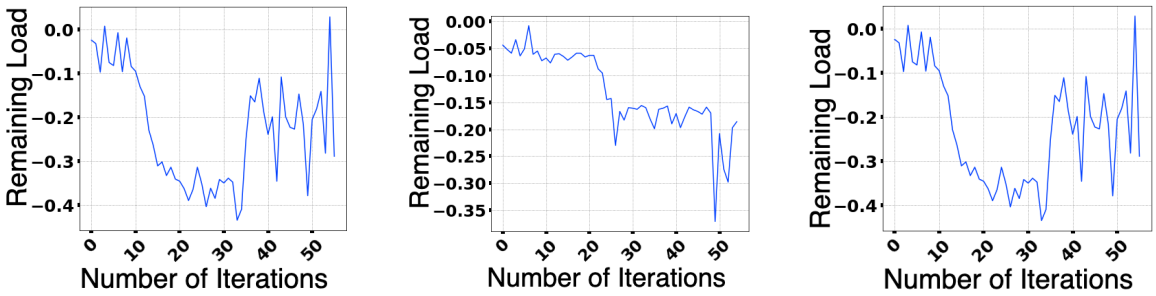


Figure 4.9: ISP Performance

is in the range between 0.14 and 0.3, including the first iterations when IScaler is making inaccurate decisions. This explains the importance of using the Optimizer for replacing IScaler. In this experiment, we queue the results of the decisions made by IScaler and the Optimizer. After every 100 iterations, the Solution Switch evaluates both decisions to decide on the right solution to use. After $300,000$ iterations, the Solution Switch silently

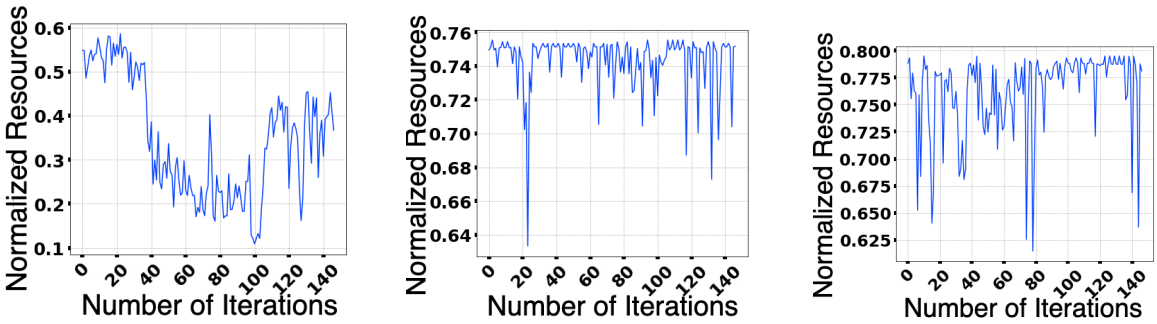
shifts from using the Optimizer to IScaler for proactive decisions. As shown in the graph, there are no jumps outside the range of $[0.14, 0.3]$ of cost because the model converges. One limitation remains when using the Optimizer, which is the inability of performing proactive decisions. Therefore, the scaling decision is made after the demands occur.

Furthermore, we study the impact of using the Optimizer on improving the quality of the decision to meet each of the objectives of our cost function described in Section 4.3.4. Therefore, Figs. 4.10 and 4.11 present the amount of resource load met for each application and the utilization of available resources on each host, respectively.



(a) E1 **(b)** E2 **(c)** E3

Figure 4.10: Resource Load for Each Service



(a) H1 **(b)** H2 **(c)** H3

Figure 4.11: Remaining Available Resources of Each Host

As shown in the results of Fig. 4.10, the required resources for each application are always met. Besides, the CPU resource load of each service has a negative value sometimes. This implies that services are assigned more resources compared to the needed ones. The

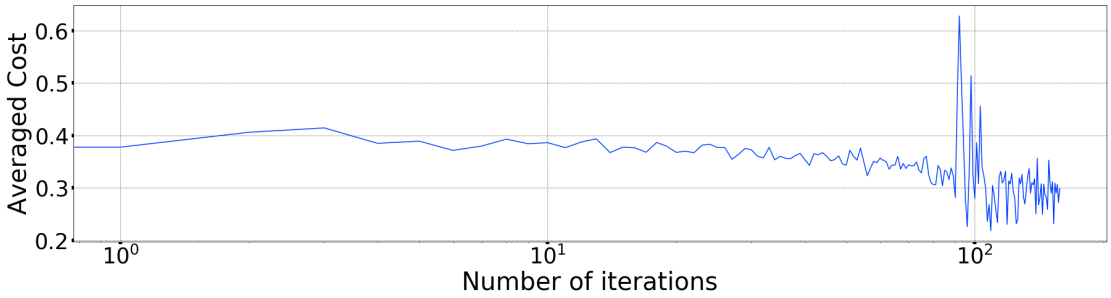
main reason behind this behavior is that meeting the resource load has more importance over minimizing the resources utilized on hosts ($\lambda_1 < \lambda_2$).

On the other hand, as shown in the results of Fig. 4.11, the available resources for H_2 and H_3 are less utilized by IScaler compared to H_1. The main reason is that the shortest distance from the users is H_1. Therefore, the IScaler decision respects the fourth objective of our MDP cost function to minimize the distance from users.

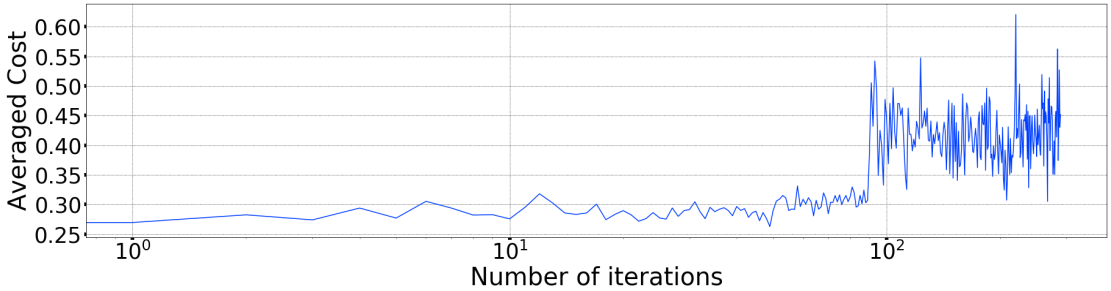
4.5.4 IScaler v.s. Model-Based Scaling

A recent literature work proposed a horizontal and vertical resource scaling for a single application using model-based reinforcement learning [9]. Despite that the service placement solution of that work is based on a heuristic solution, we compare in this section the performance of model-based RL to IScaler for a single application. Therefore, we replicate in this experiment the Dyna-Q model-based algorithm proposed in [9]. Some adjustments are applied to the state space of the Dyna-Q model to perform a fair comparison with IScaler. For instance, the features of service placement and the representation of the change of available resources are applied. Dyna-Q solution uses tabular Q-learning and estimates the probability transition matrix $\mathcal{P}r$ for learning the dynamics of the environment. Despite that estimating $\mathcal{P}r$ requires a lot of computation and sometimes is not practical in the case of large input spaces, the main objective of this experiment is to compare the behavior of Dyna-Q and IScaler when a change to the environment occurs. For this purpose, service E_1 is selected for scaling in both solutions on the three hosts. After multiple episodes of learning for IScaler, and one iteration for extracting $\mathcal{P}r$ for Dyna-Q, a major drop in demands is manually provoked in both environments. In order to compare the performance of each agent, the results of the averaged cost value are presented in Fig. 4.12b.

As shown in the results of Fig. 4.12b, the errors at the first stages of the decision making are negligible for the model-based Dyna-Q compared to IScaler performance in Fig. 4.12a.



(a) IScaler Performance - Model Free



(b) DynaQ Performance - Model Based

Figure 4.12: IScaler Performance vs Dyna-Q

This is obvious because the dynamics of the environment are known for Dyna-Q. However, once unprecedented change happens in the resource demands of the application, the errors for Dyna-Q jump, as shown in the first graph of Fig. 4.12b. This high error remains until $\mathcal{P}r$ of Dyna-Q is updated with new samples of data, making it impractical in such a dynamic environment. On the other hand, it is noticeable that IScaler is able to re-adjust the model, adapt to the environment change, and converge again in minimal time. IScaler uses model-free DRL for approximating $\mathcal{P}r$. This approximation dynamically changes with respect to changes in the environment that are interpreted through the reward signal.

4.6 Conclusion

Heading towards the development, hosting, and management of the new generation of services that are supported by 5G and 6G requires, there is a need for massive availability of computing resources, which is offered by the MEC. Due to the limitation of MEC

available resources, dynamic resource management of multiple applications on the MEC infrastructure has been identified as one of the main challenges for the future of cellular networks. Therefore, we propose in this chapter IScaler. IScaler is a DRL-based multi-applications resource scaling and service placement solution capable of overcoming the existing challenges of the dynamic environment with a stochastic change in demands to execute efficient decisions. Furthermore, adopting a DRL-based solution in 5G or 6G networks is very costly because of the errors the agent can make and the time required to learn. Thus, we propose an ISP module, which consists of IScaler, Optimizer, and Solution Switch. Through a series of experiments using the GCT dataset, we illustrated the efficiency of ISP decisions in (1) performing proactive intelligent multi-application scaling and placement decisions, (2) using the Optimizer during IScaler's model changes, and (3) demonstrating the ability of IScaler to outperform existing model-based scaling solutions.

Chapter 5

Graph Convolutional Recurrent Networks for Reward Shaping in Reinforcement Learning

In this chapter, we consider the problem of low-speed convergence in Reinforcement Learning (RL). As a solution, various potential-based reward shaping techniques were proposed to form the potential function. Learning a potential function is still challenging and comparable to building a value function from scratch. In this work, our main contribution is proposing a new scheme for reward shaping, which combines (1) the Graph Convolutional Recurrent Networks (GCRN), (2) augmented Krylov, and (3) Look-ahead advice to form the potential function. We devise an architecture for GCRN that combines Graph Convolutional Networks (GCN) to capture spatial dependencies and Bi-Directional Gated Recurrent Units (Bi-GRUs) to account for temporal dependencies. Our definition of the loss function of GCRN incorporates the message passing technique of the Hidden Markov Models (HMM). Since the transition matrix of the environment is hard to compute, we

use the Krylov basis to estimate the transition matrix, which outperforms the existing approximation bases. Unlike existing potential functions that only rely on states to perform reward shaping, we use both the states and actions through the Look-ahead advice mechanism to produce more precise advice. Our evaluations conducted on the Atari 2600 and MuJoCo games show that our solution outperforms the state-of-the-art that utilizes GCN as the potential function in most games in terms of the learning speed while reaching higher rewards.

5.1 Introduction

A Reinforcement Learning (RL) agent recognizes its position in the environment through a state defined by a Markov Decision Process (MDP). From a given state, the agent decides on the action that best maximizes the cumulative rewards. The action selection is based on a value function, which either considers states only, or states and actions. The objective of the value function is to maximize the cumulative rewards by selecting the best action for each state. The reward function is defined through the MDP and is used to evaluate the actions taken. The value function is updated iteratively using an RL algorithm, such as Q-learning, SARSA(λ), Deep Q-Network (DQN), etc [80]. RL techniques are usually time consuming; therefore, speeding the learning process is crucial for practical consideration of RL solutions in various applications. For this purpose, reward shaping techniques were invented to alter the original reward function definition, thus helping the agent reach the optimal policy [81]. Different techniques exist for designing a reward shaping function. In this work, we focus on potential-based reward shaping [81] because it guarantees invariance with respect to the optimal policy for solving an MDP. Building the potential function is not straightforward for complex environments [82]. Therefore, a potential-based reward shaping solution is still an open problem. Hence, we propose in this chapter a novel potential-based reward shaping solution using Convolutional Neural Network (CNN).

Our first step in this chapter is defining the potential function using the probabilistic inference view of RL. Using the message passing technique of Hidden Markov Models (HMM) [17], we are able to calculate the probability of the agent belonging to an optimal trajectory given the state and action [16], which we consider as an effectiveness signal for accelerating the learning. We also claim that predicting the reward shaping value at the next timestep helps accelerate further the learning. Therefore, we propose using a Recurrent Neural Network (RNN) that takes as input the state transitions from time $t = 1$ to a timestep T and predicts the reward shaping value at $T + 1$. The loss function of the RNN uses the actual labels of the reward shaping values that are calculated using the message passing technique.

The message passing technique is used to calculate the forward and backward messages. As discussed in Section 2.1.4, calculating these messages is computationally expensive, especially when dealing with a large graph of states and transitions. To overcome this issue, we propose using a GCRN model that combines a Graph Convolutional Network (GCN) and an RNN, where the GCN is responsible for computing the message passing, while the RNN predicts the next reward shaping value. In other words, the GCRN is capable of studying the spatio-temporal dependencies using GCN and RNN respectively. Due to the recursive nature of GCN, it is straightforward to perform message passing by propagating information about rewarding states between the neighbors using a filter matrix. In various research proposals that use GCN, the graph Laplacian corresponds to the filter that represents the connection of nodes in the graph or approximates the transition matrix. For instance, the authors in [16] use the graph Laplacian to perform reward shaping using GCN, while assuming that the value function is smooth over the induced MDP graph. This smoothness is defined using the Sobolov norm [21]. This assumption results in a margin of error for approximating the value function that can be reduced by computing the Krylov basis [23]. In this chapter, we also argue that if the graph Laplacian affects the accuracy of

the value function approximation (VFA), it definitely affects using GCN for reward shaping with the filter of the graph Laplacian. We elaborate on the drawbacks of using the graph Laplacian instead of the Krylov basis in Section 2.1.7.

In this work, we propose using GCRN with the Krylov basis as the filter of GCN to produce reward shaping values. The actual labels in the GCRN training are resulted from calculating the forward and backward messages. Due to the computation complexity induced for calculating the message passing, as well as the difficulty of retrieving the full transition matrix of the environment, we train GCRN on a sample of the agent transitions. When using a sub-graph of transitions, the learning accuracy of GCN is not affected, because GCN propagates information in the graph and was commonly used to perform semi-supervised learning [20]. In order to further improve the performance of GCRN, we also propose adapting an *optional* Look-ahead advice mechanism to the training process. By using the Look-ahead advice, the potential function is a function of the state and action, instead of only the state. Adapting this mechanism helps produce more precise advice at the action level [24].

The full process of our proposed GCRN scheme is presented in Figure 5.1. We collect a sample of the agent transitions for each episode in the environment. Using these transitions, we form a sub-graph as shown in Figure 5.1 (the red sub-graph). Afterwards, the input to GCN $X(S, A)$ is formed from the states $S \subset \mathcal{S}$ and actions $A \subseteq \mathcal{A}$ of the MDP, which considers incorporating the Look-ahead advice mechanism. In addition, we build the Krylov basis K (i.e. the approximated transition matrix) using the augmented Krylov algorithm presented in Section 2.1.7. The input $X(S, A)$ and the approximated transition matrix K are employed to perform the GCN training. The output from GCN up to T, referred to as GC, is passed to an RNN network composed of Bi-directional Gated Recurrent Units (Bi-GRUs) to predict the reward shaping value at $T + 1$. Our choice of the GRUs over LSTM is due to the superior performance of GRUs in our evaluations. Moreover,

we propose using bi-directional networks to predict the agent decision from the future as well as the post. Therefore, we claim that bi-directional networks further contribute to the novelty of our proposed scheme.

As a summary, our contributions in this chapter are:

(1) A novel scheme for potential-based reward shaping using a GCRN architecture that combines GCN and Bi-GRUs and benfits from the probability inference view of RL.

(2) Train GCRN on a sample of transitions that predicts the reward shaping value at the next timestep.

(3) Use the Krylov basis as a filter for the GCRN, which outperforms the graph Laplacian.

(4) Adapt an optional Look-ahead advice to produce more precise advice for the agent.

The rest of this chapter is organized as follows. In Section 5.2, we present our proposed reward shaping solution that utilizes GCRN with the Krylov basis and Look-ahead advice. The evaluations conducted on the Atari and MuJoCo games are presented in Section 5.3 compared to various baselines. We conclude the chapter in Section 5.4 with an open discussion about the limitations and future plan.

5.2 Proposed Scheme

In this section, we list the steps for constructing the potential function of our reward shaping solution using GCRN. We also present an algorithm to obtain the Krylov basis to approximate the transition matrix, and another algorithm showing the training of our proposed GCRN for potential-based reward shaping.

