TÉCNICO LISBOA

Task offloading optimization in Mobile Edge Computing based on Deep Reinforcement Learning

Carlos Alexandre Marques Alves da Silva

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. António Manuel Raminhos Cordeiro Grilo
Co-Supervisor: Prof. Naercio David Pedro Magaia

Examination Committee

Chairperson: Prof. José Eduardo Charters Ribeiro da Cunha Sanguino
Supervisor: Prof. António Manuel Raminhos Cordeiro Grilo
Co-Supervisor: Prof. Naercio David Pedro Magaia
Member of the Committee: Prof. Paolo Romano

June 2022

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Declaração

Declaro que o presente documento é um trabalho original da minha autoria e que cumpre todos os requisitos do Código de Conduta e Boas Práticas da Universidade de Lisboa.

Abstract

The Cloud Computing (CC) paradigm has risen in recent years as a solution to a need for computation and battery constrained User Equipment (UE) to run increasingly intensive computation tasks. Given its seemingly infinite amount of resources and pay-as-you-go nature, this paradigm has brought several advantages: 1) Extended battery life of User Equipment (UE)s by offloading computation; 2) Enables new types of applications intractable with User Equipment (UE)'s computation capabilities; 3) In an ever more data-focused world it allows for unlimited storage capacity. Nevertheless, given the centralized nature of the CC paradigm, this option introduces significant network congestion problems and unpredictable communication delays not suitable for real-time applications. In order to cope with these problems, the Mobile Edge Computing (MEC) concept has been introduced, which proposes to bring computation resources closer to the edge of the mobile networks in a distributed way. However, given that these edge computation resources are limited, this paradigm comes with its set of challenges that need to be solved in order to make it viable. In this work, a review of several MEC architectures is made. Subsequently, the state of the art on Deep Reinforcement Learning (DRL) and their application to the MEC challenges are explored. This work proposes to innovate by presenting a network management agent capable of making offloading decisions from a heterogeneous network of UEs to a heterogeneous network of MEC servers. This agent represents an orchestrator of a group of 5G Small Cells (SCeNBs), enhanced with computation and storage capabilities. In order to solve this high complexity problem, an Advantage Actor-Critic (A2C) agent is implemented and tested against several baselines. The proposed solution is shown to beat the baselines by making intelligent decisions taking into account computation, battery, delay and communication constraints ignored by the baselines. The solution is also shown to be scalable, data-efficient, robust, stable and adjustable to address not only overall system performance but to take into account the worst-case scenario.

Keywords: Mobile Edge Computing, Computation Offloading, Energy and Performance Optimization, Delay Sensitivity, Deep Reinforcement Learning, Cloud Computing

Resumo

Nos últimos anos o paradigma de Computação em Nuvem (CC) tem vindo a crescer como uma solução para a necessidade de correr computações cada vez mais complexas em Equipamentos de Utilizador (UE) limitados a nível de computação e bateria. Devido à quantidade aparentemente infinita de recursos computacionais e sua natureza de pagamento conforme o uso, este paradigma trouxe várias vantagens: 1) Ao transferir computação poupa a bateria dos UEs; 2) Permite novas aplicações complexas não executáveis com os recursos dos UEs; 3) Num mundo cada vez mais focado em dados, permite o armazenamento ilimitado dos mesmos. No entanto, dada a natureza centralizada do paradigma CC existem problemas de congestionamento de rede significativos e atrasos de comunicação imprevisíveis e inadequados para aplicações em tempo real. O conceito de *Mobile Edge Computing* (MEC) surgiu para lidar com estes problemas e tem como ideia principal aproximar os recursos de computação de forma distribuída ao limite das redes móveis. Por vez, como esses recursos de computação são limitados, este novo paradigma apresenta um conjunto de desafios que precisam de ser resolvidos de maneira a torná-lo viável. Neste trabalho é feita uma revisão de várias arquitecturas de MEC. Subsequentemente, o estado da arte em Aprendizagem por Reforço Profunda (DRL) e a sua aplicação aos desafios do paradigma MEC são explorados. Este trabalho propõe então inovar apresentando um agente de gestão de rede capaz de tomar decisões de offloading de uma rede heterogênea de UEs para uma rede heterogênea de servidores MEC. Este agente representa um orquestrador de um grupo de *Small Cells* (SCeNBs) de 5G, dotados de capacidades de computação e armazenamento. De maneria a resolver este problema de elevada complexidade foi implementado um agente *Advantage Action-Critic* (A2C) e este foi testado contra um grupo de algorimos base. A solução proposta demonstra capacidade de bater os algorimos base tomando decisões inteligentes que têm em conta limitações de computação, bateria, atraso e comunicação, ignoradas pelos algoritmos base. É também demonstrado que a solução é escalável, eficiente de um ponto de vista de dados, robusta, estável e ajustável para ter em consideração não só o custo do sistema como um todo mas ter também em conta o pior caso do mesmo.

Palavras Chave: *Mobile Edge Computing*, *Offloading* Computacional, Optimização Energética e de Execução, Restrições de Latência, Aprendizagem por Reforço Profunda, Computação em Nuvem

Contents

Acronyms

A2C Advantage Actor-Critic. iii, iv, 3, 14, 20, 27

A3C Asynchronous Advantage Actor-Critic. 13–15

AGA Adaptive Genetic Algorithm. 16, 17

ANNs Artificial Neural Networks. 9, 10

CC Cloud Computing. iii, iv, 1, 4

CN Core Network. 5, 6

CNNs Convolutional Neural Networks. 9

CPU Central Processing Unit. 18, 20, 23, 24

DDQN Double Deep Q-Network. 27

DNN Deep Neural Networks. 2

DQN Deep Q-Network. 9, 11, 21, 27

DRL Deep Reinforcement Learning. iii, iv, 2, 3, 22, 27–29, 39

DRQN Deep Recurrent Q-Network. 20

Dueling DQN Dueling Deep Q-Network. 13, 27

eNB Wireless Base Station. 5, 6

ETSI European Telecommunications Standards Institute. 1, 4

FC Fog Computing. 4

FMC Follow Me Cloud. 4, 5

GPU Graphics Processing Unit. 15

IaaS Infrastructure as a Service. 4

IoT Internet of Things. 1

ISG Industry Specification Group. 4

MC MobiScud Control. 5

MDP Markov Decision Process. 2, 6–8, 19, 27

MEC Mobile Edge Computing. iii–v, 1–5, 17, 18, 20–26, 29–34, 36, 37, 39, 40

MMC Mobile Micro Cloud. 4, 5

MobiScud Fast Moving Personal Cloud. 4, 5

NFV Network Function Virtualization. 5, 6

POMDP Partially Observable Markov Decision Process. 7, 20

QoS Quality-of-Service. 1, 5

RAN Radio Access Network. 4, 5

RL Reinforcement Learning. 2, 6–8, 15–17, 28, 40

RNNs Recurrent Neural Networks. 9

SCC Small Cell Cloud. 4, 5, 22

SCeNBs Small Cells. iii, iv, 5, 6, 22

SCM Small Cell Manager. 5

SDN Software Defined Network. 4–6, 19, 20

TD Temporal Difference. 9, 11–13

UE User Equipment. iii, iv, 1–6, 17–26, 29–34, 36–40

VMs Virtual Machines. 4

Chapter 1

Introduction

1.1 Context

There has been a continuous exponential increase in the number of User Equipment (UE), such as smartphones, laptops and Internet of Things (IoT) devices. Their need to run ever more complex applications, given their energy and computation constrained environment, has led to the rise of Cloud Computing (CC) as an alternative to offload computation and storage needs. In the CC paradigm, computation resources are located in centralized data centers that can be considered infinite. This paradigm brings several advantages [1]:

1. By offloading computation, it extends the battery life of UE;
2. Enables computationally complex applications intractable with UE computation capabilities;
3. Provides seemingly unlimited storage capacity.

However, there are also shortcomings. The two main problems introduced by this paradigm are:

1. The distance from UEs to these servers introduces an unpredictable communication latency that makes some delay-constrained applications unviable;
2. This increase in UEs and their communication needs with CC servers leads to ever more congested network links decreasing Quality-of-Service (QoS) for everyone in the network.

These shortcomings gave rise to a new emerging concept known as Mobile Edge Computing (MEC). The main idea of MEC is to bring computation resources closer to the edge of the mobile network enabling offloading of complex computation tasks with strict delay requirements. As defined by the European Telecommunications Standards Institute (ETSI) in [2], this can be achieved by allocating computing nodes at the network's edge in a fully distributed manner to reduce communication overhead and execution delay for UEs.

1.2 Motivation

Edge computing nodes come with limited radio, storage and computational resources, which raise three main challenges [1]:

- The **decision of which computing tasks** are profitable for the UE to offload to the MEC servers in terms of energy consumption and execution delay.
- How to efficiently **allocate the limited computation resources** within the MEC servers in order to minimize response delay and load balance the computing resources and communication links.
- **Mobility management** to guarantee MEC service continuity and efficiency for UEs roaming the network.

Given the heterogeneous and stochastic nature of network topologies, traditional optimization techniques lack the scalability and adaptability to deal with unknown network conditions. Recent breakthroughs in machine learning algorithms, showcasing their ability to solve and adapt to complex problems previously thought impossible to be solved by a computer, led many researchers to explore applying these methods to the MEC challenges.

The problem of making offloading and resource allocation decisions in MEC can be simulated and performance benchmarks are easily defined. This makes their definition as a Markov Decision Process (MDP) straightforward and allows the use of promising algorithms, like Deep Neural Networks (DNN) trained using Reinforcement Learning (RL) or Deep Reinforcement Learning (DRL) algorithms for short.

1.3 Objectives

The main goal of this work is to expand the research efforts in MEC with the development of a network management agent capable of making offloading decisions from a heterogeneous network of UEs to a heterogeneous network of MEC servers. These decisions should take into account the battery, computation and communication constrains of each UE and MEC server.

To achieve this goal multiple related works were studied in order to develop a realistic network simulation environment, understand the state-of-the-art in DRL algorithms and their application to the MEC challenges.

1.4 Contributions

The main contributions of this work are:

- The release of an open-source realistic network simulation environment as an OpenAI Gym, [3], environment. Allowing for easy experimentation with different DRL algorithms;

- The implementation of an open-source, A2C agent, capable of making intelligent offloading decisions that take into account battery, computation and communication constraints overperforming the baselines in a heterogeneous network of several UEs and MEC servers;

The code for the simulator, baselines and agent, as well as environment configurations, can be found in the following code repository: https://github.com/Carlos-Marques/rl-MEC-scheduler.