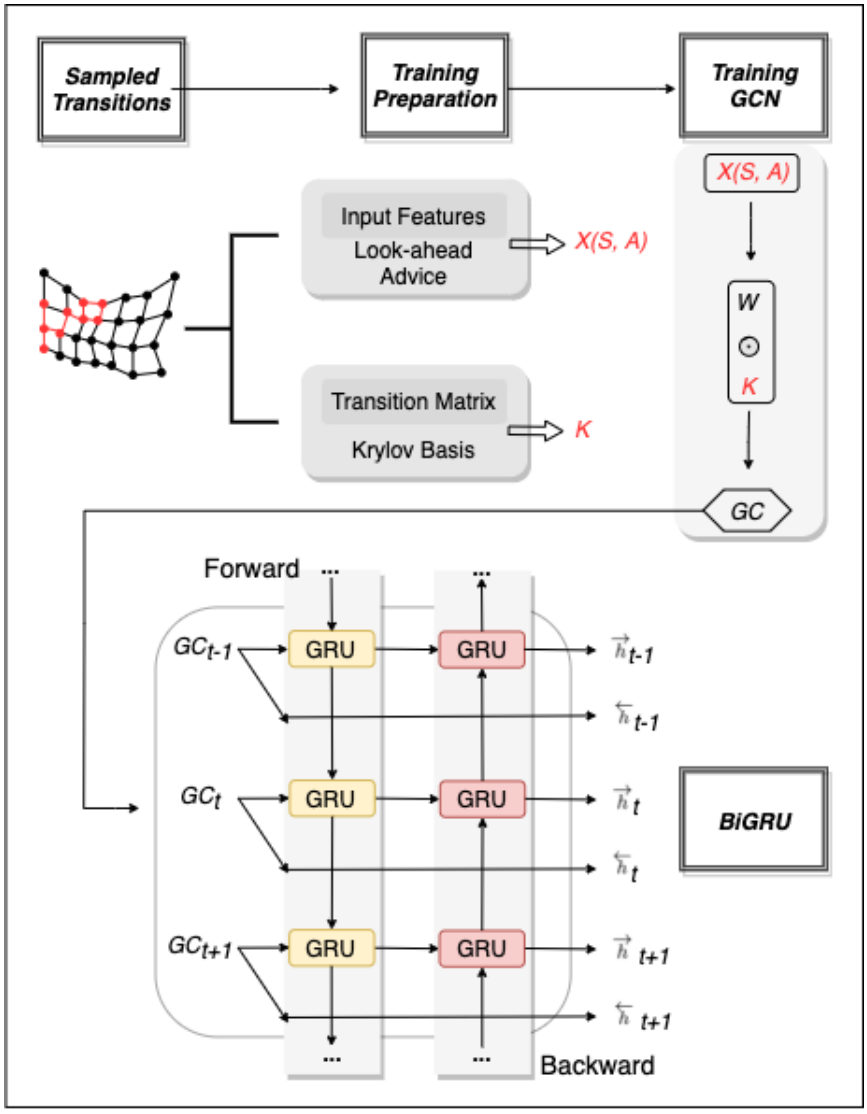


Figure 5.1: Proposed Scheme Using GCRN

5.2.1 GCRN Configuration

In this work, we propose GCRN, which combines GCN and RNN as shown in Figure 5.1. GCRN learns both the complex spatial dependencies and dynamic temporal dependencies in the graph of states. Our GCRN network is composed of GCN followed by Bi-GRU layers, where the vanilla GRU is used and the hidden layers are unchanged. However, the input to the RNN is the output of the preceding GCN. Noting that the input layer of the

GCN accepts states and actions as part of implementing the Look-ahead advice mechanism. Furthermore, the output of our GCRN is a probability distribution considered as the potential function output. The outputs of a GRU, reset, and update gates are computed as follows:

$$\begin{aligned} \mathbf{r}_t &= \sigma(W_r X_t + U_r h_{t-1} + b_r) \\ g_t &= \sigma(W_g X_t + U_g h_{t-1} + b_g) \\ \tilde{h}_t &= tanh[W_h X_t + U_h + \mathbf{r}_t \odot h_{t-1}) + b_h] \\ h_t &= (1 - g_t) \odot h_{t-1} + g_t \odot \tilde{h}_t \end{aligned} \tag{32}$$

In these equations, $\mathbf{r}$, g, and h are the reset, update, and the output gates respectively. In addition, X_t is the input at time t, σ is the logistic sigmoid activation, and $\odot$ is the Hadamard product. W_r, W_g, W_h, U_r, U_g, and U_h are the weight matrices. b_r, b_g, and b_h are the synthesis of bias vectors for the input X.

A Bi-GRU is composed of forward backward GRUs. The output vectors of the forward and backward GRUs are concatenated to get the final result. In our implementation, a forward pass in GCRN is expressed as follows:

$$\begin{aligned} GCN(X_t) &= \mathrm{K}\ ReLu(\mathrm{K} X_t W_1) W_0 \\ \overrightarrow{h_t} &= \mathrm{GRU}_{fwd}(GC(X_t), \overrightarrow{h_{t-1}}) \\ \overleftarrow{h_t} &= \mathrm{GRU}_{bwd}(GC(X_t), \overleftarrow{h_{t+1}}) \\ \phi_{GCRN}(X_t) &= LogSoftmax(\overrightarrow{h_t} \oplus \overleftarrow{h_t}) \end{aligned} \tag{33}$$

where GRU_{fwd} and GRU_{bwd} are computed following the steps in Equation 32. In this formulation, the output of GCRN as a potential function is expressed as ϕ_{GCRN}. X_t is the input matrix of size $(||S_t|| + ||A_t||)$, where $||S_t||$ and $||A_t||$ are the number of features in the state and action respectively; and K is the Krylov basis, which is computed through the augmented Krylov. The output of GCRN resembles the scaler value that gets appended to the original reward value of the transition being studied.

5.2.2 Loss Function

Updating GCRN for reward shaping entails using the message passing technique for calculating the network loss using the predicted and actual labels. The standard GCRN loss function is composed of the base and recursive cases in order to reflect the message passing mechanism as follows:

$$\mathcal{L} = \mathcal{L}_0(\mathbf{S}, \mathbf{A}) + \eta \mathcal{L}_{rec}(S, A) \tag{34}$$

where $\mathbf{S}$ and $\mathbf{A}$ are the lists of base case states and actions respectively, and S and A are the states and actions obtained from the sampled transitions. The base states and actions have a reward different from zero for the current episode, where each action is assigned to its state. It is important to propagate information in GCN from rewarding states and actions only. The actual labels for this loss are calculated using the forward and backward messages defined in Equation 3. Therefore, the base loss is calculated as follows:

$$\begin{aligned} \mathcal{L}_0 = H(p(O|\mathbf{S}, \mathbf{A}), \phi_{GCRN}(\mathbf{S}, \mathbf{A})) = \\ \sum_{s,a \in \mathbf{S},\mathbf{A}} p(O|s, a) log(\phi_{GCRN}(s, a)) \end{aligned} \tag{35}$$

where H represents the cross entropy loss between the actual and predicted values by the GCRN. In this approach, the input to the network considers both the state and action to benefit from the Look-ahead advice described in Section 2.1.6.

For the recursive case, the loss function takes the following form:

$$\mathcal{L}_{rec} = \sum_{i=1}^{||d||} \sum_{j=1}^{||e||} \mathfrak{A}_{i,j} ||\phi_{GCRN}(S_i, A_i) - \phi_{GCRN}(S_j, A_j)||^2 \tag{36}$$

where d and e are sets of identifiers for the states and their corresponding neighbors respectively. Furthermore, A_i and A_j are the actions taken at states S_i and S_j respectively, $\mathfrak{A}$ is

the adjacency matrix, and $\phi_{GCRN}(S_i, A_i)$ is the output of the shaping function for state S_i while selecting action A_i. Noting that the propagation model in GCRN uses the approximated transition matrix K to aggregate messages. Thus, a message in GCRN is written as $\mathrm{m}_i = \sigma(\sum_{j=1}^{||e||} \mathrm{K}_{i,j}\mathrm{m}_j)$, where m_j is a message from the neighbor j.

5.2.3 Computing the Krylov Basis

In our GCRN, a sample from the MDP is converted to a graph structure, where each node corresponds to a state. Furthermore, an edge resembles the transition between states for a given action, as part of the Look-ahead advice mechanism. Because constructing the whole graph iteratively is expensive when calculating the message passing, we consider using a sub-graph. The extracted sub-graph from the transition samples is sufficient to construct the shaping function [16]. To compute the Krylov basis K, we apply the augmented Krylov algorithm on the sub-graph. The resulted vectors from the augmented Krylov algorithm are appended to form the Krylov basis K as an approximation of the transition matrix. The full pseudo-code for computing K using the Krylov space and the *weighted spectral method* is presented in Algorithm 3.

Algorithm 3: Transition Matrix Approximation Using Augmented Krylov To Construct K

Input: P' - sampled transition matrix, r, e - number of eigenvectors from P', n - number of sampled transitions;
Output: K - Estimate of the transition matrix P^{π};
Compute top e eigenvectors of P': $\{q_0, q_1, \dots q_e\}$;
$q_{e+1} = r$;
for $i = 1, \dots n + e$ **do**
    if $i > e + 1$ **then**
        $q_i = P' q_{i-1}$;
    for $j = 1, \dots, (i - 1)$ **do**
        $q_i = q_i - (q_j . q_i) q_j$;
$\mathrm{K} = [q_0, q_1, \dots, q_e, \dots, q_n]$;
return K;

5.2.4 Training GCRN

A sample of the transition matrix P' is taken to form a sub-graph for training the GCRN every n steps. Once P' is retrieved, the augmented Krylov algorithm is applied to build the estimate of transitions matrix. The loss function of GCRN is applied following Equations 34, 35, and 36. The combined value function with reward shaping takes the form of $Q^{\pi}_{comb}(s,a) = \alpha Q^{\pi}(s,a) + (1-\alpha)Q^{\pi}_{\phi}(s,a)$, where $Q^{\pi}_{\phi}(s,a) = \mathbf{E}_{(s,a)}[\sum_t \gamma^t r(S_t, A_t) + \gamma\phi_{GCRN}(S_{t+1}, A_{t+1}) - \phi_{GCRN}(S_t, A_t)]$. Moreover, α is a hyperparameter indicating the amount of reward shaping decision used in the global value function.

Algorithm 4: Training GCRN

```
Create empty graph G;
for Episode=0,1,2, ... do
    for t = 1, 2, ..., T do
        Store transition (S_{t-1}, A_{t-1}, S_t, A_t);
        Build a graph of transitions in G;
    end
    if mod(Episode, N) then
        Build the sampled transition matrix P' from G;
        Construct Krylov basis K from P';
        Update GCRN using Equation (34);
    end
    Q^π_comb = αQ^π + (1 − α)Q^π_φ;
    Train DRL to maximize E_π[∇logπ(A_t|S_t)Q^π_comb(S_t, A_t)];
    Reset G to empty graph (optional);
end
```

5.3 Experiments

In this section, we provide a set of experiments for evaluating the performance of GCRN compared to: (1) Actor Critic (A2C) [83], (2) Proximal Policy Optimization (PPO) [84], (3) Random Network Distillation (RND), (4) Intrinsic Curiosity Module (ICM), (5) Learning Intrinsic Rewards for Policy Gradient (LIRPG), and (6) using GCN as the shaping

function with the graph Laplacian L_c as the filter [16]. RND and ICM are reward shaping solutions that improve exploration of actions. Furthermore, LIRPG seeks reward shaping through improving the agent performance, but it does not guarantee invariance with respect to the optimal policy. On the other hand, GCN and the proposed GCRN are potential based, scalable, and capable of improving the agent performance. LIRPG does not support continuous action spaces, thus its performance is studied in environments with discrete controls.

In the sequel, we first study the complexity of the proposed GCRN compared to the different baselines in the four rooms and four rooms traps games. Afterwards, we evaluate the performance of GCRN compared to the baselines in the Atari and MuJoCo games for discrete and continuous control.

5.3.1 Complexity

In order to study the runtime of each reward shaping method, we measure the number of frames processed per second (FPS) for each solution in Atari games. The number of FPS is a measure of the runtime for each method because it indicates the speed of learning the policy and the potential function in the environment. For every episode in GCRN, eigenvector decomposition on the sampled transition matrix is performed to compute the Krylov basis K. To avoid the expensive computation to retrieve the eigenvectors, we utilize the Singular Value Decomposition (SVD) for extracting the top eigenvectors from the sampled transition P' [85]. The time complexity for using SVD to calculate the top eigenvectors of a matrix is $O(mde)$, where m is the mean vector of the input, d is dimension, and e is the number of eigenvectors to compute. Furthermore, training GCRN consumes additional time every episode due to training the GCN and GRU layers. Therefore, GCRN has a slightly slower execution time compared to PPO and GCN. Despite this, GCRN is faster compared to RND, ICM, and LIRPG and has the best overall performance. In Table 5.1,

we provide a comparison of the frame processing time (FPS) between PPO, GCN, and our proposed GCRN for potential-based reward shaping.

Method	FPS
PPO	1122
GCRN	1028
GCN	1054
RND	1002
ICM	896
LIRPG	274

Table 5.1: FPS - Atari

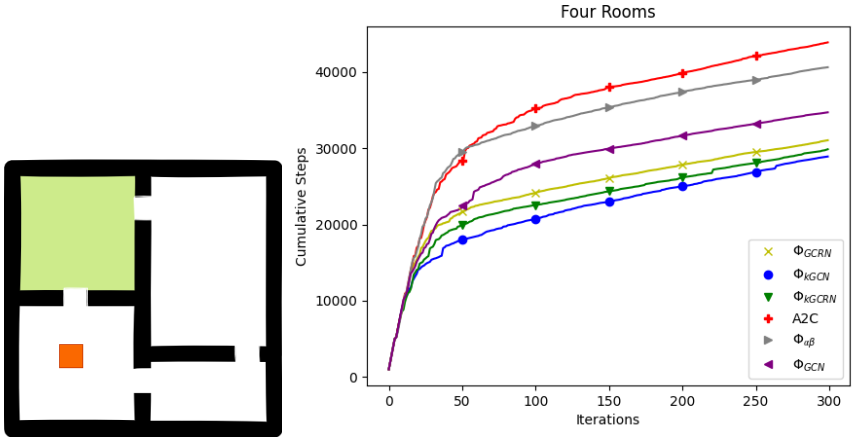
5.3.2 Performance Evaluation

To evaluate the performance of the proposed reward shaping approach, we study the impact of each of the proposed techniques for building GCRN as a potential function. Thus, we analyze the impact of (1) ϕ_{kGCRN}: GCRN using Krylov basis and look-ahead advice; (2) ϕ_{kGCN}: GCRN with Krylov basis; and (3) ϕ_{GCRN}: GCRN with look-ahead advice, compared to A2C and ϕ_{GCN} of [16] in tabular learning. We evaluate the performance of these proposed techniques compared to PPO, RND, ICM, LIRPG, and ϕ_{GCN} in 20 Atari for discrete control. Furthermore, additional experiments are performed on four Mujoco games for continuous control compared to PPO and ϕ_{GCN}.

Tabular

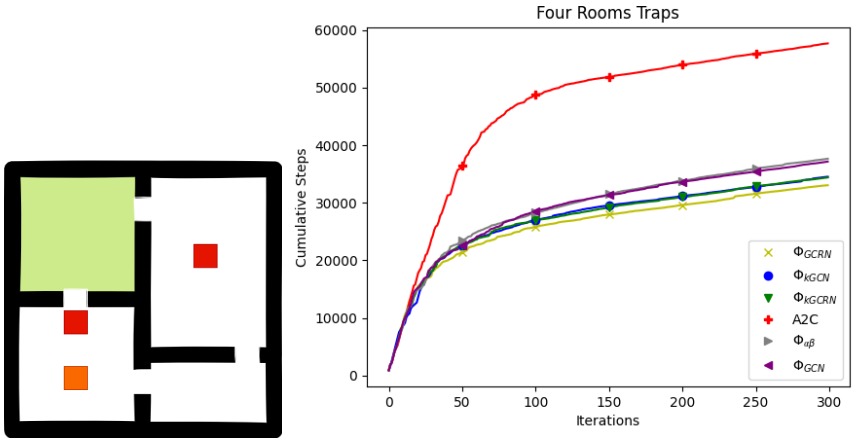
We developed two versions of the Four Rooms game to compare the learning speed and analyze the impact of ϕ_{kGCRN}, ϕ_{kGCN}, and ϕ_{GCRN} compared to A2C, ϕ_{GCN}, and $\phi_{\alpha\beta}$ as baselines, where $\phi_{\alpha\beta}$ is using the pure message passing. The two games are Four

Rooms and its variant Four Rooms Traps, where negative rewards are scattered through the room as traps. We use tabular learning in the form of A2C, where the critic uses λ-return. In such games, it is possible to calculate the results of the message passing because the environments are small. In addition, random actions are selected with a probability of 0.1. The results showing the cumulative steps to reach the goal are presented in Figures 5.2 and 5.3. As shown in these figures, the performance of ϕ_{kGCRN}, ϕ_{kGCN}, and ϕ_{GCRN}



(a) Four Rooms **(b)** Convergence Speed

Figure 5.2: Convergence speed of different solutions in the Four Rooms game



(a) Four Rooms Traps **(b)** Convergence speed

Figure 5.3: Convergence speed of different solutions in the Four Rooms Traps game

outperforms the rest of the baselines. Furthermore, the best performance can vary between

the proposed mechanisms depending on the hyperparameters used. These simple games show the importance of appending RNN to the GCN architecture. The BiGRU layer makes use of the message passing results that require a memory for the forward and backward passes in the environment. Therefore, predicting the shaping value at the next time-steps while considering what happened in the past, contributes to speeding the learning. In the next experiment, we perform our evaluations on the Atari 2600 games, where message passing cannot be directly computed.

Atari 2600

Atari 2600 games are sufficient for comparing the performance between the proposed ϕ_{kGCRN} and ϕ_{GCN} in terms of discrete action space. The Atari games offer a range of environments, which we use to show the importance of the combined solution, rather than using the separate components of augmented Krylov, look-ahead advice, and RNN. In order to add the action to the state vector, we utilize the one-hot-encoding and concatenate the state and action vectors. This step is required to evaluate the look-ahead advice mechanism in our proposed solution.

The performance of ϕ_{kGCRN}, ϕ_{GCRN}, ϕ_{kGCN}, ϕ_{GCN}, and PPO is evaluated on all Atari games. The same parameters were used for all the learning solutions for fair comparison. The parameters for the Atari games are shown in Table 5.2. Using GYM python dependency, the pixel rows are passed to a CNN for feature extraction. The input to GCRN is the output of the last hidden layer of the CNN. The experiments were executed for ten million steps for each game.

In Figure 5.4, we show the improvement achieved by ϕ_{kGCRN}, ϕ_{GCN}, RND, ICM, and LIRPG compared to PPO in log scale. Following the results displayed in Figure 5.4a, ϕ_{kGCRN} learns faster compared to PPO in all the games except RoadRunner. This implies

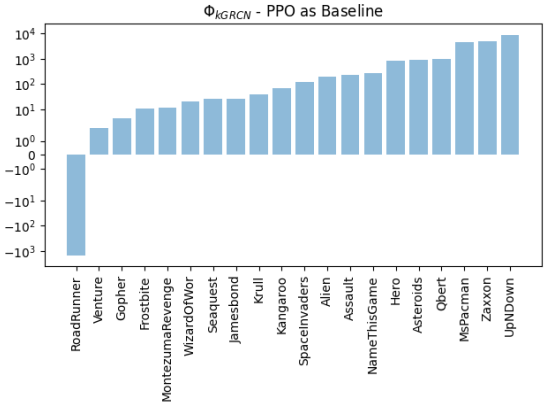
Hyperparameter	Value
Learning rate	2.5e-4
γ	0.99
λ	0.95
Entropy Coefficient	0.01
PPO steps	128
PPO Clipping Value	0.1
# of minibatches	4
# of processes	8
GCRN: α	0.9
GCRN: η	1e1
GCN: α	0.9
GCN: η	1e1

Table 5.2: Model configuration for the Atari games

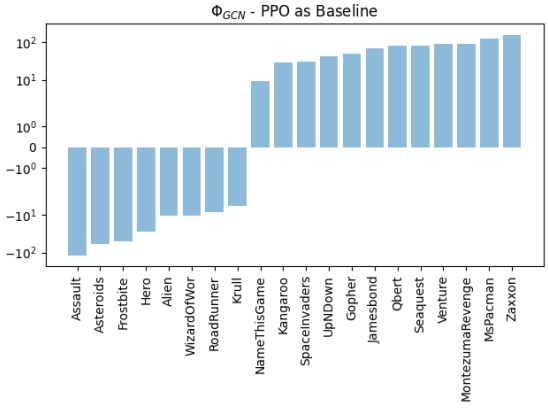
that ϕ_{kGCRN} achieves the best results in terms of the number of games where an improvement is achieved over PPO as baseline.

Furthermore, to show the importance of using GCN combined with the use of augmented Krylov, look-ahead advice, and BiGRUs, we study the improvement of each of the three proposed techniques (ϕ_{kGCRN}, ϕ_{kGCN}, and ϕ_{GCRN}) compared to ϕ_{GCN}. The results are presented in Figure 5.5 in log scale compared with ϕ_{GCN}. We chose to compare with ϕ_{GCN} as baseline because it is the most related to our proposed solution, and it achieves significant advancement in the domain of reward shaping.

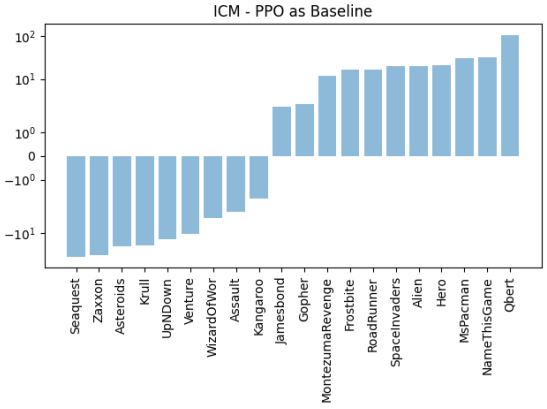
Comparing Figure 5.5a with 5.5b and 5.5c, we notice the importance of using the combined solution. Considering for example the MsPacman and UpNDown, we can see that ϕ_{kGCRN} improves in terms of convergence speed and maximum reward reached compare to ϕ_{GCN}, ϕ_{kGCN}, and ϕ_{GCRN}. In contrast, taking the example of the RoadRunner game, we can see that ϕ_{GCN}, ϕ_{GCRN}, and ϕ_{kGCN} performed better. Besides, ϕ_{kGCRN} does not always provide the best performance compared to ϕ_{GCN}. Similar to the tabular learning case, the selection of the hyperparameters also affects the performance of each solution. Therefore, studying the spatial and temporal dependencies for the Atari games by using GCRN as the potential function improves the overall learning quality compared to ϕ_{GCN}.



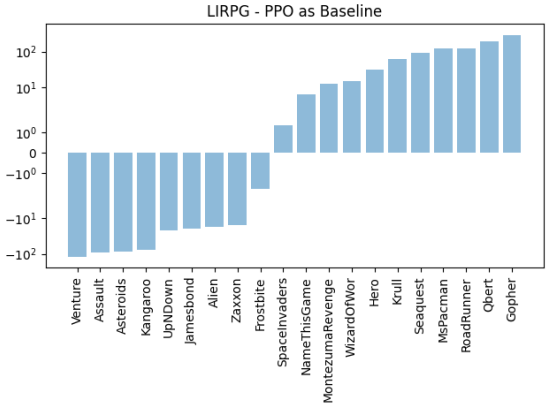
(a) ϕ_{kGCRN} performance different Atari games in log scale over PPO



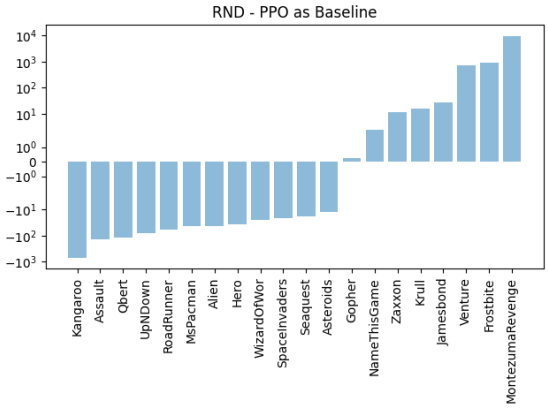
(b) GCN performance different Atari games in log scale over PPO



(c) ICM performance different Atari games in log scale over PPO



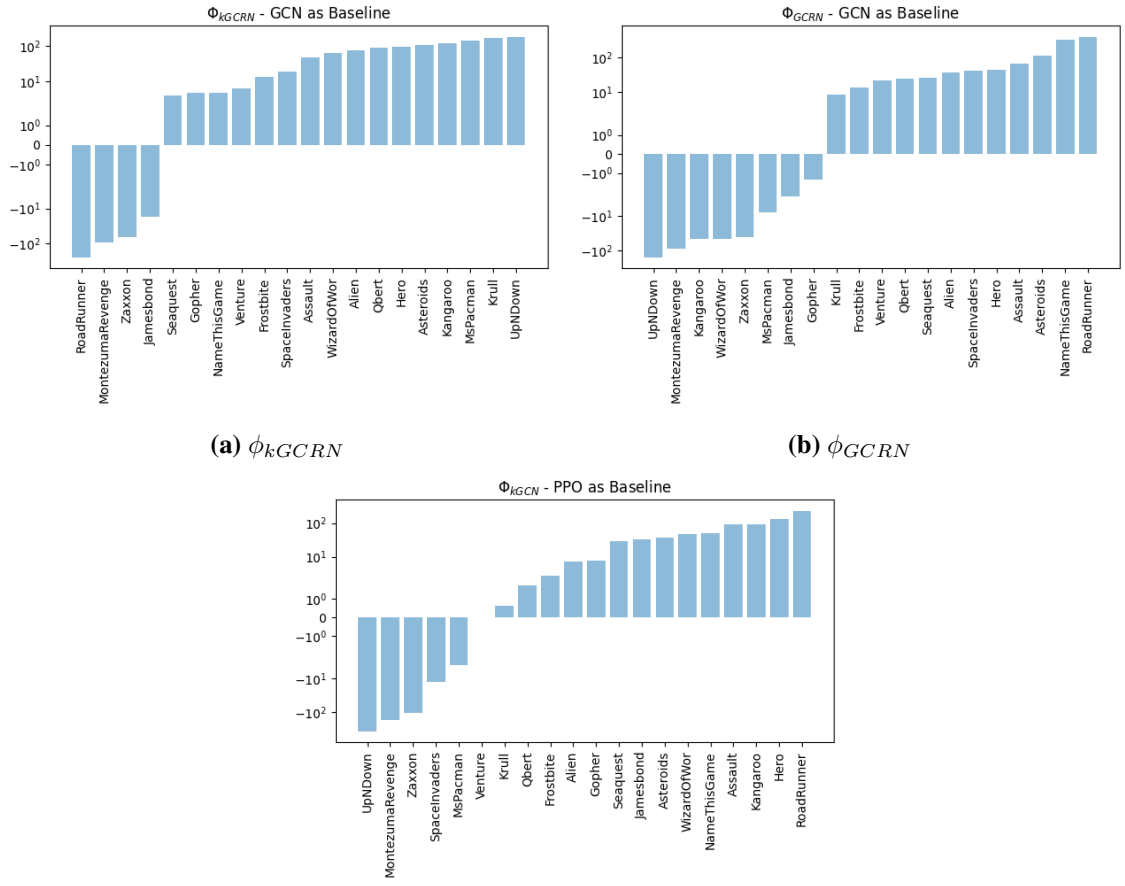
(d) LIRPG performance different Atari games in log scale over PPO



(e) RND performance different Atari games in log scale over PPO

Figure 5.4: Performance comparison between the proposed ϕ_{kGCRN} and different baselines in Atari games

Moreover, adding augmented Krylov, look-ahead advice, or both, can also result in improving the learning speed and maximum reward achieved. As a conclustion, there is no one best solution for all the Atari games, thus ϕ_{kGCRN}, ϕ_{GCRN}, ϕ_{kGCN}, and ϕ_{GCN} should be tried when tested on similar environments. In Figure 5.6, the average reward of each of the solutions for different games is shown.



(a) ϕ_{kGCRN} **(b)** ϕ_{GCRN}

(c) ϕ_{kGCN}

Figure 5.5: Performance comparison of the improvement achieved in different Atari games in log scale over ϕ_{GCN}

Mujoco

To further investigate the performance of our proposed model compared to ϕ_{GCN} and PPO, we perform experiments on continuous action space using the Mujoco environments.

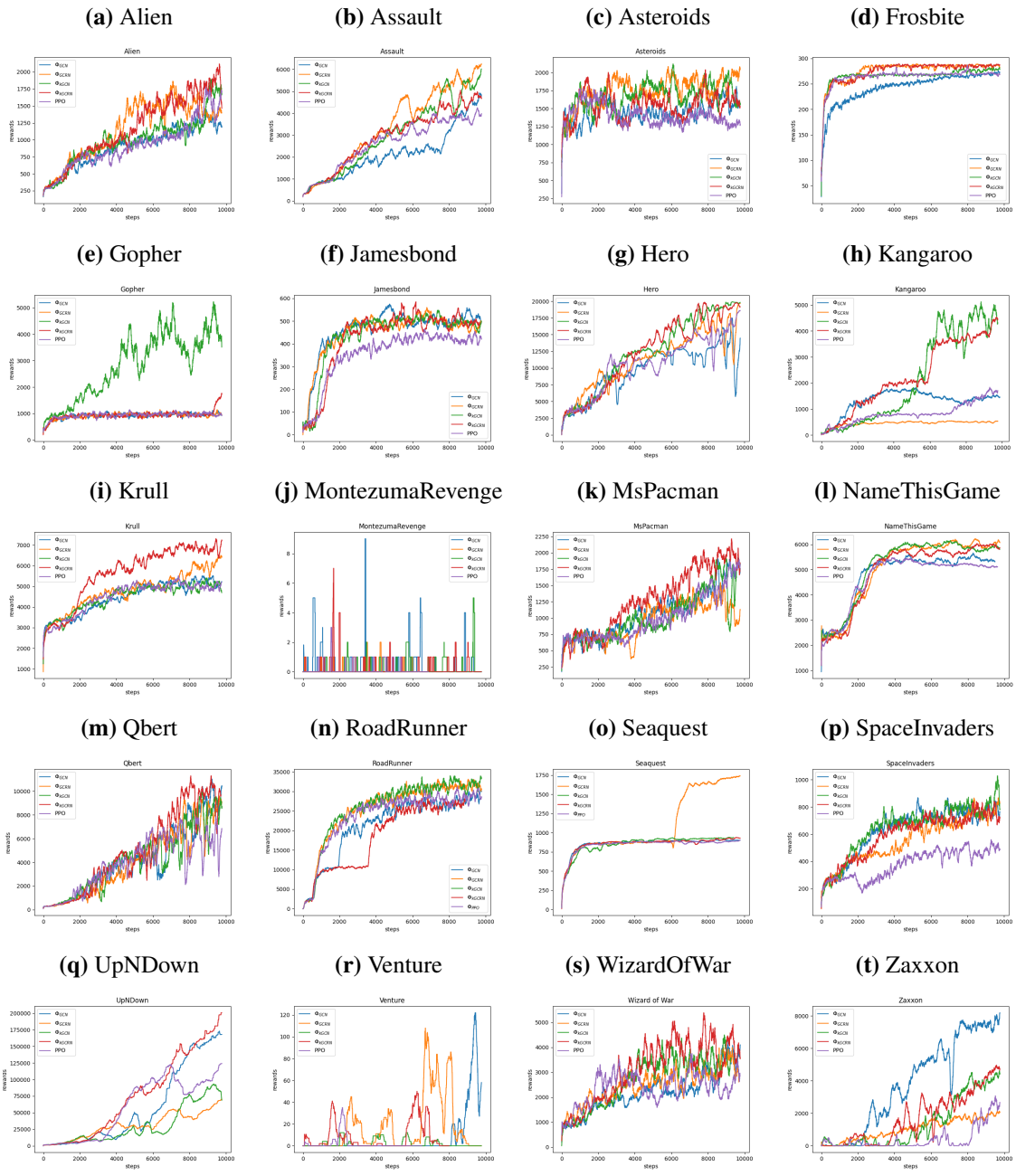


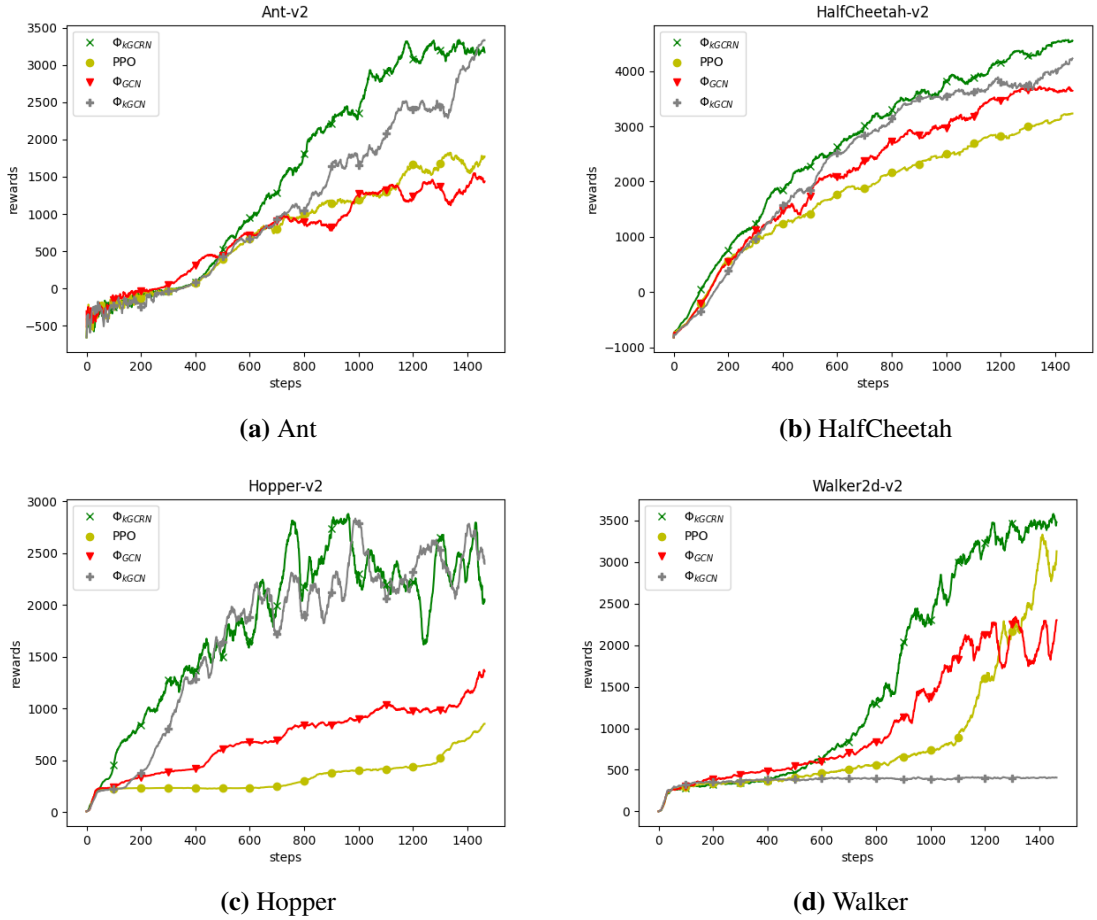
Figure 5.6: Results on 20 Atari games comapring the performance of ϕ_{CNN} to ϕ_{GCN} and PPO

We also use these environments to illustrate the advantage of combining the spatial and temporal learning through GCRN. We evaluate the performance of the three different proposed reward shaping techniques: ϕ_{kGCRN}, ϕ_{GCRN}, and ϕ_{kGCN}. The results comparing the performance of each game and technique, in terms of average rewards, are shown in Figure 5.7. We use the same parameters for evaluating the different techniques compared to ϕ_{GCN} for fair comparison. The experiments were executed for three million steps for each game. The model parameters for MuJoCo evaluation are provided in Table 5.3.

Hyperparameter	Value
Learning rate	3e-4
γ	0.99
λ	0.95
Entropy Coefficient	0.0
PPO steps	2048
PPO Clipping Value	0.1
# of minibatches	32
# of processes	1
GCRN/GCN (Walker and Ant): α	0.6
GCRN/GCN (Hopper and HalfCheetah): α	0.6
GCRN/GCN: η	1e1

Table 5.3: Model configuration for the MuJoCo games

Using our approach, learning on continuous action spaces is possible; however, the evaluation should be done for the different models with varying hyperparameters. It is not guaranteed that ϕ_{kGCRN} always achieves the best performance. Similar to the Atari games, ϕ_{kGCRN} offers the best performance among the remaining approaches. These results further highlight the importance of combining the different techniques to form ϕ_{kGCRN}. In the games of Ant, HalfCheetah, and Hopper, ϕ_{kGCN} is offering a better performance compared to ϕ_{GCN}, thus showing the advantage of using the augmented Krylov instead of graph Laplacian to approximate the transition matrix.



(a) Ant

(b) HalfCheetah

(c) Hopper

(d) Walker

Figure 5.7: Performance comparison between different reward shaping mechanisms and PPO in Mujoco environments.