1.5 Report outline

This document is structured as follows:

- Chapter 1 serves as a brief introduction and motivation for the work done in this thesis. In particular, the relevance of MEC in the current and future technological landscape and its current challenges.
- In Chapter 2, first, a review of several MEC network architectures is made. Then, DRL concepts and algorithms are introduced. Finally, related works using DRL methods to solve challenges in MEC are explored.
- Chapter 3 focuses on defining the problem statement, the proposed solution and the methods and tools.
- Chapter 4 serves to showcase baseline algorithms and test the agent's performance in terms of learning capacity, scalability, data efficiency, robustness, stability and adjustability.
- In Chapter 5, first, a summary about this work is made. Then future work is explored.

Chapter 2

State of the art

2.1 Overview of MEC architectures

One of the first efforts to bring computation resources closer to the edge was the cloudlet concept presented in [4]. The main idea behind it was to allocate powerful computers at WiFi hotspots that could sell their Infrastructure as a Service (IaaS) through the use of Virtual Machines (VMs).

Another concept to bring computation closer to UEs is the idea of an *ad-hoc* cloud like the one presented in [5]. The idea proposes that the computation power of a network of non-exclusive and sporadically available hosts can be harvested and abstracted as a service to which UEs can offload their computational needs.

In order to standardize and generalize other concepts of edge computing, the OpenFog Consortium proposes a standard Fog Computing (FC) architecture [6]. This architecture presents computation resources as a continuum between the UEs and the cloud by abstracting edge computing platforms like cloudlets or ad-hoc clouds as fog nodes that make their services available to UEs. These fog nodes can be organized physically or by a Software Defined Network (SDN) and share their workload, making offloading decisions on UE tasks to other fog nodes or CC centralized data centers whenever needed.

While FC is defined in [6] as a system-level horizontal architecture that distributes resources and services of computing, storage, control and networking anywhere along the continuum from a cloud data center down to users. the Mobile Edge Computing (MEC) concept presented by the Industry Specification Group (ISG) within the European Telecommunications Standards Institute (ETSI) in [2] is focused on integrating computation resources within the Radio Access Network (RAN) in very close proximity to mobile subscribers.

As presented in paper [1], several MEC architectures have been proposed in order to integrate and manage computation resources at the RAN level, such as Small Cell Cloud (SCC) [7], Mobile Micro Cloud (MMC) [8], Fast Moving Personal Cloud (MobiScud) [9], Follow Me Cloud (FMC) [10] and CONCERT [11]. These architectures differ mainly in terms of distance from UEs to computation resources and can then be divided into two main types: 1) edge computing at the access node level; 2) distributed data centers. Finally, the CONCERT architecture merges ideas present in SCC and MMC with the ideas

of MobiScud and FMC.

2.1.1 Edge computing access nodes

The main idea of architectures like Small Cell Cloud (SCC) and Mobile Micro Cloud (MMC) is allocating computation resources at the access node level of a mobile network.

The Small Cell Cloud (SCC) architecture was first introduced by the European project TROPIC [7] and later expanded by the SESAME project [12]. It proposes that low-powered cellular radio access nodes, i.e. SCeNBs, be enhanced with computation and storage capabilities. Because of the current popularity of SCeNBs in 4G networks and their estimated wide adoption as a solution to guarantee the data capacities of 5G networks, this architecture could provide ample computation resources for applications with high latency requirements.

These computation resources at the SCeNBs would be pooled and presented as a service to UE's applications through a management agent. This so-called Small Cell Manager (SCM) could be deployed in a distributed hierarchical manner with a local SCM managing a cluster of SCeNBs while being connected to other SCMs through a remote SCM located at the CN of the service provider. The local SCM would need to be aware of the state of its cluster of SCeNBs and make offloading decisions on how to distribute work within its cluster. The remote SCM could help distribute work between local SCMs by offloading computations to neighboring clusters. This topology can be seen as a N to N problem where N UEs offload computation to N MEC servers located at the SCeNBs level.

In the case of the Mobile Micro Cloud (MMC) presented in [8], the main idea is to allocate a MEC server along with the Wireless Base Station (eNB) instead of one at every SCeNB. This topology could then be seen as a N to 1 problem where N UEs offload computation to 1 MEC server located at the eNB level. These MEC servers would then be interconnected directly or through backhaul to each other in order to guarantee QoS in the mobility management of UEs.

2.1.2 Distributed data centers

The main idea behind proposed architectures like the Fast Moving Personal Cloud (MobiScud) [9] and Follow Me Cloud (FMC) [10] is to integrate cloud services into the mobile networks using data centers closer to the edge of the network.

As presented in [9], the Fast Moving Personal Cloud (MobiScud) architecture makes use of network operator's data centers located within RAN or close to RAN instead of allocating computation resources at the access node level like SCC or MMC architectures. By using Software Defined Network (SDN) and Network Function Virtualization (NFV) technologies, it proposes to integrate cloud services seamlessly at the mobile network level. This type of architecture introduces the concept of a MobiScud Control (MC) agent that interfaces with the mobile network, SDN switches and the cloud operator in order to orchestrate and route data between the various composing operator's clouds in order to maintain QoS when the UE moves through the network.

The Follow Me Cloud (FMC) architecture proposed in [10] follows a similar idea to the MobiScud

architecture but instead of using the operator's data centers at the RAN level, it assumes the use of distributed data centers moved farther away into the Core Network (CN) of the operator.

2.1.3 CONCERT architecture

As proposed in [11], the CONCERT architecture exploits the NFV and SDN technologies in order to define a decoupled control/data model. The data plane is composed by the eNB, SCeNBs, SDN switches and computation resources, being them at the access node level, local data centers or even remote data centers. The control plane is composed by the management agent, referred to as the conductor, which has access to the network state and manages its resources presenting them as software defined services to UE's applications. This type of centralized approach is a clear example of an orchestration approach. As an alternative, a choreography approach is when entities cooperate in a decentralized manner without holding complete knowledge of the network, as is the case of the solution presented in [13].

2.2 Reinforcement Learning (RL)

There are three main machine learning paradigms, supervised learning, unsupervised learning and reinforcement learning. While supervised learning needs a lot of labeled input/output data and unsupervised learning learns patterns from unlabeled data, Reinforcement Learning (RL) is used in problems where an agent learns to solve a closed-loop problem by changing its future actions based on a reward resulting from its actions on the environment and its effects on its state.

This paradigm has shown great potential in solving environments that can be simulated and easily benchmarked, which is the reason most research on RL is made by solving video games, which offer a perfect environment in terms of reproducibility, easy simulation and baked in benchmarks that can be used to compute a reward signal (points, levels, etc.).

RL tasks are usually modeled as Markov Decision Processes.

2.2.1 Markov Decision Process (MDP)

The Markov Decision Process is a mathematical framework that can be used to model a time-discrete decision process in a stochastic environment as long as it satisfies the Markov property, which states that given the present, the future does not depend on the past.

MDPs are usually defined by a four-element tuple $< S, A, P, R >$:

- S is the state space, which represents the set of all agent and environment states, $s_t \in S$;
- A is the action space, which represents the set of all possible agent actions, $a_t \in A$;
- $P : S \times A \times S \rightarrow [0, 1]$ is the transition probability distribution $P(s_{t+1}|s_t, a_t)$ of a new state s_{t+1} given that the system is in state s_t and action a_t is chosen;

- $R : S \times A \rightarrow \mathbb{R}$, is the reward signal, r_t, of the system when action a_t is taken from state s_t to s_{t+1}.

Many RL tasks can be approximated as MDP with careful definition of the state signal, s_t, to include past relevant information for the current action decision, a_t.

As stated in [14], given these conditions, the system dynamics can be defined entirely by the current state's probability distribution:

$$p(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a). \tag{2.1}$$

MDP assumes full observability of the current environment/agent state. Alternatively, the agent is said to have partial observability when it can only observe part of the entire environment or its observations are corrupted. These types of problems are formulated as Partially Observable Markov Decision Process (POMDP).

2.2.2 Partially Observable Markov Decision Process (POMDP)

As a generalization of a MDP, in a POMDP, the agent does not have access to the complete state. Instead, a sensor model must be defined on top of the usual MDP definition. For this reason, an POMDP is usually defined as a six-element tuple $< S, A, P, R, R, \Omega, O >$ [15], where the new Ω and O define the sensor model:

- Ω is the set of observations, where $o_t \in \Omega$ is the observation of the agent at time step t;
- O is the set of conditional probabilities of observation $O(o_t|s_{t+1}, a_t)$. This means the probability of the observation, o_t, given the new state s_{t+1} resulting from the action a_t.

2.2.3 Value function

The goal of an RL agent is to learn a policy, π, that maximizes the expected cumulative reward:

$$\pi : A \times S \rightarrow [0, 1], \tag{2.2}$$

$$\pi(a, s) = P(a_t = a|s_t = s). \tag{2.3}$$

The expected cumulative reward for a given policy starting in state $s_0 = s$, can be defined as the value function:

$$V_\pi(s) = E[R] = E[\sum_{t=0}^{\infty} \gamma^t r_t|s_0 = s], \tag{2.4}$$

where the return, R, is the sum of future rewards discounted according to the discount-rate, $\gamma \in [0, 1]$:

$$R = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{2.5}$$

The discount-rate γ is a hyper-parameter that represents how much the agent values future rewards in comparison with current rewards. If $\gamma = 0$, then the agent only cares about instant reward. If $\gamma = 1$ and the system has no final state, then the problem of calculating the return, R, becomes infinite.

The goal can then be defined as finding a policy, π, that maximizes $V_\pi(s)$:

$$V^*(s) = \max_\pi V_\pi(s). \tag{2.6}$$

An action-value function can then be defined as the expected return of taking action a, in state s, and then following policy π:

$$Q_\pi(s, a) = E[R|s, a, \pi]. \tag{2.7}$$

Given the optimal policy, π^*, we can take the optimal action by choosing the action, a, with the highest, $Q_{\pi^*}(s, a)$, value.

Given full knowledge of the MDP, this problem can be solved using classical dynamic programming techniques like value and policy iteration in which expected values are computed over the whole state-space in order to approximate the optimal action-value function, Q^*. These methods have two major problems. First, they assume complete knowledge of the MDP, wherein most RL tasks, the transition probability distribution, P, and reward signal, R, are usually unknown *a priori* and need to be learned from experience. Second, given the brute force nature of these algorithms, they become intractable for higher dimension state-spaces.