5.4 Conclusion and Discussion

Our work proposes a novel GCRN scheme for potential-based reward shaping, which guarantees invariance in the optimal policy. The shaping function of GCRN combines layers of GCN followed by RNN to capture spatio-temporal dependencies between the sampled states. The training of GCRN is performed on a sample of transitions. Our solution embeds the look-ahead advice methodology and uses the augmented Krylov algorithm to estimate the transition matrix. Computing the actual labels of our GCRN is inspired by the probabilistic view of RL to perform message passing. The proposed GCRN excels

in terms of convergence speed compared to existing potential-based reward shaping solutions. ϕ_{kGCRN} achieves state of the art learning speed in some of the games. Besides, ϕ_{GCRN} and ϕ_{kGCN} also outperform other solutions in particular environments. Therefore, we recommend trying the three techniques we experimented with when testing the agent performance including the use of the Krylov basis and look-ahead advice.

The computational complexity of training the ϕ_{kGCRN} varies depending on the size of sampled transitions. In addition, we believe that the sub-graph selection can be improved by capturing more important information from the trajectories traversed, to be included in future samples. Furthermore, we aim to augment our solution to be deployed in multi-agent systems while trying to optimize multiple objectives.

Later on, we believe that the proposed GCRN can be applied to a wider range of applications utilizing RL for solving time-sensitive problems. The results achieved improve the learning speed, which has direct impact on the feasibility of RL in various applications. For example, fast decisions are essential for autonomous driving [86, 87], resource management [88], task scheduling [89], trust-driven reinforcement selection for federated learning [90] and health-related applications.

Chapter 6

Reward Shaping Using Convolutional Neural Network

In this chapter, we propose Value Iteration Network for Reward Shaping (VIN-RS), a potential-based reward shaping mechanism using Convolutional Neural Network (CNN). The proposed VIN-RS embeds a CNN trained on computed labels using the message passing mechanism of the Hidden Markov Model. The CNN processes images or graphs of the environment to predict the shaping values. Recent work on reward shaping still has limitations towards training on a representation of the Markov Decision Process (MDP) and building an estimate of the transition matrix. The advantage of VIN-RS is to construct an effective potential function from an estimated MDP while automatically inferring the environment transition matrix. The proposed VIN-RS estimates the transition matrix through a self-learned convolution filter while extracting environment details from the input frames or sampled graphs. Due to (1) the previous success of using message passing for reward shaping; and (2) the CNN planning behavior, we use these messages to train the CNN of VIN-RS. Experiments are performed on tabular games, Atari 2600 and MuJoCo, for discrete and continuous action space. Our results illustrate promising improvements in the learning speed and maximum cumulative reward compared to the state-of-the-art. The

improvement achieved by VIN-RS can only be observed for some of the games due to the underlying nature of some environments. In terms of the studied MuJoCo games, there is on average an increase of 30% in the maximum reward reached during early stages of learning.

6.1 Introduction

A Reinforcement learning (RL) algorithm is executed against a Markov Decision Process (MDP) environment. The MDP environment is sketched by the solution designer where the agent can perform actions decided by RL. The agent sends feedback to the RL solution with a reward value to update the value function based on the actions taken. Therefore, an accurate representation of the reward function is vital for future action selection. Hence, dynamicity in the structure of the reward function is required to adapt to environmental changes and produce more effective rewards. Consequently, better rewards lead to faster convergence to near optimality with regard to agent decisions. MDP environments have different types with continuous or discrete, finite or infinite states, and action spaces. Furthermore, a transition matrix deciding the next state of the agent is most of the time unknown. These MDP properties make it challenging to develop a scalable and dynamic reward function [80].

RL algorithms are slow to converge, where most of the time is spent on exploration at the early stages of learning. There are multiple learning speedup techniques for RL such as offline learning, dynamic exploration, transfer learning, imitation learning, and reward shaping [91, 92]. Reward shaping alters the original reward function with values generated from a shaping function. The shaping values speed learning in RL by directing the reward function to speed the policy convergence [81]. One of the reward shaping mechanisms is potential-based, which ensures that updates to the original reward function do not affect the ability of an agent to reach optimal policy decisions. Due to the different types of MDP

and dynamicity of the environment, it is difficult to design a scalable and effective potential function for reward shaping that is suitable for most environments [82]. The existing reward shaping solutions suffer from one or more of the following limitations: (1) they alter the optimal policy; (2) they are based on action exploration only [93]; (3) they are not applicable in different environments; (4) they rely on transition matrix approximation; (5) have limited representation of sampled MDPs. The negative consequence of the first limitation is that obtaining the optimal policy cannot be guaranteed. Because of the second limitation, the reward function is never adapted to the environment. Due to the other limitations, the agent performance based on the potential function can be improved. The applicability of a reward shaping solution is measured by the performance in various environments and the dependability on external knowledge like expert feedback. Therefore, a potential-based reward shaping solution is still an open problem. *Hence, we propose in this chapter a novel potential-based reward shaping solution that is scalable, learns on a representation of the MDP either through frames of images or sampled MDP graphs, and estimates a transition matrix while training.*

The proposed potential function architecture follows the mechanism of the Value Iteration Network (VIN) and uses convolution layers to perform planning [28]. The original VIN module in [28] uses a Convolutional Neural Network (CNN) architecture, which can be trained using RL or Imitation Learning (IL). The CNN of the VIN can perform value iteration on an MDP for planning. The output of the CNN is part of an attention mechanism that selects actions as part of the optimal plan. In [28], learning is either done using IL, thus requiring a large number of labels, or using RL whose performance is poor on irregular graphs. Irregular graphs are problematic when training VIN with RL because the number of actions for the neighbors varies.

The proposed VIN-RS trains a novel CNN based on the probabilistic view of RL to

serve as a potential function. Our new scheme, named Value Iteration Network for Reward Shaping (VIN-RS), incorporates this novel training mode and new architecture tailored for reward shaping. Computing message passing on Hidden Markov Models (HMM) [17], composed of forward and backward messages, derives the probability of trajectory optimality, thus can be used to redirect the reward function and speed learning [16, 17]. Computing these messages in a large environment requires high computation as discussed further in Section 2.1.4. On one hand, message passing helps the agent decide if the current state belongs to an optimal trajectory. On the other hand, the CNN module of VIN provides a goal-oriented plan for an optimal trajectory. In this chapter, we will argue that the message passing and VIN combined would help accelerate learning by acting as the potential function. Therefore, we aim to train the proposed CNN of VIN-RS based on the message passing loss computed from sampled trajectories followed by the agent.

In [16], the authors propose a potential-based reward shaping solution using Graph Convolution Network (GCN). GCN has the potential in propagating the forward and backward messages of HMM using the graph operation for sharing information between nodes. Despite significant improvement achieved over existing reward shaping solutions, using GCN as the potential function still has some limitations, and it cannot generalize to all environments. Even though both CNN and GCN belong to the family of Graph Neural Networks (GNN), GCN performs message passing on a sample of the states, while CNN uses full images of the environment, which can reveal more states and speed planning. In addition, to perform the GCN layer operation, the transition matrix of the MDP should be estimated [16]. Assuming that the value function is smooth over the MDP graph, the graph Laplacian is used as an approximation of the transition matrix, resulting in a margin of error [23]. Furthermore, GCN has issues related to MDP representation and information extraction, which is due to the graph approximation technique used to represent the environment. Compared to GCN, the proposed CNN learns a representation of this transition

matrix while training. More specifically, VIN-RS estimates a new MDP, not necessarily related to the original one, which is learned using images of the environment and trained following the message passing values. Even when the environment is not captured as images, VIN-RS can train the CNN on a graph representation from this environment, which leads to promising performance as illustrated later in our experiments.

Training VIN-RS for reward shaping requires the true labels that are inferred using message passing. Therefore, the loss function incorporates the message passing mechanism. Due to (1) the computation complexity induced by calculating the message passing; and (2) the difficulty of retrieving and sometimes approximating the transition matrix of the environment, we train CNN on image frames or samples of the environment graph capturing the current state of the agent. CNN uses a transition matrix that is trained as part of the network. The output of the proposed CNN is a regression value for each possible action in the environment that estimates if the current state of the agent belongs to an optimal trajectory. Furthermore, VIN-RS embeds the look-ahead advice mechanism [24]. By evaluating the state and action, the look-ahead advice improves the quality of the potential function and thus can improve the learning speed using reward shaping. Thus, VIN-RS produces advice at the state and action levels, which is at the core of the look-ahead advice mechanism. In addition, it is possible to apply our solution in discrete and continuous action spaces. Some of the potential applications for applying VIN-RS for supporting RL agents include autonomous driving and control [94], robotics, and video games. These three applications require complex environments with high-dimensional state and action spaces representation due to the involvement of multiple input devices, while making decisions in real time. Furthermore, these applications can also offer visual inputs, which is also suitable for the underlying VIN-RS architecture that utilizes CNN to extract useful information to help the agent learn more effectively. Furthermore, VIN-RS can be also combined with other machine learning mechanisms to speed the learning process, such as federated learning [90]

and resource management [95].

The contributions of this chapter are summarized as follows:

(1) VIN-RS for potential-based reward shaping using a novel CNN architecture.

(2) CNN trained on images or graphs from the environment while using a message passing mechanism in the loss function.

(3) Estimating the transition matrix through training the CNN.

(4) Overcoming the limitations of using GCN for reward shaping as the potential function by training on an estimated MDP using CNN.

(5) State-of-the-art results in most of the games for discrete and continuous action spaces.

In Figure 6.1, we present a graphical abstract summarizing our proposed approach, snapshot of the results on the Atari and MuJoCo games, and a conclusion of findings.

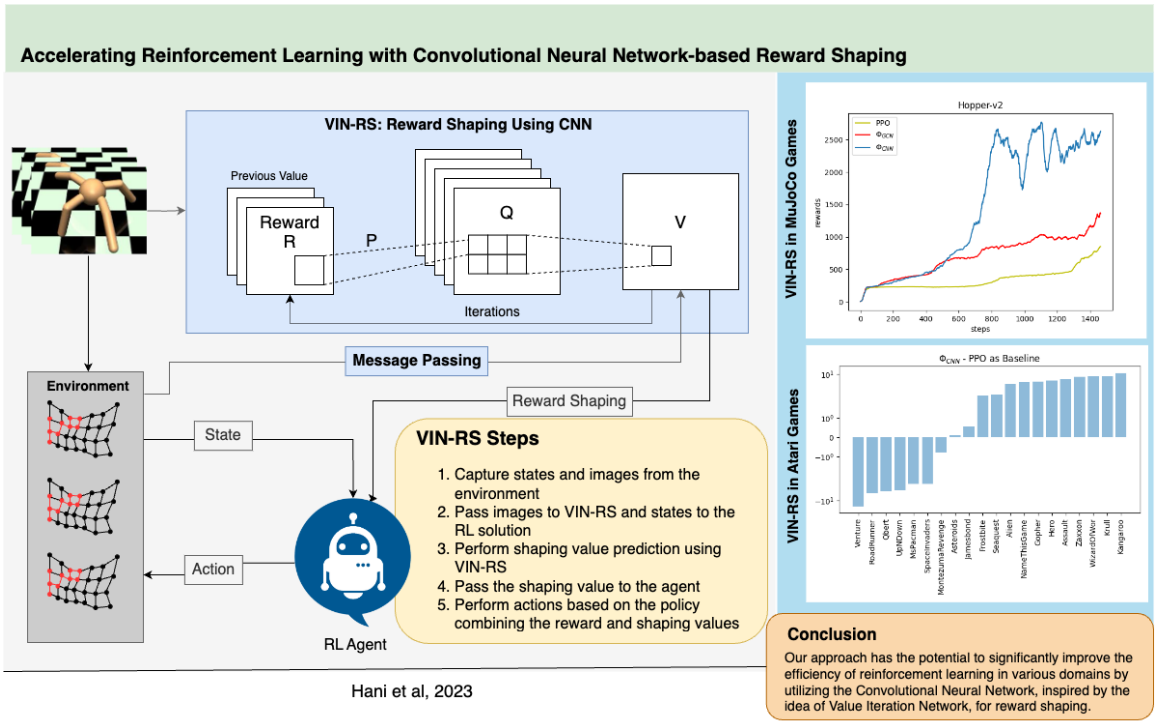


Figure 6.1: VIN-RS Graphical Abstract

The structure of the chapter is divided as follows. In Section 6.2, we present our proposed reward shaping solution "VIN-RS". The evaluations conducted on tabular, Atari and MuJoCo games are illustrated in Section 6.3. The results of our evaluation shows that VIN-RS achieves on average the best results in the Atari and MuJoCo games compared to all baselines. Finally, we present a discussion in Section 6.4 and summarize and conclude the chapter in Section 6.5 with future directions.

6.2 Proposed Scheme: VIN-RS

In this section, we start first by describing the overall proposed architecture which consists of three main components: (1) VIN-RS, (2) message passing, and (3) reinforcement learning algorithm. We describe each of the components and list the steps for constructing the potential function of our reward shaping solution using CNN and focus on the advantage of incorporating the look-ahead advice mechanism. We also discuss the technique to obtain the message passing values, which are used to compute the loss for the CNN. In addition, we show an algorithm to train VIN-RS. Finally, the policy update combining the output of CNN is presented in the main reinforcement learning algorithm.

6.2.1 Overall Architecture

In Figure 6.2, we show the overall proposed architecture that combines VIN-RS with RL. Our solution contains two main modules, the first is VIN-RS that uses a CNN on $\bar{\mathcal{M}}$, while the second is the main RL solution for the original MDP $\mathcal{M}$.

Starting with the main component which is the RL algorithm that computes the policy π to solve $\mathcal{M}$. The input to the RL module is the list of states which can either be images or states representation from the environment. These states are also called observations. Based on the observations, the agent selects the best action according to the policy π, then

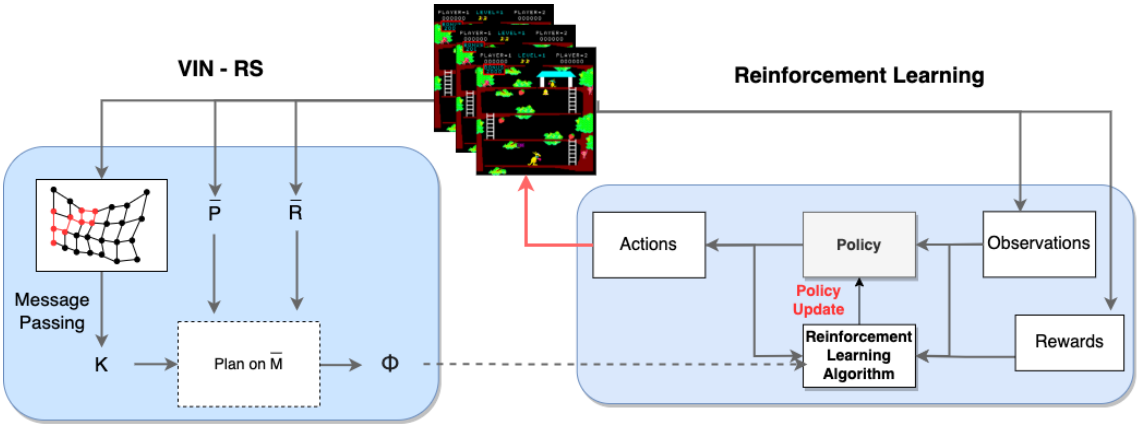


Figure 6.2: An architecture incorporating the Value Iteration Network for Reward Shaping with Reinforcement Learning

executes that action in the environment. The agent then receives the next state as well as the reward of the action taken in the environment. This information is used to update the policy using the RL algorithm. The policy then selects another action for the new observation and keeps on repeating these steps until the agent finds the optimal policy π^*. The novelty of our RL solution is that the policy update incorporates the reward shaping value, which is constructed using the output of VIN-RS.

The CNN of VIN-RS is built using three two-dimensional CNN layers. The input to CNN is a list of images captured from the environment. CNN accepts graphs, in contrast to the main RL algorithm where the input can be a state representation that combines information other than the environment images. This is similar to the case of MuJoCo environments [96]. The image is passed to the first conv layer that is responsible for processing the raw image pixels. The result of the first conv layer is passed to the second one, which is responsible of producing a reward matrix $\bar{\mathcal{R}}$. This layer has two channels, one holds the old value function matrix $\bar{\mathcal{V}}$, and one holds the current rewards. In other words, states are represented as a two-dimensional grid at each timestep, and each of these states has a reward value computed using the first layer. This layer is trained and improved over time by the network through continuous weights updates. The kernel applied to this layer

resembles the probability transition matrix $\bar{\mathcal{P}}$, which is also updated during the training. Applying this kernel to the first reward matrix resulted from the first conv layer will give us the state-action value function ($\bar{\mathcal{Q}}$-value). At this conv layer, there are x channels, where x is the number of actions in the action space $\bar{\mathcal{A}}$. Selecting the action having the maximum $\bar{\mathcal{Q}}$-value is done by applying a max-pooling for the $\bar{\mathcal{Q}}$ value of the corresponding states. The resulting matrix $\bar{\mathcal{V}}$ is passed to $\bar{\mathcal{R}}$ to be considered for the next value iteration or policy update. Furthermore, the $\bar{\mathcal{Q}}$ is flattened and a dense layer is applied to obtain the output layer. This output layer has the size of x, which is considered as a shaping value for each action, passed to the main policy update of the RL algorithm to update the policy using potential-based reward shaping. Producing a shaping value for each action of a given state is at the core of the look-ahead advice mechanism [24].

In order to train our CNN, we use the standard backpropagation by computing the labels using the message passing technique. As discussed in the background section, the message passing value including the forward and backward messages is considered as a signal that could accelerate learning. In order to compute the messages values, a graph of states is formed as shown in Fig. 6.3, this graph contains only a subset of the states. Due to the fact that computing message passing is computationally expensive for large graphs, it is enough to compute this message passing for the sampled graph of states for the current training iteration of CNN. For every training episode, the graph is emptied and a new one is formed. The output of the message passing algorithm is used as the true labels for the CNN to compute the loss function. In [16], message passing is implemented using a GCN; however, in this work, we overcome the limitations of using GCN, described in Section 2.2.3, and apply CNN to perform value iteration and compute the message passing values. Because CNN is used to perform planning in the network over $\mathcal{K}$ iterations, VIN-RS can tell if an agent state belongs to an optimal trajectory. The ability of CNN to plan using value iteration is mapped to what a message passing value represents.