One algorithm that tries to learn the action-value function from experience without needing to maintain a model of the environment is the Q-learning algorithm.

2.2.4 Q-learning

The Q-learning algorithm tries to approximate the action-value function through continued experience on the environment.

$$Q : S \times A \to \mathbb{R} \tag{2.8}$$

This can be done my maintaining a Q-table with a row for each state and a column for each possible action. Each of the cells of the Q-table correspond to the action-value, $Q(s, a)$, and is usually initialized as an arbitrary fixed value.

At each time step t, an action a_t, is taken on the state s, resulting in a new state s_{t+1}, and in a reward r_t. Given these values, the Q-table is updated according to:

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_{a_{t+1}} Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)), \tag{2.9}$$

where $\alpha \in [0, 1]$ is the learning rate, which is a hyper-parameter that determines how much new

information should affect the current belief.

Unlike Monte Carlo methods, which separate sampling from the environment from updating the policy based on the results, Q-learning is a Temporal Difference (TD) method, which means that as it follows the policy to explore the environment, it also updates the action-value estimations in an online fashion.

The simple Q-table approach to Q-learning has a glaring issue with higher dimension state and action spaces because it maintains a Q value for each state and action combination. An alternative to the Q-table is using the capabilities of information encoding of Artificial Neural Networks (ANNs) to approximate the action-value function.

2.2.5 Deep Q-Network (DQN)

DQN was proposed in [16] as a non-linear approximator of the action-value function in Q-learning. It proposes to solve the memory issues and the time required to explore each state in Q-tables and other function approximation methods. Due to the capabilities of ANNs to encode information and generalize lessons from experiences to unseen inputs, DQNs are able to solve problems with state and action spaces of higher dimensions, like playing video games from pixels and game scores, while requiring less memory and computation than other methods.

The main idea is to substitute the Q-table or other function approximation methods with ANNs composed of multiple layers of neuron-inspired nodes with different activation functions. A diagram comparing the Q-table to the DQN architecture can be seen in Figure 2.1. These ANNs have been shown to be able to solve complex representation problems by maintaining simpler internal representations. A simple model of this neuron behavior can be defined as:

$$y_j = f(\sum w_{ij} x_i + b), \tag{2.10}$$

where y_j is the output resulting from the application of a non-linear activation function, $f(.)$, to the sum of each input x_i multiplied by the learned weights w_{ij} offset by a bias value, b.

Based on the internal structure and composition of each layer, there are many types of ANNs. One important type are the Convolutional Neural Networks (CNNs), which maintain convolutional layers that are specially equipped for tasks where spatially closer inputs can be assumed to be somehow related. These visual cortex inspired processes makes them very good at processing high dimension input images.

Another important class of ANNs are the Recurrent Neural Networks (RNNs), which maintain an internal state memory in some neurons, making them good candidates in problems with correlated sequences of inputs.

The proposed ANN in [16] is a CNN that can be defined by its weights, θ_i, at each iteration. In order to solve the instability and divergence issues found when using a non-linear function approximator of the action-value function, two methods are proposed. Because these issues arise from the correlations present in the sequence of observations, the paper first proposes a biologically inspired mechanism termed experience replay that collects samples from interaction with the environment in a replay buffer,

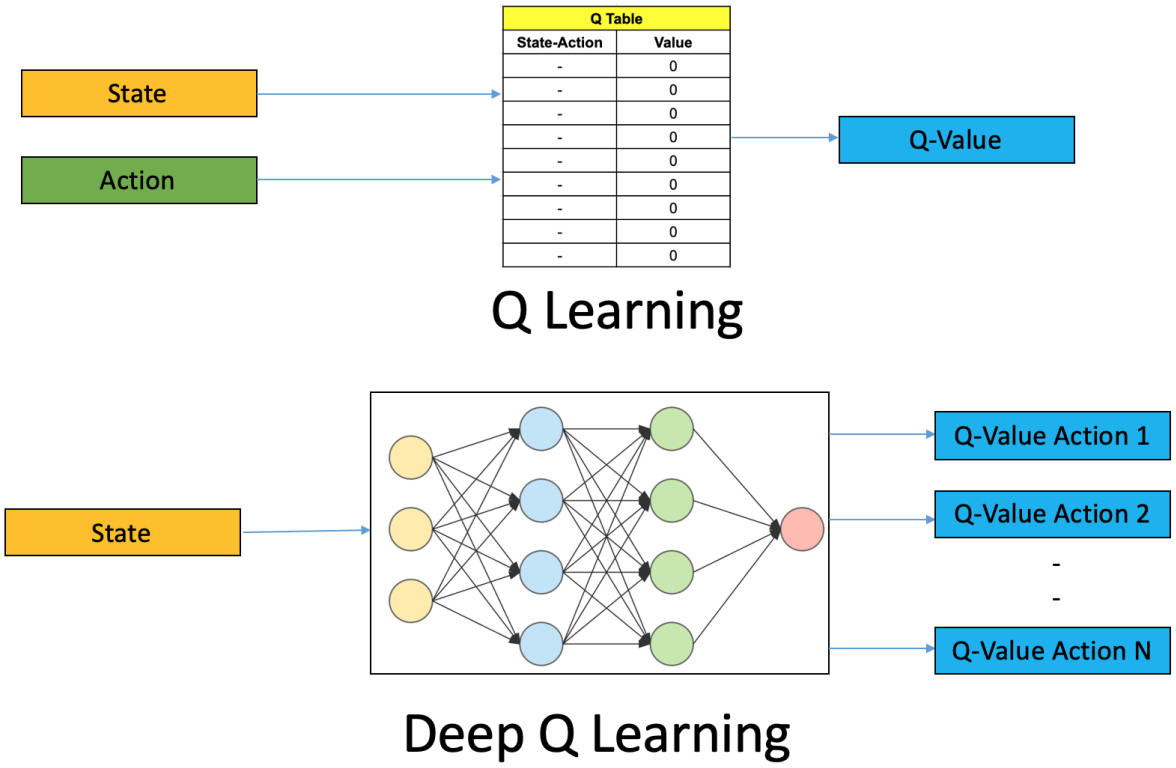

Figure 2.1: Deep Q Learning architecture comparison with Q-table from [17].

$U(D)$, and samples randomly from it for training. The second method is an iterative update policy that only adjusts the target weights, θ_i^-, to the new computed weights, θ_i, every C steps.

Since we are dealing with reinforcement learning, the loss function of these ANNs can not be derived directly from the difference between the expected true value and output, like with supervised learning. Instead, the loss function must be calculated according to:

$$L_i(\theta_i) = E_{(s_t,a_t,r_t,s_{t+1})\sim U(D)}[(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_i^-) - Q(s_t, a_t; \theta_i))^2] \tag{2.11}$$

Where the loss, L_i, of iteration, i, can be calculated by uniformly sampling a mini-batch of experiences from $U(D)$ and calculating the expected value from the difference between our target $t_i = r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_i^-)$, calculated according to the non-updated weights, θ_i^-, and the resulting action-value predicted by our model.

The gradient can then be calculated by:

$$\nabla L_i(\theta_i) = E[(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_i^-) - Q(s_t, a_t; \theta_i)) \cdot \nabla Q(s, a; \theta_i)] \tag{2.12}$$

And popular training methods of ANN using gradient descent can be used.

2.2.6 Double Deep Q-Network (DDQN)

Another proposed method to try to solve the correlation issues present with DQNs is the idea presented in [18] of maintaining two action-value estimation networks, Q^A and Q^B. Whereas in [16] the loss is calculated according to a target computed with the maximal valued action of a frozen network, θ_i^-, in paper [18] it is proposed that at each update step of one of the networks, the target, t_i, must be calculated using the maximal valued action of the other network.

$$t_i^A = r + \gamma \max_{a_{t+1}} Q^B(s_{t+1}, a_{t+1}) \tag{2.13}$$

$$t_i^B = r + \gamma \max_{a_{t+1}} Q^A(s_{t+1}, a_{t+1}) \tag{2.14}$$

While each Q network still serves the same purpose as with the DQN algorithm, of approximating the action-value function of the environment, by using a different network's maximal valued action, this algorithm allows for the decoupling from action selection and its evaluation. This, coupled with different experience sets for training each network, results in a decrease in the correlation issues present in the DQN algorithm. For picking the next action, the average of the two Q values is calculated in order to evaluate its quality.

2.2.7 Prioritized experience replay

Improving on the concept of experience replay introduced in paper [16], paper [19] proposes a way of prioritizing the experience replay buffer, $U(D)$, in order to make the algorithm more efficient and effective. The main component of prioritized experience replay is the criterion used to measure the importance of each experienced transition. As proposed in [19] the TD-error, δ, between the target, t_i, and the predicted value $Q(s_t, a_t)$ can be used as a proxy for how unexpected the transition is.

With this TD-error, δ, a greedy TD-error prioritization algorithm can be devised where experiences for the update step are sampled from the $U(D)$ in order of the highest absolute TD-error first. This is done with the caveat that new transitions without a known TD-error are prioritized above all other experiences in order to guarantee that all experiences are seen at least once.

This greedy TD-error prioritization algorithm comes with several problems. First, in order to avoid recalculating TD errors for the entire replay memory, they are only updated on replayed transitions. This has the consequence that transitions with low TD error at first might not be replayed in a long time and due to the limited size of the replay buffer and its sliding window nature, might never be replayed. Other problems with this algorithm are its sensitivity to noise spikes and its tendency to over-fit on a small subset of transitions that have high initial TD-error.

In order to solve these issues, the authors of [19] present a stochastic sampling method that interpolates between pure greedy prioritization and uniform random sampling. To achieve this, they propose

that sampling on $U(D)$ be made according to a sampling probability of each transition i defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{2.15}$$

where $p_i > 0$ is the priority of transition i and α is a hyper-parameter that determines how much prioritization should be used, with $\alpha = 0$ corresponding to the uniform sampling case.

The priority of transition can be defined as:

$$p_i = |\delta_i| + \epsilon, \tag{2.16}$$

where δ_i is the TD-error of transition i and ϵ is a small positive constant that prevents edge-cases of transitions not being revisited once their error reaches zero.

An alternative formulation of the priority of transition is:

$$p_i = \frac{1}{rank(i)}, \tag{2.17}$$

where $rank(i)$ is the rank of transition i when the replay memory is sorted in descending order according to transition TD-errors, $|\delta_i|$. This variant is more likely to be more robust because it is insensitive to outliers.