6.2.2 VIN-RS Module

Our VIN-RS builds and solves $\bar{\mathcal{M}}$, where the parameters of the policy $\bar{\pi}$ include f_r and f_p. These two functions are differentiable and learned while training the CNN. The CNN is trained using the message passing result as the label. To obtain $\bar{\pi}^*$ for solving $\bar{\mathcal{M}}$, value iteration is applied as follows:

$$\bar{\pi}^*(\bar{s}) = \text{argmax}_{\bar{a}} \bar{\mathcal{R}}(\bar{s}, \bar{a}) + \gamma \sum_{\bar{s}'} \bar{\mathcal{P}}(\bar{s}'|\bar{s}, \bar{a}) \bar{V}^*(\bar{s}') \tag{37}$$

where $\bar{s}'$ is the state at the next timestep. Furthermore, we construct a VIN-RS tailored for reward shaping in the context of RL. Mainly, we train VIN-RS using the message passing values and consider its output as the shaping value that is used to update the original policy of the RL algorithm. An illustration on the training and integration between the CNN of VIN-RS and the RL module is shown in Figure 6.2. In addition, the CNN in VIN-RS is trained in a separate network to solve its own policy and not combined with the RL network. As shown in Figure 6.3, the input to CNN is the list of images extracted from the environment. The input can also be a graph representation of the states from the environment, in case images are not available. Using graphs instead of images can reduce the number of states the CNN trains on; however, it makes our solution applicable to high dimensional state spaces. In other words, using grid of pixels is practical when the environment image is two-dimensional or covers the current state and end goal. Using f_r, which corresponds to the weights of the first conv layer, the $\bar{\mathcal{R}}$ matrix is computed. $\bar{\mathcal{R}}$ has the dimension of l, m, n, where l is number of channels and m and n represent the image dimensions. The extracted reward $\bar{\mathcal{R}}$ is passed to the next conv layer, where the $\bar{\mathcal{Q}}$ values are computed. The $\bar{\mathcal{Q}}$ conv layer contains x channels, or a channel for each action in $\bar{\mathcal{M}}$. $\bar{\mathcal{Q}}_{\bar{a},i',j'}$ represents the $\bar{\mathcal{Q}}$-value for a state defined at positions between i, i' and j, j'

respectively for a particular action $\bar{a}$. $\bar{\mathcal{Q}}_{\bar{a},i',j'}$ is computed as follows [28]:

$$\bar{\mathcal{Q}}_{\bar{a},i',j'} = \sum_{l,i,j} W^{\bar{a}}_{l,i,j} \bar{\mathcal{R}}_{l,i'-i,j'-j} \tag{38}$$

In Equation 38, the reward matrix is multiplied by the weights or a representation of the transition matrix $\bar{\mathcal{P}}$. From the resulting $\bar{\mathcal{Q}}$ matrix, we apply max-pooling by selecting the highest $\bar{\mathcal{Q}}$-value from the list of actions or channels to obtain $\bar{\mathcal{V}}$. An element in $\bar{\mathcal{V}}$ at i, j is:

$$\bar{\mathcal{V}}_{i,j} = \max_{\bar{a}} \bar{\mathcal{Q}}(\bar{a}, i, j) \tag{39}$$

where $\bar{\mathcal{Q}}(\bar{a}, i, j)$ is the result of the $\bar{\mathcal{Q}}$ function for a given state and action. Following the computation of $\bar{\mathcal{V}}$, we update the first channel of the $\bar{\mathcal{R}}$ matrix. In addition, a dense layer is applied after flattening $\bar{\mathcal{Q}}$. Finally, a fully connected output layer is added, which results in the shaping values for each action. In order to obtain the true labels for the shaping value, we apply the message passing mechanism on the extracted graph from the observations, and pass this value to the loss of CNN. A forward pass in this CNN is considered as a single value iteration. Assuming that the number of steps required by the agent to reach the goal from the current state is $\mathcal{K}$, then the ideal number of value iterations required in CNN for value function updates is $\mathcal{K}$. After each iteration, $\bar{\mathcal{V}}$ is calculated using $\bar{\mathcal{Q}}$ and appended to $\bar{\mathcal{R}}$ for the next iteration, as shown in Figure 6.3. In Algorithm 5, we show a pseudo-code of the training steps in our proposed CNN. The pseudo-code of Algorithm 5 presents how the CNN of VIN-RS is trained for a single step. Lines 1-5 perform a forward pass in the algorithm. Lines 6-10 perform $\mathcal{K}$ value iterations. Lines 11-13 flatten the $\bar{\mathcal{Q}}$ matrix and obtain the output ϕ. Lines 14-15 compute the message passing values and compute the loss. Line 16 performs backpropagation to compute the gradients of the network weights. Line 17 updates the weights based on the computed gradient.

Algorithm 5: A training step with $\mathcal{K}$ value iterations of CNN in VIN-RS

Input: X: a list of images sampled from the environment.
$\mathcal{G}$: a graph constructured from the sampled images or the list of states encountered.;
$cnnH$: Conv layer that processes the image pixels;
$cnnR$: Conv layer that computes the $\bar{\mathcal{R}}$ matrix;
$cnnQ$: Conv layer that computes the $\bar{\mathcal{Q}}$ matrix;
wV: Initialize weights for computing $\bar{\mathcal{Q}}$ matrix;
Fn: Fully connected layer;
Opt: Output layer;
Output: ϕ: shaping value for the corresponding images/states.;
Function *evaluateQ*$(r, v, cnnQ, wV)$\{ ▷ Perform value iteration;
    $rv = \text{Concat}(r, v)$ ▷ Concatenate r and v;
    $wQwV = \text{Concat}(cnnQ.\text{weights}, wV)$;
    $q = Conv(rv, wQwV)$ ▷ Apply a Convolutional layer;
    return q;
};
$p = \text{Normalize}(X)$ ▷ or G as input when applicable;
$h = cnnH(p)$;
$r = cnnR(h)$;
$q = cnnQ(r)$;
$v = \text{Max}(q)$ ▷ Get the maximum or apply max pooling;
for $i = 1, \ldots \mathcal{K}$ **do**
    ▷ Perform $\mathcal{K}$ value iterations;
    q = *evaluateQ*$(r, v, cnnQ, wV)$;
    $v = \text{Max}(q)$;
$q =$ *evaluateQ*$(r, v, cnnQ, wV)$;
$v = \text{Max}(q)$;
$flatten\_q = \text{Reshape}(q)$ ▷ Flatten the matrix v;
$fn = Fn(flatten\_q)$;
$\phi = Opt(fn)$;
$label = \text{Message\_Passing}(\mathcal{G})$;
$loss = \mathcal{L}(label, \phi)$;
$backpropagate()$;
$update\_gradient()$;

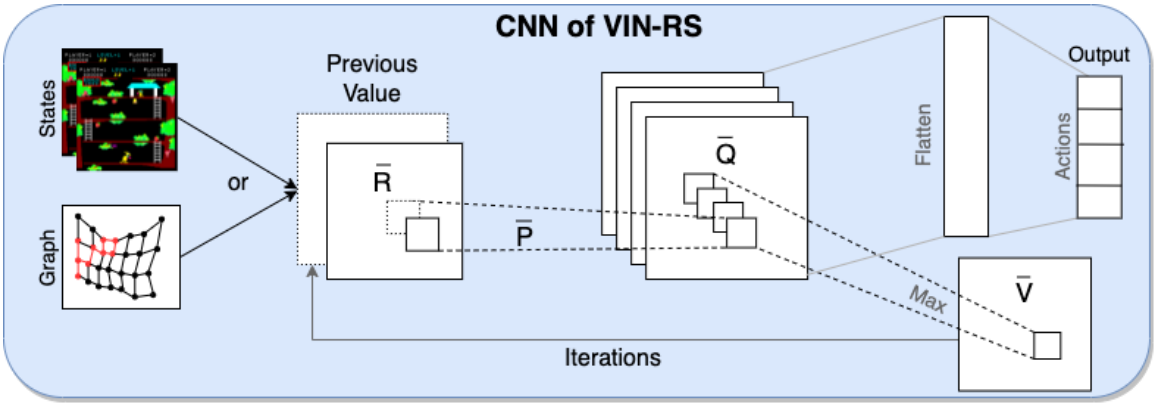


Figure 6.3: The CNN architecture of the proposed VIN-RS module

6.2.3 Loss Function: Message passing

Following the success of using message passing for reward shaping, we propose training the CNN of VIN-RS using the message passing values as the true labels. Using the probability inference view of RL, a solution is to find the distribution of the optimality variable $\mathcal{O} = 1$ for a given state and action. This structure is analogous to HMM, where forward and backward messages can be used to compute this probability distribution. Combining the forward and backward messages results in a policy that looks backward and forward in time. In other words, the resulting values from message passing tells if a state belongs to an optimal trajectory. More details about the messages calculation is presented in Section 2.1.4. Thus, the combined forward and backward messages (α and β) are expressed as follows:

$$p(O_t|S_t, A_t) \simeq \alpha(S_t, A_t)\beta(S_t, A_t) \tag{40}$$

Thus, the potential function is expressed as $\phi_{\alpha,\beta} = \alpha(S_t, A_t) \times \beta(S_t, A_t)$. Compared to VIN, a CNN can also produce optimal plans/trajectories for the agent on $\bar{\mathcal{M}}$. Based on this observation, we propose VIN-RS that incorporates the message passing results in the training process to produce the shaping values.

In VIN-RS, we benefit from message passing to compute the loss of CNN. To compute those messages, base and recursive losses are required. In our loss formulation, the

state and action are passed as input when computing the loss. Such formulation boosts the performance of VIN-RS by applying the look-ahead advice naturally in the CNN implementation, due to the fact that each channel in $\bar{\mathcal{Q}}$ is a Q-function for each action in the environment. Therefore, the loss function is computed in two steps as follows [16]:

$$\mathcal{L} = \mathcal{L}_0(\bar{\mathbf{S}}, \bar{\mathbf{A}}) + \eta \mathcal{L}_{rec}(\bar{\mathcal{S}}, \bar{\mathcal{A}}) \tag{41}$$

where $\mathcal{L}_0$ is the base case, $\mathcal{L}_{rec}$ is the recursive case, $\bar{\mathbf{S}}$ and $\bar{\mathbf{A}}$ are the lists of base case states and actions respectively in $\bar{\mathcal{M}}$, and $\bar{\mathcal{S}}$ and $\bar{\mathcal{A}}$ are the states and actions retrieved from the graph of experiences. Noting that the base states are composed of the rewarding states. In VIN-RS, we consider states that are used to form the graph $\mathcal{G}$, where each state in this graph is an images of the environment. Furthermore, the base loss is computed using the rewarding states only in $\mathcal{G}$, in order to extract information only from important states. The base loss is the usual supervised loss that considers the true and predicted labels.

$$\begin{aligned} \mathcal{L}_0 = H(p(\bar{O}|\bar{\mathbf{S}}, \bar{\mathbf{A}}), \phi(\bar{\mathbf{S}}, \bar{\mathbf{A}})) = \\ \sum_{s,a \in \bar{\mathbf{S}}, \bar{\mathbf{A}}} p(\bar{O}|s,a) \log(\phi(s,a)) \end{aligned} \tag{42}$$

The recursive loss written as $\mathcal{L}_{rec}$ is computed by aggregating the messages with the neighboring states using the adjacency matrix of the graph $\mathcal{G}$. $\mathcal{L}_{rec}$ is formulated as follows:

$$\mathcal{L}_{rec} = \sum_{i=1}^{||d||} \sum_{j=1}^{||e||} \mathfrak{A}_{i,j} ||\phi(\bar{\mathcal{S}}_i, \bar{\mathcal{A}}_i) - \phi(\bar{\mathcal{S}}_j, \bar{\mathcal{A}}_j)||^2 \tag{43}$$

In Equation 43, d and e are the list of states and corresponding neighbors respectively. In addition, $\mathfrak{A}$ is the adjacency matrix. Getting ϕ for a given state and action is at the output layer Opt of CNN for VIN-RS. Compared to [16], our loss function considers both the states and actions for activating the look-ahead advice mechanism.

6.2.4 Look-Ahead Advice

The Look-ahead advice mechanism proposed in [24] suggests considering the action as part of the potential-based reward shaping function. Advising on specific actions is a more rigorous method taken at the level of actions instead of being general for the whole state. The shaping function produced by CNN in VIN-RS after applying the look-ahead advice takes the following form:

$$F(\bar{s}, \bar{a}, \bar{s}', \bar{a}') = \gamma\phi(\bar{s}', \bar{a}') - \phi(\bar{s}, \bar{a}) \tag{44}$$

where ϕ is the potential shaping function that considers states from $\mathcal{S}$ and actions from $\mathcal{A}$ to result in a scalar value. Hence, the updated shaping function considering the action taken becomes:

$$R(s, a, s', a') = r(s, a) + F(s, a, s', a') \tag{45}$$

By augmenting the action values, the shaping function could potentially speed the learning speed further. Therefore, we propose adding the look-ahead advice in the design of VIN-RS, which is naturally embedded at the $\bar{\mathcal{Q}}$ layer.

6.2.5 Training RL with VIN-RS

After training the CNN of VIN-RS, the resulting shaping values for each state and action are passed to the RL algorithm for training and policy update. In Algorithm 17, we show the steps followed to train RL and benefit from the shaping value to obtain the $\mathcal{Q}_{comb}$, which is a combination of the original $\mathcal{Q}$ value and the one obtained using Equation 6 with ϕ from CNN as the shaping value. The algorithm starts by initializing the CNN and RL networks, as well as an empty graph $\mathcal{G}$ to hold the list of transitions. In each epoch of training, for a number of iterations T (for each trajectory followed by the agent), images are stored to later train the CNN. In addition, the list of transitions are stored in graph $\mathcal{G}$ to

later compute the loss of CNN. For every N episodes, the CNN is trained with the sampled images and graphs of transitions. Training the CNN every N episodes is more efficient and reduces the runtime of using VIN-RS in combination with the RL solution to speed learning. The loss is computed using Equation 41 and the CNN is trained for $\mathcal{K}$ iterations following the steps of Algorithm 5. Noting that if images are not available for training the CNN, the graph $\mathcal{G}$ can be used instead.

The combined value function with reward shaping is expressed as [16]:

$$\begin{aligned} &\mathcal{Q}^{\pi}_{comb}(s,a) = \alpha\mathcal{Q}^{\pi}(s,a) + (1-\alpha)\bar{\mathcal{Q}}^{\bar{\pi}}_{\phi}(s,a) \\ &\text{where } \bar{\mathcal{Q}}^{\bar{\pi}}_{\phi}(s,a) = \mathbf{E}_{(s,a)}[\sum_{t} \gamma^{t} r(S_t, A_t) + \gamma\phi(S_{t+1}, \\ &A_{t+1}) - \phi(S_t, A_t)] \end{aligned} \tag{46}$$

$\phi(S_t, A_t)$ is the shaping value at the output layer Opt of the CNN for state S_t and action A_t. Moreover, α is a hyperparameter the amount of the reward shaping value considered for updating the state-action value function. At the end of an epoch, the graph $\mathcal{G}$ can be emptied.

6.3 Experiments

In this section, the evaluation consists of experiments on two environments with discrete and continuous state and control. We use twenty Atari 2600 games from the Gym environment and four games from MuJoCo. In order to analyse the performance of VIN-RS and illustrate its advantage, we compare with the Proximal Policy Optimization (PPO) [84]; using GCN (denoted as ϕ_{GCN}) as the shaping function with the graph Laplacian as the filter [16]; LIRPG [62]; RND [63]; and ICM [64]. Details about the implementation and machines used are provided in the next subsection. First, we analyze the time complexity

Algorithm 6: Training RL with VIN-RS

```
Create the CNN network for VIN-RS;
Create empty graph 𝒢;
Create the RL networks;
for Episode=0,1,2, ... do
    for t = 1, 2, ..., T do
        Store images of all transitions;
        Perform the best action based on π;
        Get the state and reward from the environment;
        Add the transition to graph 𝒢;
    if mod(Episode, N) then
        Pass the list of images to CNN;
        Compute the loss for CNN using Equation 41;
        Train CNN following Algorithm 5 for 𝒦 iterations;
    Obtain ϕ for the list of states and actions;
    Q^π_comb = α Q^π + (1 − α) Q̄^π̄_ϕ;
    Train RL networks by updating the policy to maximize
      E_π[∇ log π(A_t|S_t) Q^π_comb(S_t, A_t)];
    Reset 𝒢 to empty graph (optional);
```

of VIN-RS compared to PPO and GCN. Afterward, we evaluate the performance of VIN-RS in different games for the Atari 2600 [97], and MuJoCo [96] environments compared to various baselines.

6.3.1 Implementation and Setup

The source code is written in the Python programming language. In our implementation, we used the PyTorch library to build our VIN-RS and combine it with the implementation of the Actor Critic (A2C) and Proximal Policy Optimization (PPO) algorithm. We also utilize the OpanAi Gym [97] and MuJoCo [96] libraries to simulate the environments of all the games. Images are passed to CNN to train in both the Atari and MuJoCo environments. The state representation of the MuJoCo games contains additional information about the state and not only the raw pixels. Therefore, we utilized the camera option in the MuJoCo

package to build the CNN input. In case there is no option to get images from the environment, a graph of states can be used as input to CNN. The same graph of states is used to compute the loss function, through message passing, when training the CNN. Passing images as input results in more information about the environment, which further improves the performance of CNN. As described in Section 6.2, the look-ahead advice mechanism is naturally embedded within the VIN-RS design and implementation, which offers a level of advantage for the proposed scheme compared to existing baselines. A link to our source code will be provided upon request.

For each run, a single GPU (NVIDIA V100 Volta (16GB HBM2 memory)) and eight CPUs (Intel E5-2683 v4 Broadwell @ 2.1GHz) were used with 32 GB of RAM. Details about the network configuration of each environment are provided in Sections 6.3.3 and 6.3.3.

6.3.2 Complexity

In this section, we show the results of our analysis in terms of runtime of the proposed VIN-RS when combined with PPO compared to GCN and vanilla PPO. In Table 6.1, we show the number of frames processed per second (FPS) using each of the solutions. As presented in Algorithm 5, the CNN of VIN-RS is only trained every $N = 1000$ episodes, that's why the FPS is very close compared to the other solutions. Therefore, when comparing the performance of VIN-RS to the other baselines, it is enough to compare the cumulative steps to converge or the average reward achieved over the number of iterations.

Method	FPS
PPO	1126
GCN	1054
VIN-RS	1051

Table 6.1: Frame Per Second (FPS) evaluated on Atari 2600

Training VIN-RS consumes additional time every couple of episodes due to the added

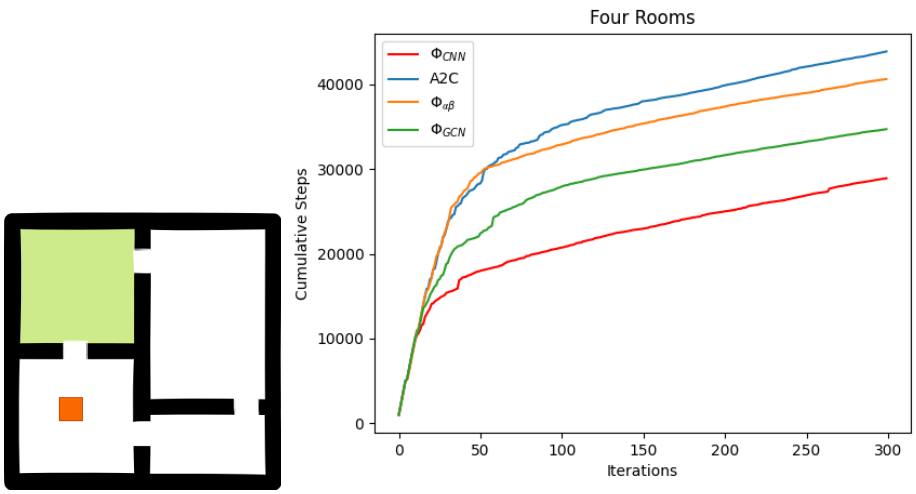
computation of the loss using Equation 41, in addition to the steps of training CNN following Algorithm 5. Therefore, VIN-RS has comparable execution time compared to PPO and GCN, thus not affecting the speed of learning. Therefore, studying the number of iterations to convergence has the same effect as measuring the time in seconds when comparing to different baselines in the following subsections. Besides, the implemented CNN network architecture contains three 2D Convolutional layers. The first layer considers the input size as the input channel extracted from the environment snapshot. The second layer uses 32 as the input channel. Finally, the third layer uses the size of the action space as the channel size to compute the Q-value. As shown in Section 6.3.3, this architecture is enough for achieving improved performance when using VIN-RS as a reward shaping solution in the tabular, Atari and MuJoCo environments.

6.3.3 Performance Analysis

In our evaluations, we consider three different environments to test the performance of the proposed VIN-RS. First is the Tabular with the four rooms game, second is the Atari 2600, and third is the MuJoCo. All these environments are similar to the evaluation criteria followed in [16] that proposes the use of GCN to perform message passing and predict the shaping value. In addition, it is important to study the performance on tabular, discrete, and continuous action spaces. In discrete action space, it is easier for the agent to perform and explore the finite set of discrete actions. Henceforth, the decision of deciding upon the nature of the action space (discrete or continuous) can directly affect the performance of the RL algorithm. Even-though the ϕ_{CNN} (using VIN-RS) approach for reward shaping achieves considerable improvement over the PPO algorithm in [16], we still provide a comparison with A2C and PPO as baselines.

Tabular Learning

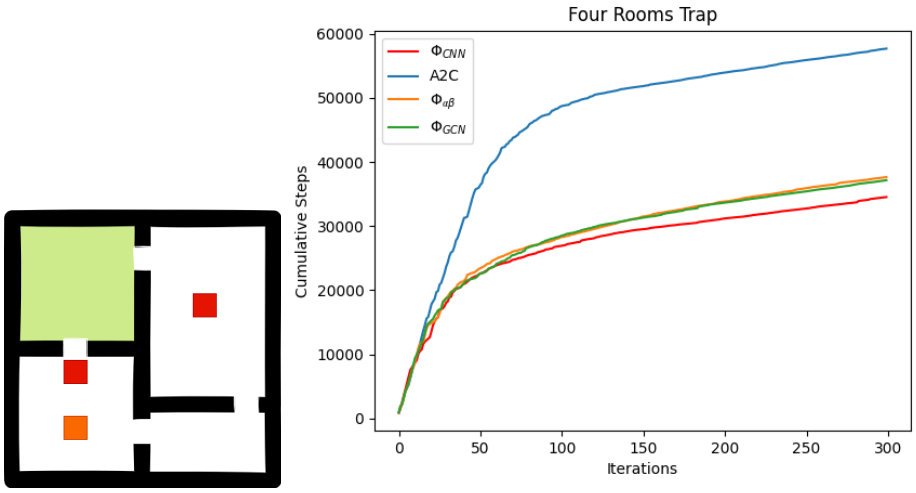
In this experiment, we present two setups of the Four Rooms game to evaluate the performance of VIN-RS ϕ_{CNN} that uses A2C. The analysis conducted on VIN-RS is compared to A2C, ϕ_{GCN}, and $\phi_{\alpha\beta}$. In $\phi_{\alpha\beta}$, message passing is computed due to the small environment spaces. Furthermore, λ-return is used as the critic part of A2C. A tabular RL solution is enough for the four rooms game shown in Figure 6.4a. In such an environment, it is possible to compute the actual message passing value $\phi_{\alpha\beta}$. However, in larger environment sizes and dimensions, it is not feasible to compute those messages. The two games we evaluate are the Four Rooms (Figure 6.4a) and its variant Four Rooms Traps (Figure 6.5a), where negative rewards are scattered across the four rooms as traps. Moreover, the exploration rate is maintained by setting the probability of random action selection to 0.1, thus allowing the agent to explore new series of actions. The results showing the cumulative steps are presented in Figures 6.4b and 6.5b. As shown in Figure 6.4b, ϕ_{CNN} has faster convergence



(a) Four Rooms **(b)** Convergence Speed

Figure 6.4: Cumulative steps over the number of iterations in Four Rooms

speed indicated by less number of cumulative steps compared to A2C, $\phi_{\alpha\beta}$, and ϕ_{GCN}. In



(a) Four Rooms Traps **(b)** Convergence speed

Figure 6.5: Cumulative steps over the number of iterations in Four Rooms Traps

particular, after 300 episodes, VIN-RS, helps the agent plan the optimum trajectory by considering long-term dependencies and making well-informed decisions benefiting from the message passing technique of HMM utilized by CNN. This approach leads to more efficient learning and faster convergence to an optimal policy with fewer training iterations. The planning factor is not possible when using message passing through GCN as the potential function, because (1) a subset of the states is only considered when training, and (2) value iteration is not performed to ensure planning. Similarly, the proposed ϕ_{CNN} solution is able to converge in fewer number of steps compared to the benchmarks for the four rooms traps, as shown in Figure 6.5b. Henceforth, we can observe that ϕ_{CNN} is performing better than A2C, $\phi_{\alpha\beta}$, and ϕ_{GCN} in the tabular settings.

Atari 2600

The Gym library [97] offers environments for twenty different Atari 2600 games. The main property of these games is that the action space is discrete. In this section, we evaluate the performance of our proposed solution ϕ_{CNN} in each of the games compared to four

baselines, which are PPO [84], ϕ_{GCN} [16], LIRPG [62], RND [63], and ICM [64]. In terms of VIN-RS implementation, we use the states which are raw pixel representation as input to the CNN. Furthermore, the number of channels at the $\bar{Q}$ layer is equal to the number of actions of the game.

We experiment with the twenty different Atari 2600 games using ϕ_{CNN}, ϕ_{GCN}, LIRPG, RND, ICM, and PPO. We execute each algorithm on every game for ten million steps, the same parameters are used as shown in Table 6.2.

Table 6.2: VIN-RS and RL configuration for the Atari 2600 games

Hyperparameter	Value
Learning rate	2.5e-4
γ	0.99
λ	0.95
Entropy Coefficient	0.01
PPO steps	128
PPO Clipping Value	0.1
# of minibatches	4
# of processes	8
CNN: α	0.9
CNN: η	1e1
GCN: α	0.9
GCN: η	1e1

In Figure 6.6a, we show the improvement achieved by ϕ_{CNN} using VIN-RS over PPO. In addition, we present the improvement of each of the baselines ϕ_{GCN}, LIRPG, RND, and ICM compared to PPO in Figures 6.6b, 6.6c, 6.6d, and 6.6e respectively. The results in these figures are shown in logarithmic scale. The mean difference between each solution

and PPO is computed, then a log is applied.

As shown in Figure 6.6a, the proposed ϕ_{CNN} is able to improve the performance of 13 different Atari games compared to PPO as a baseline. In the games of Venture, Qbert, UpNDown, MsPacman, and SpaceInvaders, ϕ_{CNN} is not performing well compared to PPO. In contrast, the performance of ϕ_{GCN} in these game is better compared to PPO. On the other hand, in most of the other games, ϕ_{CNN} is outperforming ϕ_{GCN} and improving over PPO. These information are extracted from Figures 6.6a and 6.6b. Furthermore, the proposed ϕ_{CNN} has the best results in terms of the number of games where the performance is better than PPO compared to the rest of the baselines. More specifically, the number of games with an improvement of PPO are classified as ϕ_{CNN} : 13, ϕ_{GCN} : 9, LIRPG: 11, RND: 7, ICM: 12.

In Figure 6.7, we present in more details the performance of ϕ_{CNN} compared to ϕ_{GCN} and PPO as baselines. In particular, we present a separate graph per Atari game showing the evolvement of the reward score over the number of learning steps, with a total of 10 million.

These results highlight the capabilities of performing value iteration or message passing using the proposed CNN architecture. Compared to ϕ_{CNN}, LIRPG does not guarantee invariance for the optimal policy, thus it is not potential-based. On the other hand, RND and ICM provide reward shaping through exploration and can only support discrete action spaces. In the sequel, we analyse the performance of ϕ_{CNN} compared to the baselines by comparing the games in every row of Figure 6.7. In Figures 6.7a, 6.7b, 6.7c, and 6.7d, we can see that the performance of ϕ_{CNN} is better compared to ϕ_{GCN} and PPO for the Alien and Frostbite games, while it is comparable to them in the Assault and Asteroids games. In the Alien and Frostbite games, the proposed ϕ_{GCN} is better than the other benchmarks due to its underlying effective planning behavior using the VIN mechanism to update the

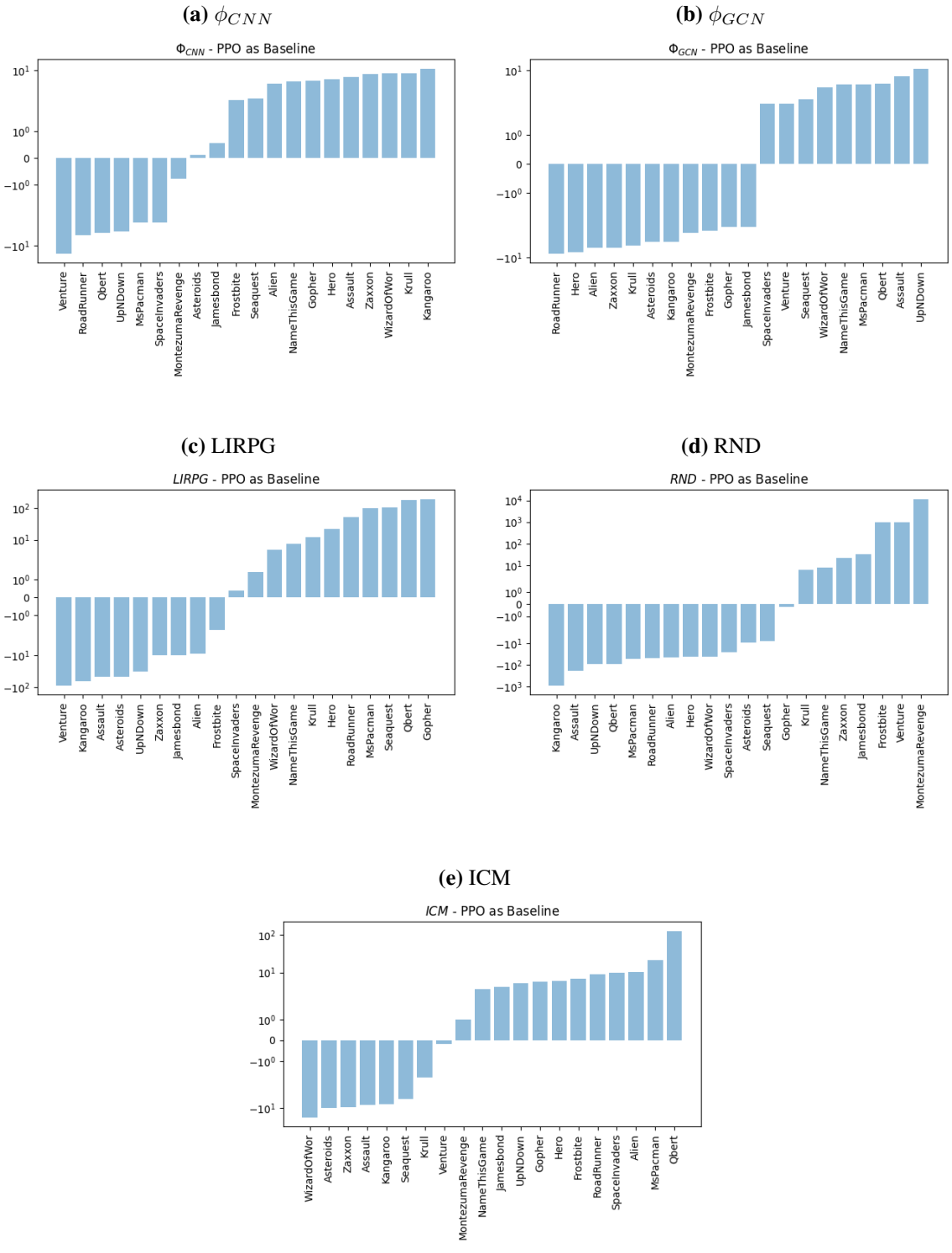


Figure 6.6: Performance comparison of each learning and reward shaping algorithm on Atari games in log scale over ppo

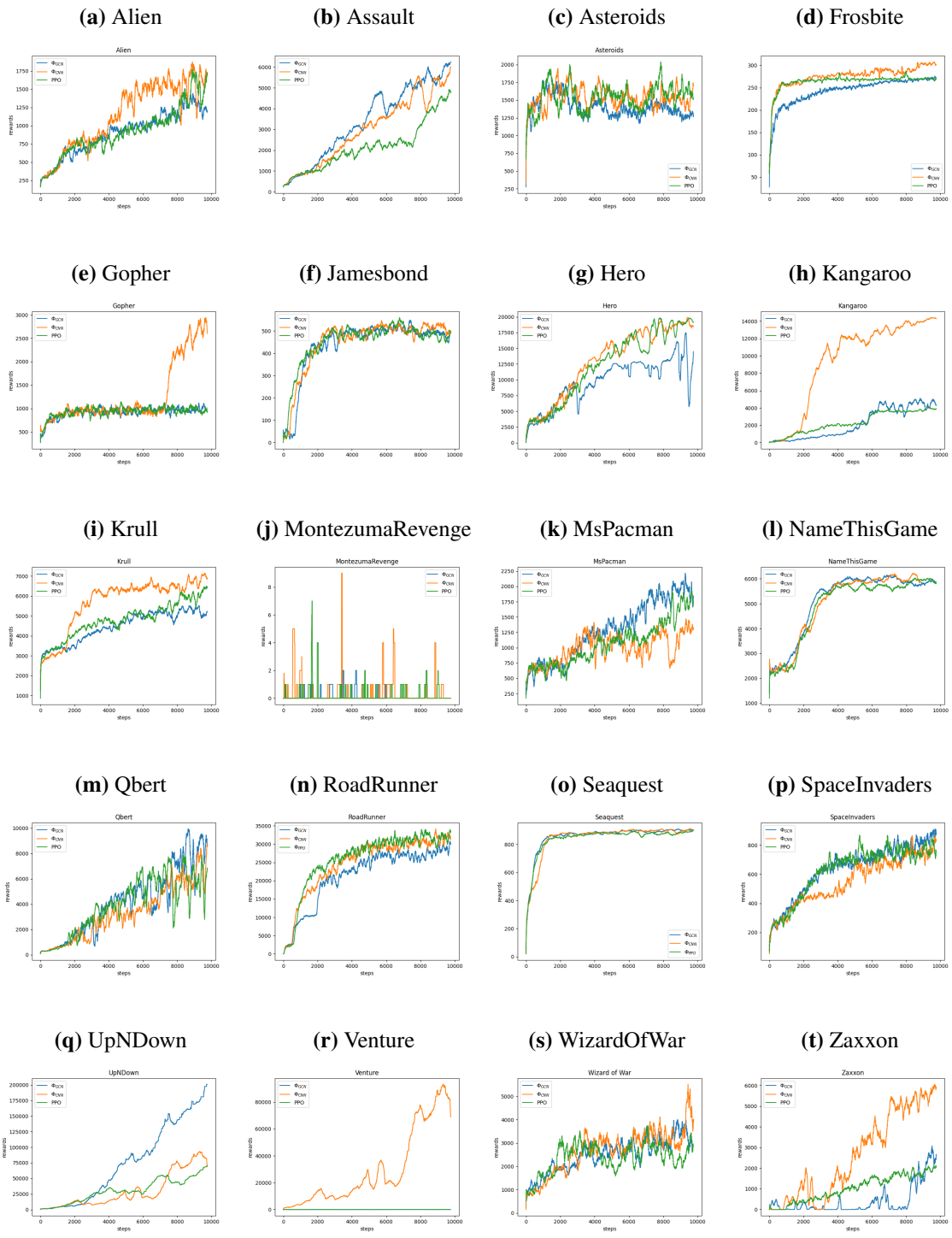


Figure 6.7: Results on 20 Atari games comapring the performance of ϕ_{CNN} to ϕ_{GCN} and PPO. The x-axis shows the number of steps$\times 10^3$.

shaping network. In particular, Alien is more challenging when it comes to obtaining the optimal behavior due to the need of exploring the environment for finding the hidden objects. Similarly, the frostbite game requires the agent to learn to jump over the obstacles and avoid falling into the water. In contrast, the Assault and Asteroids games required more straightforward behaviors that can be easily expressed without the need for a reward shaping function, which might add unnecessary complexity. This analysis explains why sometimes ϕ_{CNN} is either performing similar or worse compared to PPO. Similar to our previous analysis, the proposed ϕ_{CNN} performs better than existing baselines in the Gopher and Kangaroo games of Figures 6.7e and 6.7h due to the need for a reward shaping solution that can help the agent in collecting items in the case of Gopher, while jumping over obstacles to avoid enemies in the Kangaroo game. However, only a simple reward signal is sufficient for the Jamesbond and Hero games in Figures 6.7f and 6.7g, thus performing very closely to PPO or ϕ_{GCN}. The same concept applies for the rest of the games shown from Figure 6.7i to Figure 6.7t.