2.2.8 Dueling Deep Q-Network (Dueling DQN)

Improving upon the concept of double Q-learning presented in [18], the authors of [20] present an alternative architecture that tries to explore the insight that, for many states, the action taken does not affect what happens. The idea is then to separate the single-stream architecture of [16] into two streams that estimate the value, $V(s)$, and advantage, $A(s, a)$, functions, respectively. A diagram showcasing this split is shown in Figure 2.2.

The value function, $V(s)$, as stated in Equation (2.4), indicates the expected cumulative reward from beginning in state s and following the policy, π, and the action-value, $Q(s, a)$, indicates how good is an action, a, given the state, s. Then, the advantage, $A(s, a)$, can be defined as:

$$Q(s, a) = V(s) + A(s, a). \tag{2.18}$$

This corresponds to the advantage that the agent gains from taking action, a, in state, s.

As shown in [20], special consideration must be taken when designing the final aggregating module to output Q. A simple aggregation module like the one presented in Equation (2.18) would make the $V(s)$ inseparable from the $A(s, a)$. So the authors propose an alternative module that forces the advantage function estimator to have zero advantage at the chosen action:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \alpha)), \tag{2.19}$$

where α and β are hyper-parameters of each stream of fully-connected layers, $A(s, a)$ and $V(s)$ streams, respectively and θ are the network weights.

By exploiting the insight that some actions do not affect the state value, this architecture allows the $V(s)$ function to be learned more efficiently and faster, resulting in faster convergence of the agent to an optimal policy.

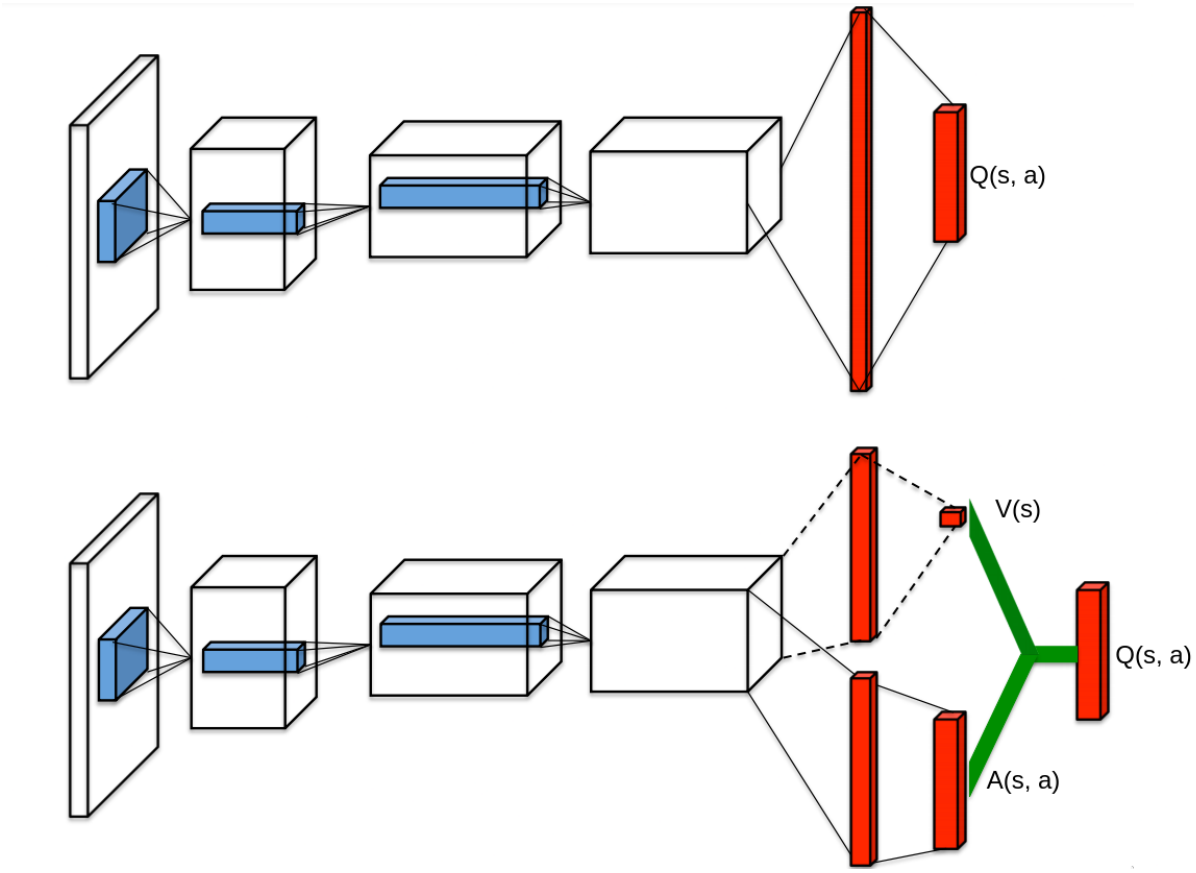

Figure 2.2: Dueling DQN architecture (bottom) comparison with normal DQN (top), taken from [20].

2.2.9 Asynchronous Advantage Actor-Critic (A3C)

Actor-critic methods try to combine the benefits of both value-iteration methods, like Q-learning and policy-iteration methods, like Policy Gradient, by separating the agent as an actor network and a critic network. The actor network contains the policy that maps states to actions by a set of parameters θ. While the critic network evaluates the value function $V(s)$, the action-value $Q(s, a)$ or the advantage of an action $A(s, a)$, and, based on the TD-error updates both the actor network and itself. In the case of the A3C, presented in [21], the critic network tries to estimate the $V(s)$, and the actor network tries to estimate the policy, $\pi(s)$ by maintaining a shared network with two streams like in the case of Dueling DQN, [20]. A diagram showcasing the A3C architecture is shown in Figure 2.3.

The gradient of the loss function can then be defined as:

$$\nabla L(\theta) = \nabla_{\theta'} log\pi(a_t|s_t;\theta')A(s_t,a_t;\theta,\theta_v). \tag{2.20}$$

Advantage can be defined as in Equation (2.18), and by using the relationship between the Q and the V functions from the Bellman optimality equation, the following relationship can be established:

$$Q(s_t,a_t) = E[r_{t+1} + \gamma V(s_{t+1})], \tag{2.21}$$

$$A(s_t,a_t;\theta,\theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_v) - V(s_t;\theta_v), \tag{2.22}$$

where k is the number of steps used to the compute new update and γ is the discount-rate. This bypasses the need for multiple neural networks as Q and V function approximators.

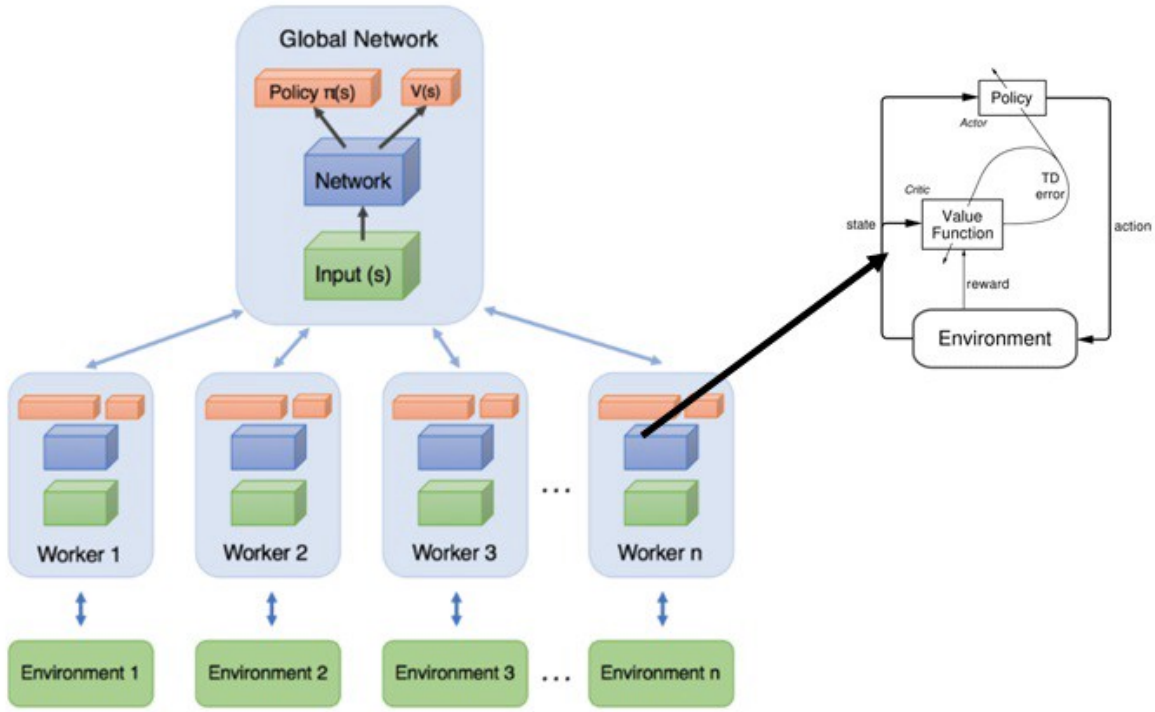

Figure 2.3: Asynchronous Advantage Actor-Critic (A3C) architecture, taken from [22].

The main difference between the A3C method and the alternative A2C method proposed in [23] and [24] is that in A3C each worker works independently from other workers running k number of experiences and then computing the gradient and updating with a central network asynchronously, whereas in A2C there is a synchronization step where each worker waits for all workers to finish their segments of experience and compute their gradient before updating the central network. Also of note, the authors of the A2C architecture tested if the distributed asynchronous nature of the A3C network gave it better performance given the stochastic nature of its updates but found no such advantage. The A2C method

runs more effectively when using single GPU machines, which perform best on large batch sizes of similar computations.

2.3 Exploitation vs Exploration

One major challenge a RL agent faces is the exploration vs exploitation dilemma. Given a fixed set of resources that must be allocated between competing choices in a way that maximizes expected reward in an unknown transition probability and reward signal environment, the agent must choose to perform actions that exploit known strategies versus performing unknown actions to explore the environment for better strategies. These types of problems have been studied extensively as multi-armed bandit problems.

2.3.1 ϵ-greedy algorithm

The most common algorithm used to solve this dilemma is the ϵ-greedy algorithm in which the agent performs a greedy selection of the best action most of the times with a ϵ probability of choosing a random action.

The best action, in the case of a Q-learning algorithm, can be defined as:

$$a^* = \arg\max_a Q(s, a). \tag{2.23}$$

Where in the case of an actor-critic method like A3C, it can be defined as:

$$a^* = \arg\max_a A(s, a). \tag{2.24}$$

To guarantee sufficient exploration at the beginning versus convergence of the RL agent policy at the end of the training, usually, the ϵ value shrinks as training goes on. This reflects the intuition that as training goes on, the model becomes better at understanding the environment and the effects of its actions which in turn makes the need to explore less important.

2.3.2 Boltzmann exploration

As explained in paper [25], the Boltzmann exploration policy is widely used in RL problems. The main idea of this algorithm is to build a Boltzmann distribution where instead of using the state's energy, the evaluation of the quality of an action, $Q(s, a)$, in the case of Q-learning algorithms or its advantage, $A(s, a)$, in the case of the A3C algorithm is used. The probability of choosing each action can be defined according to:

$$p_i = \frac{e^{-\varepsilon_i/\tau}}{\sum_{j=1}^{M} e^{-\varepsilon_j/\tau}}, \tag{2.25}$$

where ε is the value of the action, M is the total number of actions and τ is a hyper-parameter usually called temperature that defines a spectrum between picking the optimal action or a completely random action. When the τ parameter is close to zero, the Boltzmann policy becomes like the greedy policy, picking the highest valued action. Instead, when the τ parameter is high, the probability is more widely distributed between other actions.

As with the ϵ-greedy, this τ parameter usually shrinks as training goes on, reflecting the intuition that exploration becomes less important as the agent learns the environment.

2.3.3 Adaptive Genetic Algorithm (AGA)

In RL problems with high-dimensional action spaces, methods like the ϵ-greedy policy are very inefficient in their exploration of the environment. To address this problem, the authors of [26] propose an Adaptive Genetic Algorithm (AGA) to make optimal policy convergence faster by reducing useless exploration without reducing performance.

The main idea behind AGA is to try to use the critic network to evaluate expected reward for each state-action. The presented solution in [26] is a Q actor-critic architecture where the AGA takes as input the output action from the actor network. A diagram for the proposed architecture can be seen in Figure 2.4.

If the loss of the critic network is larger than a defined hyper-parameter, φ, it is assumed that the critic network cannot evaluate actions very well so a random action is generated. This φ hyper-parameter can be set as the convergence loss obtained by pre-training the critic network.

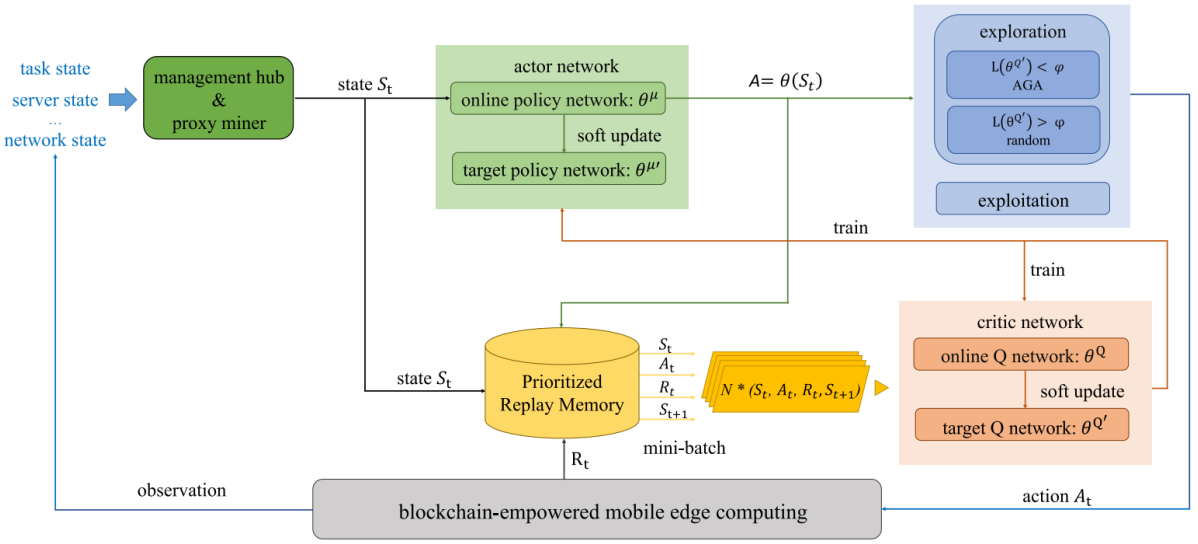

Figure 2.4: Actor-critic with AGA from [26].

When the critic network can evaluate the state-action pairs, the AGA works by taking the output action, A, from the actor network and generating $m - 1$ random actions obtaining an initial population of candidate solutions of size, m, where each action is considered an individual of the population. The genetic operators of crossover and mutation can then be used to create different generations of actions.

Using the critic network evaluation as the fitness function in roulette wheel genetic selection, the best actions are selected, and the new generation replaces the population.

The crossover and mutation genetic operators are applied according to their respective probabilities p_c and p_m.

$$p_c = \begin{cases} \frac{k_1(R_{min}-R')}{\overline{R}-R_{min}} & R' \leq R_{min} \\ k_3 & R' > R_{min} \end{cases}, \tag{2.26}$$

$$p_m = \begin{cases} \frac{k_2(R-R_{min})}{\overline{R}-R_{min}} & R \geq R_{min} \\ k_4 & R < R_{min} \end{cases}, \tag{2.27}$$

where $\overline{R}$ and R_{min} are the average and minimum values of the population, respectively. These parameters are used to increase p_c and p_m when the population converges to a local minimum. In order to combat disruptions in the global minimum solution, p_c should depend on its parent action value R' and p_m should depend on the value of the mutating action. $k1$, $k2$, $k3$ and $k4 < 1$ are hyper-parameters that define how much crossovers and mutations should happen.

AGA ends when a K number of iterations are reached and the highest valued action, A^*, is selected. This K value can be adapted by:

$$K = \begin{cases} \max(0, K-1) & A^* = A \\ \min(K + \phi(||A - A^*||_2), K_{max}) & A^* \neq A \end{cases}. \tag{2.28}$$

Where ϕ is a strictly monotonically increasing function and K_{max} is an upper limit on the number of generations.

2.4 Related work

In order to understand the state of the art in solving MEC challenges using RL, several works were reviewed. Their problem statements and solutions are summarized in the following sections.

2.4.1 User side task offloading

Research papers such as [27] and [28] present a local offloading decision algorithm that considers a 1 UE to 1 MEC server topology in a 4G environment, where the UE has information about the required computation graph of its application and makes decisions on which computations to offload and which computations to execute locally.

In [27], the algorithm takes a sequentially dependent list of application computation components and classifies each one as local or remote execution. It assumes that each component's input data amount depends on the output data of the component preceding it and tries to estimate its work as a function of input data and computation complexity as presented in:

$$W_c = V \cdot O_n \cdot d_{c-1,c}, \tag{2.29}$$

where W_c is the CPU clock cycles of the computation component (c_n), V denotes the number of clock cycles a processor will perform per byte, O_n is the computation complexity of c_n and $d_{c-1,c}$ is the output data amount of the previous component. The O_n notation is introduced as the data amplification factor of c_n, given that the input data and the processed data are not equal in amount since the input data can be processed several times.

This paper takes into account the time required to do the computation locally, the transfer delay of uploading input data to the MEC server defined in Equation (2.30) and the transfer delay of the downloading of output data defined in Equation (2.31):

$$r_u = n\frac{B}{N}log_2(1 + \frac{p_u|h_{ul}|^2}{\Gamma(g_{ul})d^\beta N_0}), \tag{2.30}$$

$$r_d = n\frac{B}{N}log_2(1 + \frac{p_d|h_{dl}|^2}{\Gamma(g_{dl})d^\beta N_0}), \tag{2.31}$$

where B is the bandwidth, β is the path loss exponent, d is the distance between the UE and the MEC server, n is the number of subcarriers that are allocated for the transmission, N_0 is the noise power, p_u and p_d are the transmit powers, h_{ul} and h_{dl} are the channel fading coefficient for uplink and downlink, and g_{ul} and g_{dl} are the required bit error rate for the uplink and downlink. The SNR margin, Γ, required to satisfy the bit error rate (g_{ul} and g_{dl}) with quadrature amplitude modulation constellation, can be calculated according to:

$$\Gamma(g_l) = \frac{-2log5g_{ul}}{3}. \tag{2.32}$$

Data is only transferred as it is needed. This means that when two sequential components are offloaded to the MEC server, the output of the first component is needed as input for the second component. However, since they were both computed on the same machine, there are no upload or download delays between the two.

The proposed classification algorithm is a Deep Neural Network (DNN) trained using the exhaustively computed best solutions from 10,000 randomly generated states for an application consisting of 100 sequential components. This training method quickly becomes intractable for higher complexity problems.

This work does not take into account energy costs of transmission, idle energy costs of waiting for remote computation execution, assumes sequential data dependency in the execution graph and constant CPU allocation from the MEC server.

2.4.2 Centralized Fog Network managers

Paper [29] presents a centralized Software Defined Network (SDN) controller which maintains global knowledge of the network of fog nodes. Each fog nodes receives requests from UEs, and the SDN controller decides if these tasks should be computed on the receiving node or offloaded to a neighboring node.

This problem statement is formalized as an MDP, consisting of a state-space (S), action-space (A), transition probability distribution (P) and a reward (R).

- $S = \{s = (n^l, w, Q)\}$ is the state space:
  - $n^l \in \mathbb{N}\ (1 \leq n^l \leq N)$, is the fog node requested for tasks by end users;
  - $w \in \mathbb{N}(1 \leq w \leq W_{max})$, is the number of requested tasks per unit time;
  - $Q = \{(Q_1, ..., Q_N) | Q_i \in \{0, 1, ..., Q_{i,max}\}$, is the number of tasks in the node's queue.
- $A = \{a = (n^0, w^0)\}$ is the action space:
  - $n^0 \in \mathbb{N}\ (1 \leq n^0 \leq N, n^0 \neq n^l)$, is a neighboring node to which node n^l is going to offload to;
  - $w^0 \in \mathbb{N}\ (1 \leq w^0 \leq W_{max})$, is the number of tasks to be offloaded to the neighboring node n^0. A node can only offload to nodes with an equal or less number of tasks currently requested. The tasks not offloaded are computed locally (w^l).
- $P : S \times A \times S \rightarrow [0, 1]$, is the transition probability distribution $P(s'|s, a)$ of a new state s' given that the system is in state s and action a is chosen.
- $R : S \times A \rightarrow \mathbb{R}$, is the reward of the system in state s and action a is taken.
  - $R(s, a) = U(s, a) - (D(s, a) + O(s, a))$:
    - $U(s, a) = r_u log(1 + w^l + w^0)$, is the immediate utility and where r_u is a utility reward;
    - $D(s, a) = \chi_d \cdot \frac{t^w + t^c + t^e}{(w^l + w^0)}$, is the immediate delay and where $t^w$is the average wait time at queue of node n^l and the node n^0, t^c is the communication delay between nodes and t^e is the execution time by node n^l and n^0. The χ_d is a hyper-parameter that defines importance of the delay;
    - $O(s, a) = \chi_0 \cdot \frac{w^l \cdot P_{overload,l} + w^0 \cdot P_{overload,0}}{w^l + w^0}$, is the overload probability and where χ_0 is an overload weight hyper-parameter and $P_{overload,i}$ is the task arrival rate at node.