In certain Atari games, the behavior differences between ϕ_{CNN}, ϕ_{GCN}, and PPO can be explained by the games' complexity and the unique characteristics of each algorithm. For Hero, where a simple reward signal is sufficient, ϕ_{CNN} and PPO perform similarly, while ϕ_{GCN} shows fluctuating behavior due to potential limitations in its graph-based approach. In games like Kangaroo and Zaxxon, ϕ_{CNN} outperforms the other solutions, as its value iteration approach enables better planning and decision-making in more complex environments, resulting in faster learning and higher rewards at early stages.

In summary, ϕ_{CNN} achieves an average increase of approximately 20% in learning speed and higher rewards at early stages of learning in 9 out of 20 games.

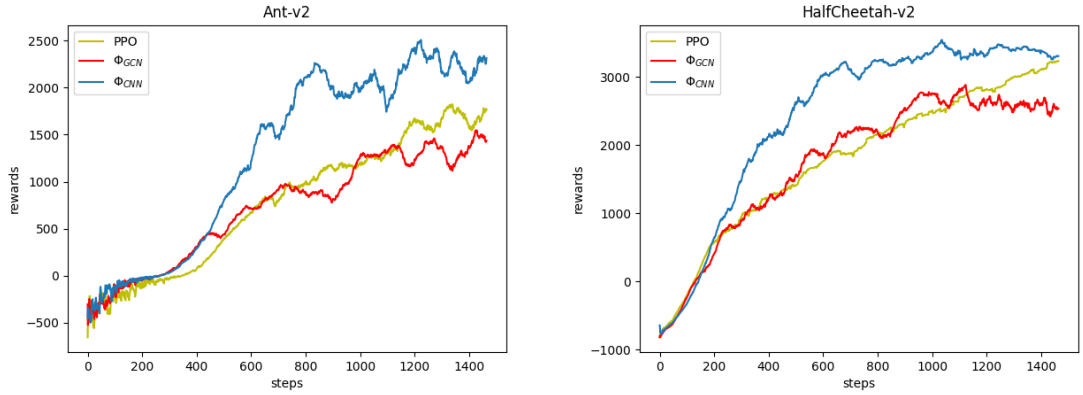
Mujoco

We evaluate the performance of VIN-RS in continuous action space on the MuJoco games compared to baselines. In the CNN implementation of VIN-RS, we deal with the continuous action space by discretizing that space. The values of two actions at the output layer resemble the range of the continuous actions in $\mathcal{M}$. The rest of the implementation is similar to the Atari games. In terms of baselines comparison, we compare with PPO and ϕ_{GCN}. It is not possible to compare with RND and ICM because these solutions do not support continuous control. We only compare with ϕ_{GCN} as a reward shaping solution because it is the closest to our work in terms of potential-based solution and the use of message passing. The results comparing the performance of each game to the other baselines are shown in Figure 6.8. The different settings related to this experiment are provided in Table 6.3. We use the same parameters as in [16] for evaluating the different techniques and baselines for fair comparison. The experiments were executed for three million steps for each game. We performed the experiment for each solution on each game for five times. The results in Figure 6.8 show the mean for each of the games in terms of rewards with respect to the number of steps in the environment.

As shown in the results of Figure 6.8, the proposed ϕ_{CNN} solution using VIN-RS out-

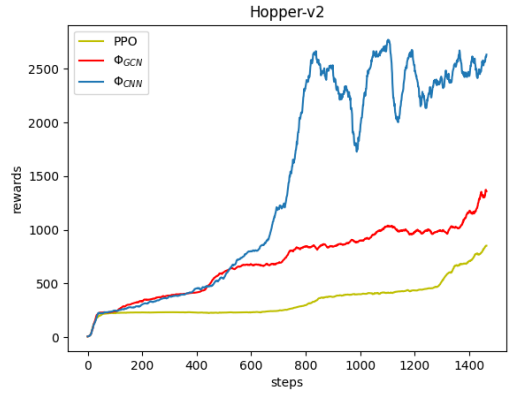
Table 6.3: VIN-RS and RL configurations for the MuJoCo games

Hyperparameter	Value
Learning rate	3e-4
γ	0.99
λ	0.95
Entropy Coefficient	0.0
PPO steps	2048
PPO Clipping Value	0.1
# of minibatches	32
# of processes	1
CNN/GCN (Walker and Ant): α	0.6
CNN/GCN (Hopper and HalfCheetah): α	0.6
CNN/GCN: η	1e1

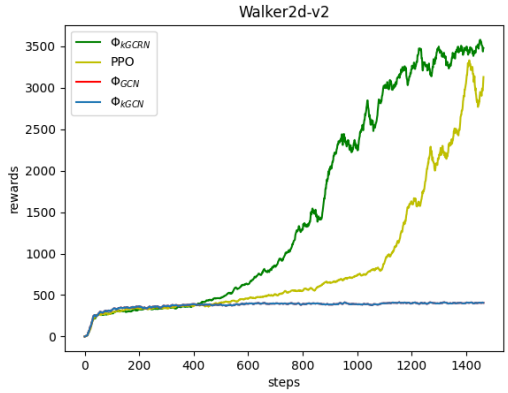


(a) Ant

(b) HalfCheetah



(c) Hopper



(d) Walker2d

Figure 6.8: Performance comparison between different reward shaping mechanisms and PPO in Mujoco environments.

performs PPO and ϕ_{GCN} in all the MuJoCo games. This improvement is measured by the convergence speed and the ability to reach high rewards at early stages of learning. In addition, the planning capability of ϕ_{CNN} is reflected in the performance by reaching high rewards that are not observed by the other solutions. Starting with the Ant game, the performance of PPO and ϕ_{GCN} is close, while reaching a maximum reward of around 1700 after 1400 steps of learning. Using the proposed ϕ_{CNN}, the RL agent is able to reach the 1700 reward score following 700 steps, which is half the number of steps needed by PPO and ϕ_{GCN}. Afterward, the score of 2500 was reached at around 1200 steps. After 1400

steps, we can observe that ϕ_{CNN} is able to converge to a score ranging between 2000 and 2300. This performance illustrates the ability of ϕ_{CNN} to reach higher rewards at early stages and converge faster compared to the benchmarks. The second MujoCo game we study is the HalfCheetah. In this game, the performance of ϕ_{GCN} is better than PPO during the first stages of learning up to 1000 steps. Following the 1000 steps, the performance of ϕ_{GCN} degrades on average to converge to around 2500 as reward score. On the other hand, ϕ_{CNN} can is able to reach higher rewards compared to PPO and ϕ_{GCN}, reaching around 3700 as reward score at the 1000 steps of learning. Furthermore, ϕ_{CNN} further converges at the 1000 steps, leading to the same conclusion for faster convergence and higher rewards. In the results of the Hopper game of Figure 6.8c, ϕ_{CNN} reaches higher rewards, which is at least two times better than ϕ_{GCN} and PPO at early stages, with a reward score exceeding 2500 at 800 steps. On the other hand, PPO and ϕ_{GCN} do not exceed 1200 as the reward score after 1400 steps. Finally, the fourth MuJoCo game Walker ensures that the performance of ϕ_{CNN} is better than PPO and ϕ_{GCN} in terms of convergence speed while reaching higher rewards at early stages. Specifically, PPO reaches 3000 as the reward score at the 1400 step, while ϕ_{GCN} can only reach around 2000 score at the 1400 step. However, ϕ_{CNN} is able to exceed 3000 steps after 1000 steps. In summary, ϕ_{CNN} is able to achieve approximately 30% increase in learning speed on average over PPO and ϕ_{GCN} in the four MuJoCo games.

6.4 Discussion

Our VIN-RS results show a substantial improvement over previous work's limitations in potential-based reward shaping. By leveraging message passing and CNN, VIN-RS enhances learning speed and decision quality in various MDP environments, especially in dynamically changing and time-sensitive domains like autonomous driving [15], resource

management [88, 98], health applications, Blockchain [86, 99, 100], and financial applications [101]. This integration also enables more efficient learning and faster convergence to an optimal policy, addressing challenges faced by traditional potential-based reward shaping methods in complex environments with continuous action spaces. Overall, VIN-RS offers a promising and flexible approach with significant potential for real-world applications.

The results provide promising evidence of its effectiveness. However, we recognize certain limitations that warrant consideration. Specifically, further investigations on larger-scale environments and more complex action spaces are necessary to enhance the generalizability and robustness of our approach. Addressing these aspects will strengthen the overall understanding and applicability of VIN-RS in a wider range of real-world applications.

6.5 Conclusion

In this chapter, we propose VIN-RS, a potential-based reward shaping mechanism that employs a novel CNN architecture as a potential function. VIN-RS performs planning in the environment by implementing a value iteration functionality. The loss function of CNN for value iteration is computed using the message passing mechanism that embeds forward and backward messages, resulting in an effective potential function for reward shaping. The training of CNN is performed on a sample of transitions. Our solution embeds the look-ahead advice mechanism inside the design of CNN for VIN-RS. We then propose the use of CNN as the potential function to produce shaping values. The resulting shaping values from sampled trajectories at each action are passed to the RL algorithm to update the policy. In our evaluations, we showed that the complexity of combining VIN-RS with other RL solutions is minimal. Furthermore, we evaluated the performance of VIN-RS compared to other baselines in Tabular, Atari 2600, and MuJoCo environments.

In the Atari experiments, VIN-RS achieves significant improvements over PPO and

ϕ_{GCN} in several games, while demonstrating comparable performance in others. Moreover, the statistical analysis reveals that VIN-RS leads to an average increase of 20% in learning speed and cumulative reward at earlier stages of learning across 9 out of 20 Atari games. Furthermore, the proposed ϕ_{CNN} solution outperforms these benchmarks in four of the MuJoCo games with continuous action space with approximately 30% increase in learning speed. In summary, VIN-RS achieves state-of-the art results in various games. Besides, through planning, the CNN for reward shaping can converge faster and reach high rewards that are not observable by other solutions in some of the games which introduce additional complexity to compute an effective policy.

Chapter 7

Conclusion and Future Direction

7.1 Conclusion

In this thesis, we have explored the challenges and opportunities presented by the rapidly evolving landscape of computing resource management in the context of edge and fog computing. We have addressed the critical need for effective Intelligent Computing Resource Management (ICRM) solutions to ensure efficient resource allocation, service placement, and proactive scaling in dynamic and diverse application environments. Through a comprehensive literature review, we identified gaps in existing approaches and proposed a novel framework that leverages the power of Deep Reinforcement Learning (DRL) to overcome these limitations. We have also focused on addressing the problems of slow convergence speed of DRL though potential-based reward shaping solutions.

Our research contributions have been diverse and aimed at providing robust, adaptive, and efficient solutions to the complex resource management problems. We have designed

and developed the Intelligent Fog and Service Placement (IFSP) algorithm, which combines DRL with offline learning to enable proactive and intelligent service placement decisions. IFSP demonstrated superior performance compared to existing heuristic and DRL-based solutions, effectively adapting to changing user demands and achieving higher Quality of Service (QoS).

Following our IFSP contribution, we augmented the IFSP solution by proposing IScaler. In IScaler, we have addressed different limitations related to placement while accounting to changes in the available resources. Furthermore, IScaler is able to perform horizontal and vertical scaling to better utilize the computing resources based on the change in availabilities. In addition, we presented as part of this architecture the solution switch, which supports the DRL agent during the first stages of learning through the use of an Evolutionary Memetic algorithm solution. Through a series of simulations, we illustrated the ability of IScaler to outperform existing auto-scaling solutions for intelligent and heuristic-based computing resource management.

To address the challenge of slow learning in DRL algorithms, we introduced the Graph Convolutional Recurrent Network (GCRN) and the Value Iteration Network for Reward Shaping (VIN-RS). These innovations provided significant improvements in convergence speed and decision-making quality, showcasing the potential of combining reinforcement learning with novel reward shaping techniques. In the case of GCRN, we presented a shaping function that combines GCN and RNN with the use of message passing to train the network. On the other hand, we presented the use of the VIN network architecture as the shaping function. The VIN was also trained using the message passing algorithm and takes an image as input. Both potential-based reward shaping approaches were tested on tabular, Atari, and MuJoCo games. Through our simulations, we have presented superiority of each of the solutions compared to state of the art work in terms of convergence speed and reaching higher rewards at early stages of learning.

7.2 Future Directions

Throughout this thesis, we have showcased the effectiveness of our proposed solutions through rigorous experiments and evaluations on real-world datasets, emphasizing their potential to transform the field of computing resource management. However, it is important to acknowledge that the journey of intelligent computing resource management is ongoing. The field continues to evolve, and new challenges and opportunities are likely to emerge. Our work opens opportunities for further research and development, including the exploration of hybrid solutions that combine the strengths of DRL for resource management and reward shaping for form more sophisticated learning algorithms, in addition to the integration of AI-driven resource management into diverse application domains.

In summary, this thesis has made significant contributions in advancing the field of Intelligent Computing Resource Management for edge and fog computing. By harnessing the capabilities of Deep Reinforcement Learning, we have contributed novel solutions that offer adaptability, intelligence, and efficiency in resource allocation and service placement. Furthermore, the reward shaping techniques presented have opened new opportunities for DRL solutions to be applied for solving time-sensitive real-work problems, such as computing resource management. As technology continues to evolve and the demands of modern applications grow, our research provides a foundation upon which future innovations can build to ensure optimal computing resource utilization and improved user experiences and quality of service across various domains. We summarize in the following list the main persisting research gaps that we believe are worth investigating in the future:

- **Hybrid DRL and Reward Shaping Techniques:** There is still no work done in the literature that combines and effective DRL-based solution with reward shaping and a complete framework. In Figure 7.1, we present a promising architecture where we combine our state-of-the-art ICRM solution with both GCRN and VIN-RS algorithms for reward shaping. In this figure, we elaborate an augmented ICRM solution

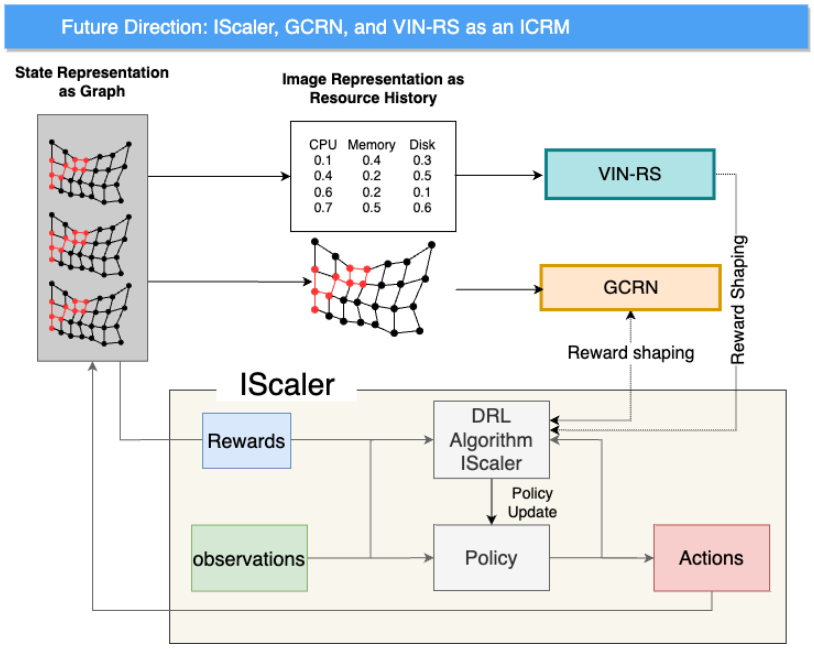


Figure 7.1: Combining IScaler, GCRN, and VIN-RS to build and ICRM.

that combines DRL and two effective reward shaping solutions. The first reward shaping based on GCRN measures the dependencies between the services to provide a better understanding of the current placement and demands. The second uses VIN-RS which studies the grid representation of the historical resource availability and utilization for each host.

Collectively, this augmented ICRM solution showcases a sophisticated approach to fog/edge resource management. By seamlessly integrating DRL with the GCRN and VIN-RS reward shaping techniques, the system offers an intelligent, holistic, and adaptable means of allocating resources. This ultimately results in enhanced operational efficiency, improved service delivery, and maximized utilization of cloud resources in a dynamically changing computing environment.

- **AI-Driven Resource Management Integration:** Extending the application of AI-driven resource management beyond edge and fog computing opens avenues for innovation. Investigating how these techniques can be seamlessly integrated into cloud

computing, data centers, and Internet of Things (IoT) networks provides an opportunity to tailor existing algorithms to diverse application domains. Adapting and customizing resource management strategies to accommodate domain-specific constraints and requirements is pivotal to enhancing resource efficiency across a wide spectrum of real-world scenarios.

- **Real-World Deployment and Evaluation:** To validate the practicality and effectiveness of the proposed solutions, extensive real-world deployment and evaluation are essential. Thoroughly testing the Intelligent Fog and Service Placement (IFSP) algorithm and IScaler in varied application scenarios and dynamic environments can shed light on their scalability, reliability, and feasibility for large-scale distributed systems. This empirical exploration helps bridge the gap between theoretical advancements and their tangible impact in improving resource utilization and user experiences.