To solve this MDP, the authors of this work propose a Q-table approach using the ϵ-greed algorithm as its policy, which makes its solution not very scalable to higher dimensional problems and makes it suffer from action value overestimation in noisy environments.

This work does not consider the possibility of local UE task execution and associated battery and computation constraints nor the delay in returning computation results from fog nodes back to the user device. It also assumes independence between tasks. Despite mentioning the possibility of software-defined fog nodes to be composed of many heterogeneous devices, this is not explored.

2.4.3 Decentralized Fog Network managers

Paper [13] continues the work of paper [29] by introducing a new architecture with two important concepts. First, it introduces the notion of dividing the computation resources of fog nodes into network slices, allowing for the allocation of certain slices to services with specific latency and resource requirements. The second important change it makes in its architecture is the change from a centralized SDN controller with knowledge of the entire network conditions to many distributed controllers located at each fog node working together.

This new problem can be formulated as a POMDP in which each fog node only knows its local state and makes decisions on which tasks to offload, where to offload them and how to allocate local slices resources to queued tasks. By doing this, no state messages are sent between fog nodes and only locally computed rewards are shared between them. These reward messages can be shared asynchronously from the decision process.

As a solution, each fog node has a Deep Recurrent Q-Network (DRQN) that takes as input a local observation stack of the previous four observations and makes a local offloading and resource allocation decision. A variation of the ϵ-greedy algorithm is used in which the ϵ is reset to the starting value with a small decay δ^{ϵ} $(0 < \delta^{\epsilon} < 1)$ every R^{ϵ} steps to solve the insufficient exploration in large state-action spaces that ϵ-greedy algorithm suffers from. These neural networks are trained with the experience replay technique trying to maximize the total system reward composed of the local fog node rewards resulting from their localized actions and their effect on the system as a whole.

The work presented in [30], builds on the work done in [13] by substituting the fog node's DRQNs by an A2C architecture with an actor for each network but a common critic estimator for all agents. This critic needs to be updated with all actions and states from each of the fog nodes but can do so asynchronously from the decision process.

2.4.4 Centralized MEC controller

Finally, the work presented in [31] present a central decision algorithm that makes offloading and resource allocation decisions between a network of UEs and one MEC server, as represented in Figure 2.5. The algorithm takes as input a list of all tasks that need to be computed by UEs at that time step and makes a decision of which tasks should be computed on the UE and which should be offloaded to the MEC server. It also decides how much of the MEC server's computation power should be allocated to each offloaded task.

It assumes that each UE n has a computation-intensive task $R_n \triangleq (B_n, D_n, \tau_n)$ at each time step to be executed, where B_n represents the input data for the computation, D_n represents the total number of CPU cycles required to compute the task and τ_n represents the maximum delay amount of that task.

This decision takes into account the delay and energy costs of computing locally, the delay of transmission to the MEC server, the energy needed to transmit and the idle energy consumption of the UE when waiting for the remote computation. Due to the assumption of a very high download rate and smaller size of output data than input data, this work ignores the delay and energy cost of download-

ing the processed result from the MEC server to the UE. To solve the problem statement, the authors present a simple DQN.

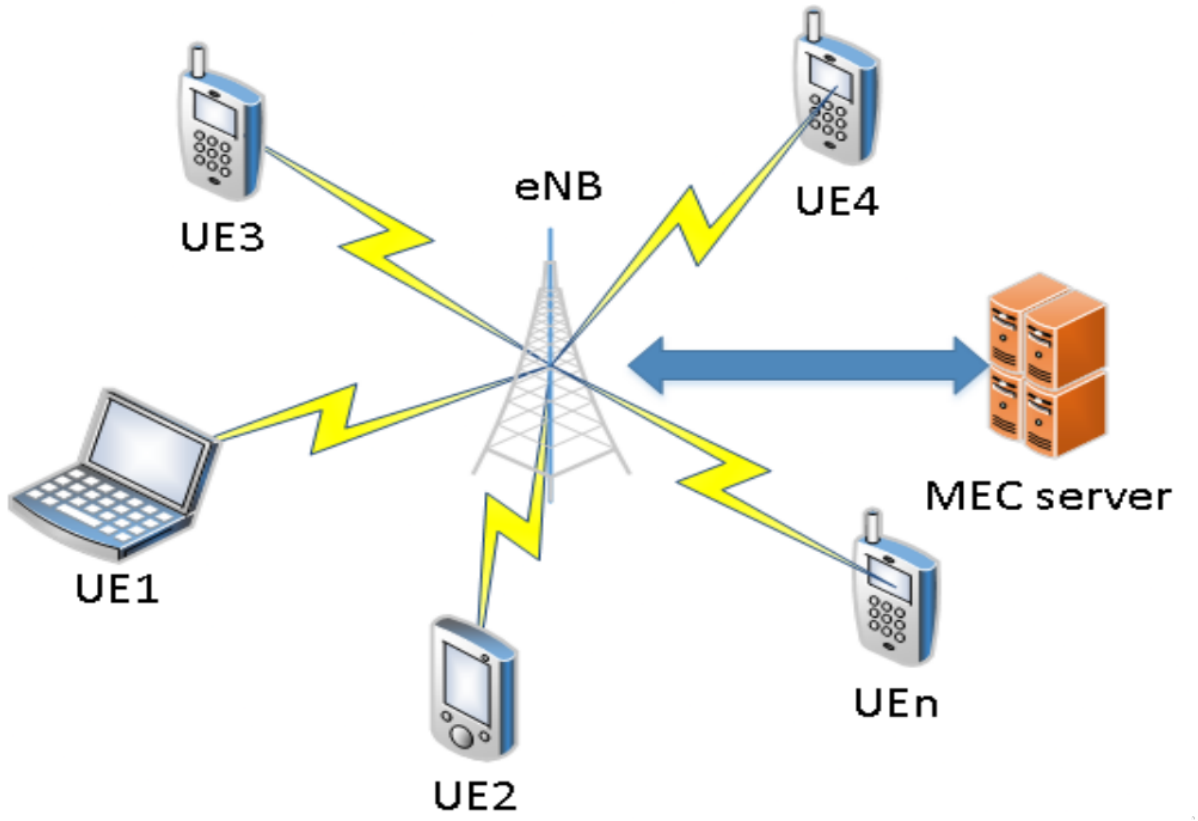

Figure 2.5: Network model from [31].

Chapter 3

Thesis proposition

3.1 Problem statement

The goal of this thesis is the design of a management agent capable of making offloading decisions from a heterogeneous network of UEs to a heterogeneous network of MEC servers. This work proposes to innovate by presenting a more complete system and exploring a network topology not present in related works. The proposed network manager could be seen as the conductor in the CONCERT architecture proposed in [11] or the small cell manager in the SCC architecture, [12]. This manager would be deployed locally and would manage a group of M MEC servers and N UEs, making offloading decisions on which UE tasks to compute locally, which tasks to offload and where tasks should be offloaded to. This decision should take into account communication delays, computation constraints and battery consumption.

Building upon the system presented in [31], the proposed system plans address its shortcomings by:

- Expanding the number of MEC servers that must be orchestrated from one to M servers, better representing a 5G network with several SCeNBs enhanced with computation capabilities;
- Testing the management agent in a heterogeneous environment, with UEs and MEC servers of different computation capabilities;
- Taking into account the delay and energy cost of downloading the processed result from the MEC server to the UE;
- Exploring modern DRL algorithms to solve the increased complexity problem.

3.1.1 System Model

The proposed network model considers *N* UEs and *M* MEC servers, as represented in Figure 3.1.

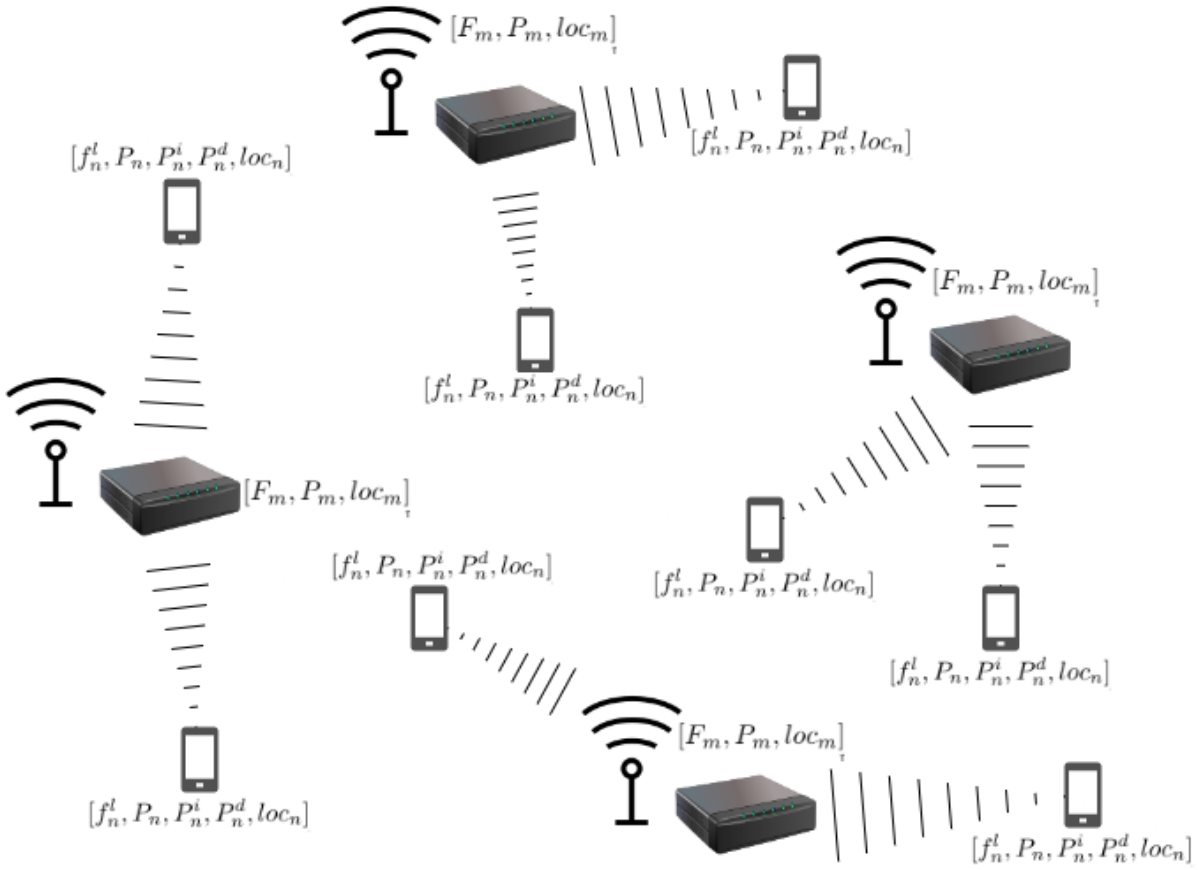

Figure 3.1: Proposed network model.

The set of UE is denoted as $\mathcal{N} = \{1, 2, ..., N\}$ while the set of MEC servers can be denoted as $\mathcal{M} = \{1, 2, ..., M\}$. For simplicity MEC servers are assumed to be connected to the power grid so their energy consumption is ignored.

- Each UE, $n \in \mathcal{N}$ can be described by its computing capacity, f_n^l (CPU cycles per second), transmit power, P_n, idle power consumption, P_n^i, download power consumption, P_n^d, and location, $loc_n = (x_n, y_n, z_n)$. The UE, n, can then be described by the vector, $u_n = [f_n^l, P_n, P_n^i, P_n^d, loc_n]$, and the system's UEs can be defined as the vector $\mathcal{U} = [u_1, u_2, ..., u_n]$.

- Each MEC server, $m \in \mathcal{M}$, can be described by its computing capacity, F_m (CPU cycles per second), its transmit power, P_m, and its location $loc_m = (x_n, y_n, z_n)$. The MEC server, m, can then be described by the vector, $s_m = [F_m, P_m, loc_m]$ and the system's MEC servers can be defined as the vector $\mathcal{S} = [s_1, s_2, ..., s_n]$.

- At each time step each UE is assumed to have a computation task to be completed. This task can either be computed locally or offloaded to one of the available MEC servers. The offloading decision of each computation task is denoted as $\alpha_n \in \{0, 1, ..., M\}$, where $\alpha_n = 0$ means local computation and $\alpha_n \in \mathcal{M}$ means offloading the task to MEC server $m = \alpha_n$. The total offloading decision can be defined as the decision vector $\mathcal{A} = [\alpha_1, \alpha_2, ..., \alpha_N]$ with a decision for each task.

- Each computation task, R_n, can be defined by an input data amount, B_n (bits), an output data amount, B_d (bits), the total number of CPU cycles required to compute it, D_n and the importance weights of time and energy costs, I_n^t and I_n^e. The importance weights of the task must satisfy $0 \leq I_n^t \leq 1$, $0 \leq I_n^e \leq 1$ and $I_n^t + I_n^e = 1$. The task, R_n, can then be described by the vector, $R_n = [B_n, B_d, D_n, I_n^t, I_n^e]$ and the system's tasks can be defined as the vector $\mathcal{R} = [R_1, R_2, ..., R_N]$.

If the network manager decides to compute the task, R_n, of UE n locally then a local computation model can be defined by a local execution delay T_n^l and an energy consumption E_n^l:

$$T_n^l = \frac{D_n}{f_n^l}, \tag{3.1}$$

$$E_n^l = z_n D_n, \tag{3.2}$$

where z_n represents the energy consumption per CPU cycle and is set to $z_n = 10^{-27}(f_n^l)^2$ according to practical observations made in [32].

Based on the computation delay and energy consumption of task, R_n, a local cost can be calculated according to:

$$C_n^l = I_n^t T_n^l + I_n^e E_n^l. \tag{3.3}$$

If the network manager decides to offload the computation to the MEC server $m = \alpha_n$, then the offload computation model can be defined by an upload delay, $T_{n,t}^m$, an upload energy consumption, $E_{n,t}^m$, an offload execution delay, $T_{n,p}^m$, an idle energy consumption, $E_{n,p}^m$, a download delay, $T_{n,d}^m$ and its corresponding download energy consumption, $E_{n,d}^m$.

Firstly, the UE n must upload the input data, B_n from the task R_n to the decided MEC server m. This upload has an associated delay defined as:

$$T_{n,t}^m = \frac{B_n}{r_u}, \tag{3.4}$$

where r_u is the uplink rate of UE n computed according to Equation (2.30) from [27] and the distance, d_n^m, between the MEC server m and the UE n can be calculated using the euclidean norm between their locations.

$$d_n^m = ||loc_m - loc_n||_2 \tag{3.5}$$

This upload has an associated energy consumption:

$$E_{n,t}^m = P_n T_{n,t}^m = \frac{P_n B_n}{r_u}. \tag{3.6}$$

After the data is uploaded, the MEC server then computes the task resulting in an offload execution delay:

$$T_{n,p}^{m} = \frac{D_n}{f_m}, \tag{3.7}$$

where f_m is the amount of the MEC server, m, computation capacity, F_m, allocated to the offloaded task. To simplify the system the computation capacity of a MEC server is equally divided by all tasks offloaded to it:

$$f_m = \frac{F_m}{N_m}, \tag{3.8}$$

where N_m is the number of tasks offloaded to MEC server, m.

While the UE waits for the task to be computed, it stays idle, which has an associated energy consumption:

$$E_{n,p}^{m} = P_n^i T_{n,p}^{m} = \frac{P_n^i D_n}{f_m}. \tag{3.9}$$

Finally the computation results are downloaded to the UE n with an associated delay:

$$T_{n,d}^{m} = \frac{B_d}{r_d}, \tag{3.10}$$

where B_d is the size of the computation output and r_d is the download rate of UE n according to the Equation (2.31) from [27].

This download step has an associated energy consumption that can be calculated according to:

$$E_{n,d}^{m} = P_n^d T_{n,d}^{m}. \tag{3.11}$$

By taking into account the delays defined in Equations (3.4), (3.7) and (3.10), we can compute the total offload delay, T_n^m as the sum of all delays:

$$T_n^m = T_{n,t}^{m} + T_{n,p}^{m} + T_{n,d}^{m}. \tag{3.12}$$

The total energy consumption of offloading to the MEC server m, can be calculated by adding all energy consumptions defined in Equations (3.6), (3.9) and (3.11):

$$E_n^m = E_{n,t}^{m} + E_{n,p}^{m} + E_{n,d}^{m}. \tag{3.13}$$

Based on the computation delay and energy consumption of offloading task R_n to MEC server m, a cost can be calculated according to:

$$C_n^m = I_n^t T_n^m + I_n^e E_n^m. \tag{3.14}$$

The cost of the offloading decision $\alpha_n \in \{0, 1, ..., M\}$ can be computed according to:

$$C_n = \begin{cases} C_n^l & \alpha_n = 0 \\ C_n^m & \alpha_n \in \mathcal{M} \end{cases}. \tag{3.15}$$

The system's costs can be defined as the vector:

$$\mathcal{C} = [C_1, C_2, ..., C_N]. \tag{3.16}$$

The mean cost of the MEC system at each iteration can then be defined as:

$$C_{mean} = \frac{\sum_{n=1}^{N} C_n}{N}. \tag{3.17}$$

The worst-case cost of the MEC system at each iteration can then be defined as:

$$C_{max} = \max \mathcal{C}. \tag{3.18}$$

This corresponds to the UE that has the highest cost at each iteration.

3.1.2 Problem formulation

The offloading decision needs to be formulated in a way that can both optimize overall system performance while considering the worst-case serviced UE.

Given this, the weighted cost at each iteration can be defined as:

$$C = W_{mean} * C_{mean} + W_{max} * C_{max}. \tag{3.19}$$

By picking the importance weights of the costs that satisfy:

$$\begin{aligned} 0 \leq W_{mean} \leq 1, \\ 0 \leq W_{max} \leq 1, \\ W_{mean} + W_{max} = 1. \end{aligned}$$

The priorities of the system can be adjusted to satisfy both overall system performance and consider the worst-case cost of the system.

The optimization function can then be defined with the objective of finding the offloading decision vector $\mathcal{A} = [\alpha_1, \alpha_2, ..., \alpha_n]$ that minimizes the weighted average of the mean and max cost of the system at each iteration:

$$\begin{aligned} \min_{\mathcal{A}} \quad & C \\ \text{s.t.} \quad & C1 : \alpha_n \in \{0, 1, ..., M\}, \forall n \in \mathcal{N} \end{aligned}$$

3.1.3 Solution

Given the complex nature of the proposed problem and the lack of perfect knowledge of network conditions this thesis proposes to use a model-free DRL agent to manage the network. This means that the network manager does not have access to the transition probability, $P(s_{t+1}|s_t, a_t)$, nor the reward function $R(s, a)$ and must learn them by experimenting on the environment.

To do this the problem is formulated as an MDP, $< S, A, P, R >$:

- $S = \{s = (\mathcal{R})\}$ is the state space, which contains all requested tasks, $\mathcal{R}$;
- $A = \{a = (\mathcal{A})\}$ is the action space, which contains the offloading decision vector for all tasks, $\mathcal{A}$;
- $P : S \times A \times S \rightarrow [0, 1]$ is the transition probability distribution $P(s_{t+1}|s_t, a_t)$;
- $R = -C(s, a)$ is the reward function, which is defined as the inverse of the system cost, $C(s, a)$, given that the agent will be trying to maximize the reward.

At each time step, t, this network manager takes a state, s_t, makes a decision, a_t, that results in the state, s_{t+1}, and a reward r_t. The goal is then finding the policy, π, that maximizes the expected return, $V_\pi(s)$ as defined in Equation (2.4).

To achieve this, several DRL algorithms were explored: DQN, DDQN, Dueling DQN and A2C. Given the advantages presented in Section 2.2.9, the A2C algorithm was chosen. The pseudocode of the A2C algorithm is presented in Algorithm 1, as defined in [21]. To solve the exploration vs exploitation dilemma, the A2C algorithm uses Boltzmann exploration as defined in Section 2.3.2.