- **Adaptive Learning and Transferability:** A promising direction involves developing mechanisms that enable DRL agents to swiftly adapt to evolving environments, dynamic application demands, and fluctuating resource availability. The adaptable learning strategies empowers agents to maintain optimal resource allocation even through unpredictable changes. Additionally, investigating techniques to transfer learned policies or knowledge across diverse resource management contexts has the potential to expedite learning and enhance the agents' ability to generalize insights.

- **Exploration of Other Deep Learning Architectures:** Beyond Graph Convolutional Recurrent Networks (GCRN) and Value Iteration Networks (VIN), exploring alternative deep learning architectures for reward shaping and resource management introduces exciting possibilities. Delving into the utility of architectures

like Transformers, Variational Autoencoders (VAEs), or Generative Adversarial Networks (GANs) holds the potential to uncover novel ways of enhancing learning efficiency and decision-making quality in resource management tasks.

Bibliography

[1] Hani Sami and Azzam Mourad. Dynamic on-demand fog formation offering on-the-fly iot service deployment. *IEEE Transactions on Network and Service Management*, 2020.

[2] Fabio López-Pires and Benjamín Barán. Many-objective virtual machine placement. *Journal of Grid Computing*, 15(2):161–176, 2017.

[3] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. *Big data and internet of things: A roadmap for smart environments*, pages 169–186, 2014.

[4] Hani Sami and Azzam Mourad. Towards dynamic on-demand fog computing formation based on containerization technology. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 960–965. IEEE, 2018.

[5] Gigi Sayfan. *Mastering Kubernetes*. Packt Publishing Ltd, 2017.

[6] Peter Farhat, Hani Sami, and Azzam Mourad. Reinforcement r-learning model for time scheduling of on-demand fog placement. *The Journal of Supercomputing*, 76 (1):388–410, 2020.

[7] David M Gutierrez-Estevez, Marco Gramaglia, Antonio De Domenico, Nicola Di Pietro, Sina Khatibi, Kunjan Shah, Dimitris Tsolkas, Paul Arnold, and Pablo

Serrano. The path towards resource elasticity for 5g network architecture. In *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 214–219. IEEE, 2018.

[8] Deepak Vohra. *Kubernetes microservices with Docker*. Apress, 2016.

[9] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications*, 2020.

[10] JV Bibal Benifa and D Dejey. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications*, 24(4):1348–1363, 2019.

[11] Nouha Kherraf, Hyame Assem Alameddine, Sanaa Sharafeddine, Chadi M Assi, and Ali Ghrayeb. Optimized provisioning of edge computing resources with heterogeneous workload in iot networks. *IEEE Transactions on Network and Service Management*, 16(2):459–474, 2019.

[12] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[13] Khaled B Letaief, Wei Chen, Yuanming Shi, Jun Zhang, and Ying-Jun Angela Zhang. The roadmap to 6g: Ai empowered wireless networks. *IEEE Communications Magazine*, 57(8):84–90, 2019.

[14] Ivan Stojmenovic and Sheng Wen. The fog computing paradigm: Scenarios and security issues. In *2014 federated conference on computer science and information systems*, pages 1–8. IEEE, 2014.

[15] Hani Sami, Azzam Mourad, and Wassim El-Hajj. Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services. *IEEE/ACM Transactions on Networking*, 28(2):778–790, 2020.

[16] Martin Klissarov and Doina Precup. Reward propagation using graph convolutional networks. In *NeurIPS*, 2020.

[17] Marc Toussaint and Amos Storkey. Probabilistic inference for solving discrete and continuous state markov decision processes. In *ICML*, pages 945–952, 2006.

[18] Brian D. Ziebart, Andrew L. Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, pages 1433–1438. AAAI Press, 2008.

[19] Lawrence Rabiner and Biinghwang Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, 3(1):4–16, 1986.

[20] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[21] Fan RK Chung. *Spectral graph theory*, volume 92. American Mathematical Soc., 1997.

[22] Sridhar Mahadevan and Mauro Maggioni. Value function approximation with diffusion wavelets and Laplacian eigenfunctions. *NeurIPS*, 18:843, 2006.

[23] Marek Petrik. An analysis of laplacian methods for value function approximation in mdps. In *IJCAI*, pages 2574–2579, 2007.

[24] Eric Wiewiora, Garrison W Cottrell, and Charles Elkan. Principled methods for advising reinforcement learning agents. In *ICML*, pages 792–799, 2003.

[25] Anna Harutyunyan, Tim Brys, Peter Vrancx, and Ann Nowé. Shaping mario with human advice. In *AAMAS*, pages 1913–1914, 2015.

[26] Marek Grześ and Daniel Kudenko. Online learning of shaping rewards in reinforcement learning. *Neural Networks*, 23(4):541–550, 2010.

[27] Ilse CF Ipsen and Carl D Meyer. The idea behind krylov methods. *The American mathematical monthly*, 105(10):889–899, 1998.

[28] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *NIPS*, pages 2146–2154, 2016.

[29] Min Chen, Wei Li, Giancarlo Fortino, Yixue Hao, Long Hu, and Iztok Humar. A dynamic service migration mechanism in edge cognitive computing. *ACM Transactions on Internet Technology (TOIT)*, 19(2):1–15, 2019.

[30] Zhiqing Tang, Xiaojie Zhou, Fuming Zhang, Weijia Jia, and Wei Zhao. Migration modeling and learning algorithms for containers in fog computing. *IEEE Transactions on Services Computing*, 12(5):712–725, 2018.

[31] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Elastic deployment of software containers in geo-distributed computing environments. In *Proc. of IEEE ISCC'19*, 2019.

[32] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. Towards qos-aware fog service placement. In *2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC)*, pages 89–96. IEEE, 2017.

[33] Hadi Goudarzi and Massoud Pedram. Energy-efficient virtual machine replication and placement in a cloud computing system. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 750–757. IEEE, 2012.

[34] Nicolò Maria Calcavecchia, Ofer Biran, Erez Hadad, and Yosef Moatti. Vm placement strategies for cloud scenarios. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 852–859. IEEE, 2012.

[35] Lei Zhao and Jiajia Liu. Optimal placement of virtual machines for supporting multiple applications in mobile edge networks. *IEEE Transactions on Vehicular Technology*, 67(7):6533–6545, 2018.

[36] Alberto Ceselli, Marco Premoli, and Stefano Secci. Mobile edge cloud network design optimization. *IEEE/ACM Transactions on Networking*, 25(3):1818–1831, 2017.

[37] Alireza Sadeghi, Fatemeh Sheikholeslami, and Georgios B Giannakis. Optimal and scalable caching for 5g using reinforcement learning of space-time popularities. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):180–190, 2017.

[38] Xiaosheng Lin, Yuhao Tang, Xianfu Lei, Junjuan Xia, Qingfeng Zhou, Huijun Wu, and Liseng Fan. Marl-based distributed cache placement for wireless networks. *IEEE Access*, 7:62606–62615, 2019.

[39] Yifei Wei, F Richard Yu, Mei Song, and Zhu Han. Joint optimization of caching, computing, and radio resources for fog-enabled iot using natural actor–critic deep reinforcement learning. *IEEE Internet of Things Journal*, 6(2):2061–2073, 2018.

[40] Yaohua Sun, Mugen Peng, and Shiwen Mao. Deep reinforcement learning-based mode selection and resource management for green fog radio access networks. *IEEE Internet of Things Journal*, 6(2):1960–1971, 2018.

[41] Ziad A Al-Sharif, Yaser Jararweh, Ahmad Al-Dahoud, and Luay M Alawneh. Accrs: autonomic based cloud computing resource scaling. *Cluster Computing*, 20(3): 2479–2488, 2017.

[42] Chunlin Li, Hezhi Sun, Yi Chen, and Youlong Luo. Edge cloud resource expansion and shrinkage based on workload for minimizing the cost. *Future Generation Computer Systems*, 101:327–340, 2019.

[43] Jitendra Kumar, Ashutosh Kumar Singh, and Rajkumar Buyya. Self directed learning based workload forecasting model for cloud resource management. *Information Sciences*, 543:345–366, 2020.

[44] In Kee Kim, Wei Wang, Yanjun Qi, and Marty Humphrey. Forecasting cloud application workloads with cloudinsight for predictive resource management. *IEEE Transactions on Cloud Computing*, 2020.

[45] Constantinos Bitsakos, Ioannis Konstantinou, and Nectarios Koziris. Derp: A deep reinforcement learning cloud system for elastic resource provisioning. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 21–29. IEEE, 2018.

[46] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi, and Giovani Estrada. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 64–73. IEEE, 2017.

[47] Michele Scarpiniti, Enzo Baccarelli, Paola G Vinueza Naranjo, and Aurelio Uncini. Energy performance of heuristics and meta-heuristics for real-time joint resource scaling and consolidation in virtualized networked data centers. *The Journal of Supercomputing*, 74(5):2161–2198, 2018.

[48] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. Applications of deep reinforcement learning

in communications and networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(4):3133–3174, 2019.

[49] Rongpeng Li, Zhifeng Zhao, Qi Sun, I Chih-Lin, Chenyang Yang, Xianfu Chen, Minjian Zhao, and Honggang Zhang. Deep reinforcement learning for resource management in network slicing. *IEEE Access*, 6:74429–74441, 2018.

[50] Helin Yang, Zehui Xiong, Jun Zhao, Dusit Niyato, Liang Xiao, and Qingqing Wu. Deep reinforcement learning based intelligent reflecting surface for secure wireless communications. *IEEE Transactions on Wireless Communications*, 2020.

[51] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1179–1191, 2020.

[52] Kerong Wang, Hanye Zhao, Xufang Luo, Kan Ren, Weinan Zhang, and Dongsheng Li. Bootstrapped transformer for offline reinforcement learning. In *Advances in Neural Information Processing Systems*, 2022.

[53] Chenjia Bai, Lingxiao Wang, Zhuoran Yang, Zhi-Hong Deng, Animesh Garg, Peng Liu, and Zhaoran Wang. Pessimistic bootstrapping for uncertainty-driven offline reinforcement learning. In *International Conference on Learning Representations*, 2021.

[54] Seunghyun Lee, Younggyo Seo, Kimin Lee, Pieter Abbeel, and Jinwoo Shin. Offline-to-online reinforcement learning via balanced replay and pessimistic q-ensemble. In *Conference on Robot Learning*, pages 1702–1712. PMLR, 2022.

[55] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[56] Omar Abu Arqub and Zaer Abo-Hammour. Numerical solution of systems of second-order boundary value problems using continuous genetic algorithm. *Information sciences*, 279:396–415, 2014.

[57] Zaer Abo-Hammour, Omar Abu Arqub, Shaher Momani, and Nabil Shawagfeh. Optimization solution of troesch's and bratu's problems of ordinary type using novel continuous genetic algorithm. *Discrete Dynamics in Nature and Society*, 2014, 2014.

[58] Sufeng Niu, Siheng Chen, Hanyu Guo, Colin Targonski, Melissa C Smith, and Jelena Kovačević. Generalized value iteration networks: Life beyond lattices. In *AAAI*, pages 6246–6253, 2018.

[59] Wei Li, Bowei Yang, Guanghua Song, and Xiaohong Jiang. Dynamic value iteration networks for the planning of rapidly changing UAV swarms. *Frontiers of Information Technology & Electronic Engineering*, pages 1–10, 2021.

[60] Shu Yang, Jinglin Li, Jie Wang, Zhihan Liu, and Fangchun Yang. Learning urban navigation via value iteration network. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 800–805, 2018.

[61] Ashok Kumar Khatta, Jagdeep Singh, and Gurjinder Kaur. Vehicle routing problem with value iteration network. In *Advanced Network Technologies and Intelligent Computing: Second International Conference, ANTIC 2022, Varanasi, India, December 22–24, 2022, Proceedings, Part I*, pages 3–15. Springer, 2023.

[62] Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. *NeurIPS*, pages 4649–4659, 2018.

[63] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

[64] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, pages 2778–2787, 2017.

[65] Hani Sami, Jamal Bentahar, Azzam Mourad, Hadi Otrok, and Ernesto Damiani. Graph convolutional recurrent networks for reward shaping in reinforcement learning. *Information Sciences*, 608:63–80, 2022.

[66] Nadia Moati, Hadi Otrok, Azzam Mourad, and Jean-Marc Robert. Reputation-based cooperative detection model of selfish nodes in cluster-based qos-olsr protocol. *Wireless personal communications*, 75(3):1747–1768, 2014.

[67] Sawsan Abdul Rahman, Azzam Mourad, May El Barachi, and Wael Al Orabi. A novel on-demand vehicular sensing framework for traffic condition monitoring. *Vehicular Communications*, 12:165–178, 2018.

[68] Ali A Abdallah, Samer S Saab, and Zaher M Kassas. A machine learning approach for localization in cellular environments. In *2018 IEEE/ION Position, Location and Navigation Symposium (PLANS)*, pages 1223–1227, 2018.

[69] Wissam Fawaz, Ribal Atallah, Chadi Assi, and Maurice Khabbaz. Unmanned aerial vehicles as store-carry-forward nodes for vehicular networks. *IEEE Access*, 5: 23710–23718, 2017.

[70] Wissam Fawaz. Effect of non-cooperative vehicles on path connectivity in vehicular networks: A theoretical analysis and uav-based remedy. *Vehicular Communications*, 11:12–19, 2018.

[71] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[72] Xin Xu, Lei Zuo, and Zhenhua Huang. Reinforcement learning algorithms with

function approximation: Recent advances and applications. *Information Sciences*, 261:1–31, 2014.

[73] Samer S Saab and Dong Shen. Multidimensional gains for stochastic approximation. *IEEE transactions on neural networks and learning systems*, 31(5):1602–1615, 2019.

[74] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[75] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[76] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, pages 1–14, 2011.

[77] Nasa dataset - two months of http logs from a busy www server, 1996. URL https://ita.ee.lbl.gov/html/contrib/NASA-HTTP.

[78] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[79] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[80] Junhyuk Oh, Matteo Hessel, Wojciech M Czarnecki, Zhongwen Xu, Hado P van

Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. In *NeurIPS*, 2020.

[81] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

[82] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.

[83] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[84] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[85] Michael E Wall, Andreas Rechtsteiner, and Luis M Rocha. Singular value decomposition and principal component analysis. In *A practical approach to microarray data analysis*, pages 91–109. Springer, 2003.

[86] Ahmad Hammoud, Hani Sami, Azzam Mourad, Hadi Otrok, Rabeb Mizouni, and Jamal Bentahar. AI, blockchain, and vehicular edge computing for smart and secure IoV: Challenges and directions. *IEEE Internet of Things Magazine*, 3(2):68–73, 2020.

[87] Gaith Rjoub, Omar Abdel Wahab, Jamal Bentahar, and Ahmed Saleh Bataineh. Improving autonomous vehicles safety in snow weather using federated YOLO CNN learning. In Jamal Bentahar, Irfan Awan, Muhammad Younas, and Tor-Morten Grønli, editors, *Mobile Web and Intelligent Information Systems - 17th International*

Conference, MobiWIS 2021, Virtual Event, August 23-25, 2021, Proceedings, volume 12814 of *Lecture Notes in Computer Science*, pages 121–134. Springer, 2021.

[88] Hani Sami, Hadi Otrok, Jamal Bentahar, and Azzam Mourad. AI-based resource provisioning of IoE services in 6G: A deep reinforcement learning approach. *IEEE Transactions on Network and Service Management*, 18(3):3527–3540, 2021.

[89] Gaith Rjoub, Jamal Bentahar, Omar Abdel Wahab, and Ahmed Saleh Bataineh. Deep and reinforcement learning for automated task scheduling in large-scale cloud computing systems. *Concurrency and Computation: Practice and Experience*, 33 (23), 2021.

[90] Gaith Rjoub, Omar Abdel Wahab, Jamal Bentahar, and Ahmed Saleh Bataineh. Trust-driven reinforcement selection strategy for federated learning on IoT devices. *Computing*, in press, 2022. doi: 10.1007/s00607-022-01078-1.

[91] Mohammed Shurrab, Shakti Singh, Rabeb Mizouni, and Hadi Otrok. Iot sensor selection for target localization: A reinforcement learning based approach. *Ad Hoc Networks*, 134:102927, 2022.

[92] Ahmed Alagha, Shakti Singh, Rabeb Mizouni, Jamal Bentahar, and Hadi Otrok. Target localization using multi-agent deep reinforcement learning with proximal policy optimization. *Future Generation Computer Systems*, 136:342–357, 2022.

[93] Luíza Caetano Garaffa, Maik Basso, Andréa Aparecida Konzen, and Edison Pignaton de Freitas. Reinforcement learning for mobile robotics exploration: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2021. doi: 10.1109/TNNLS.2021.3124466.

[94] Hani Sami, Reem Saado, Ahmad El Saoudi, Azzam Mourad, Hadi Otrok, and Jamal Bentahar. Opportunistic uav deployment for intelligent on-demand iov service management. *IEEE Transactions on Network and Service Management*, 2023.

[95] Gaith Rjoub, Jamal Bentahar, Omar Abdel Wahab, and Ahmed Saleh Bataineh. Deep and reinforcement learning for automated task scheduling in large-scale cloud computing systems. *Concurr. Comput. Pract. Exp.*, 33, 2021. doi: https://doi.org/10.1002/cpe.5919.

[96] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[97] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[98] Hani Sami, Azzam Mourad, Hadi Otrok, and Jamal Bentahar. Demand-driven deep reinforcement learning for scalable fog and service placement. *IEEE Transactions on Services Computing*, 2021.

[99] Hani Sami, Rabeb Mizouni, Hadi Otrok, Shakti Singh, Jamal Bentahar, and Azzam Mourad. Learnchain: Transparent and cooperative reinforcement learning on blockchain. *Future Generation Computer Systems*, 150:255–271, 2024.

[100] Maha Kadadha, Hadi Otrok, Rabeb Mizouni, Shakti Singh, and Anis Ouali. On-chain behavior prediction machine learning model for blockchain-based crowdsourcing. *Future Generation Computer Systems*, 2022.

[101] Avraam Tsantekidis, Nikolaos Passalis, Anastasia-Sotiria Toufa, Konstantinos

Saitas-Zarkias, Stergios Chairistanidis, and Anastasios Tefas. Price trailing for financial trading using deep reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*, 32(7):2837–2846, 2020.