Algorithm 1 Advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter T = 0
// Assume thread-specific parameter vectors θ' and θ'_v
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
  Synchronize thread-specific parameters: $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$

$$R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$$

  for $i \in t - 1, ..., t_{start}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ' : $d\theta \leftarrow d\theta + \nabla_{\theta'} log_\pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$
    Accumulate gradients wrt θ'_v : $d\theta_v \leftarrow d\theta_v + d(R - V(s_i; \theta'_v))^2/d\theta'_v$
  end for
  Perform synchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

As a function approximator for $V(s_t|\theta_v)$ and $\pi(a_t|s_t;\theta)$ a single neural network with two dense hidden layers was used. The hidden layers consist of 256 neurons each using the $tanh$ activation function.

The input layer of this model consists of the flattened state vector, S. Since the state vector is a vector of length N, for the number of tasks and each task is described by a 5 variable vector, $R_n = [B_n, B_d, D_n, I_n^t, I_n^e]$, the total length of the input vector is $5 \times N$.

The output of the model consists of two parts. First a single linear output representing the value function, $V(s_t|\theta_v)$. Secondly, a softmax output for each action dimension with one entry per action, representing the probability of selecting the action, $\pi(a_t|s_t;\theta)$.

Since the agent must decide for each task (N) between local computation or offloading to one of the MEC servers (M), there are N softmax functions with $M + 1$ entries each, representing the possible actions. A diagram of the model is shown in Figure 3.2.

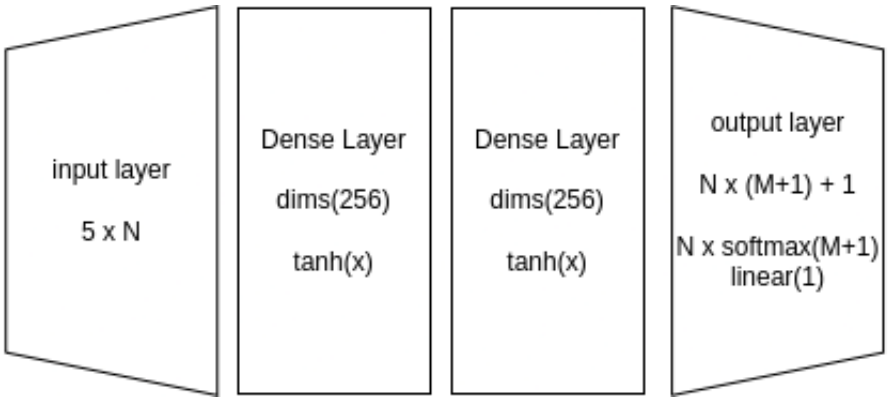

Figure 3.2: Model architecture.

3.2 Methods and tools

In order to implement a network manager capable of dealing with the proposed system, two main challenges need to be addressed: 1) the implementation of the DRL algorithms; 2) the implementation of the proposed system simulator. The programming language that was used to implement all algorithms and the simulator was Python. The reason behind this decision is that given its popularity as the second most used programming language overall [33], and the most used in the machine learning context [34], most of all popular DRL tools are written in Python.

As for the implementation of the DRL algorithms, two main tools were considered, Tensorflow 2.0 [35] with the Keras API layer and Pytorch [36]. Due to the higher popularity of Tensorflow in the reinforcement learning context, it was chosen as the neural network library used to write all DRL algorithms. To help with the distributed nature of some of the RL algorithms, the open-source library RLlib [37] was used.

The second major challenge is the implementation of the simulator. OpenAI Gym was introduced in [3] in order to standardize reinforcement learning environments and benchmarks. Due to its easy integration with Keras demonstrated in [38] and [39], it was chosen as the tool for implementing the simulation environment of the proposed system.

Chapter 4

Results

4.1 Baselines

In order to benchmark the various DRL algorithms and test the implemented simulator, several baseline algorithms were developed:

- Full Local: all UEs execute their tasks locally, in terms of the decision vector $\mathcal{A} = [0_1, 0_2, ..., 0_N]$. This represents the case where the system has no offloading capabilities.
- Random Offload: where the offloading decision is completely random, each task is either executed locally or offloaded to one of the available MEC servers randomly;
- Full nearest MEC: all UE tasks are offloaded to the nearest MEC server according to Equation (3.5). This represents the case of offloading to the nearest node without regarding their computation constraints and other offloading decisions. This baseline also requires knowledge of the network topology, which the agent does not have.

The plan is to create several system configurations of increasing complexity. The baseline algorithms will be used in order to benchmark the quality of trained agents. It is expected that the proposed network manager will surpass their performance in all situations by making intelligent decisions taking into account computation, battery, delay and communication constraints, which are ignored by the baselines.

4.2 Simple test

In order to put these baselines, the agent and the simulator to the test, a simple test case was devised. In this test, there are five UEs and five MEC servers. The UEs and MEC servers were randomly distributed in a 2D plane with $x \in [0, 200]$ and $y \in [0, 200]$, resulting in the distribution seen in Figure 4.1.

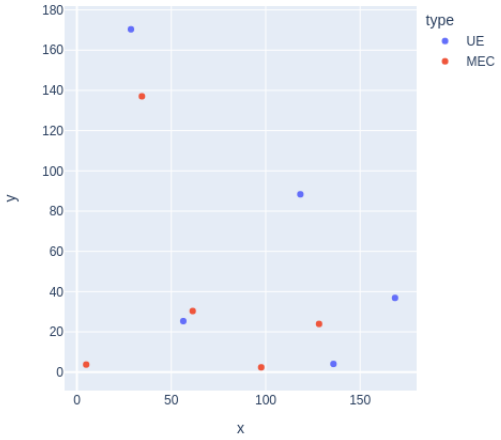

Figure 4.1: Example layout for the simple test.

For simplicity, all UEs and MEC servers are considered to have the same specifications. The system's hyper-parameters were set according to the values present in Table 4.1. In order to test this configuration, the task parameters, B_n, B_d and D_n were sampled from uniform distributions between $(300, 500)$ Kbits, $(10, 15)$ Kbits and $(900, 1100)$ Megacyles, respectively.

Variable	Value	Variable	Value
B	10×10^6	f_n^l	1×10^9
n	10	I_n^t	0.5
β	-4	I_n^e	0.5
$h_u l$	100	P_m	200
$h_d l$	100	P_n	500×10^{-3}
$g_u l$	1	P_n^i	100×10^{-3}
$g_d l$	1	P_n^d	200×10^{-3}
N_0	5×10^{-5}	F_m	5×10^9
W_{mean}	1	W_{max}	0

Table 4.1: System hyper-parameters.

The agent described in Section 3.1.3 was trained with the following hyper-parameters:

Variable	Value
episode length	10
learning rate, α	0.0001
discount factor, γ	0.99
batch size	200

Table 4.2: Training hyper-parameters.

Each algorithm ran for 100 episodes and an average per episode of offloading decisions is shown in Table 4.3.

Algorithm	Average Reward (R)	Reduction
Full Local	-49.47	96.65%
Random Offload	-9.68	82.89%
Full nearest MEC	-2.18	24.06%
Agent (A2C)	-1.66	-

Table 4.3: Average Reward (R) over 100 iterations.

As expected, the Full Local baseline performed the worst given that it does not make use of the MEC servers computing capabilities, leading to an increased cost due to increased energy consumption and delay of the tasks. The Random Offload baseline is the second worst, given that although it gains from offloading to MEC servers, it does so without any consideration for the system's configuration. From the baselines, Full nearest MEC performed the best since it takes into consideration the distance between the UE and the MEC server. However, it comes with the disadvantage of not only requiring knowledge of the network topology (positions of UEs and MECs) but also not taking into account anything else.

As expected, the proposed agent outperformed the baselines in all situations, which demonstrates that the agent is able to learn not only the topology of the network by trying out different offloading decisions but also make intelligent decisions based on computation, battery, delay and communication constraints ignored by the baselines.

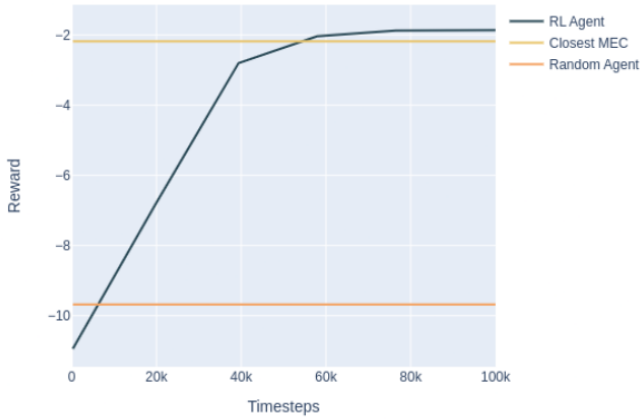

Figure 4.2: Agent reward while training.

As shown in Figure 4.2, the agent starts with an equivalent reward to the Random Offload, which is expected since the agent has not yet learned the environment. As the agent interacts with the environment making offloading decisions (actions), it receives state transitions and rewards. This allows it to quickly learn the system and the reward increases rapidly in the first iterations. After about 55K steps in the environment, the agent surpasses the best offloading baseline, Full nearest MEC. As expected, the reward starts to stabilize as the agent learns the final details of the system increasing slightly the final performance.

4.3 Scalability and Data Efficiency

In this section, the scalability and data efficiency of our agent is explored. Scalability in this context can be defined as the ability of the agent to learn and outperform the baselines in higher complexity problems.

Data efficiency in this context can be defined as the ability of the agent to learn in as few possible steps as possible. While it is expected that the agent takes longer to learn in more complex problems, the complexity of the underlying policy should not grow linearly with the state and action spaces. The reason for this is that it is expected that the agent is able to generalize concepts instead of brute forcing a solution for each problem set.

Since our state space is defined as the requested tasks at each step and each task is defined by a set of 5 parameters, the complexity of the state space grows linearly with the amount of UEs, *N*. The size of the state space should be $N \times 5$.

On the other side, the action space is defined as the offloading decision of each task. Given that each task can be computed locally or offloaded to one of the MEC servers, *M*, this gives us the option of $M + 1$ actions per task. This results in the action space growing exponentially with the amount of UEs, *N*, since there are $(M + 1)^N$ possible actions at each decision step.

With this in mind, the stability and data efficieny of the system can be tested by setting the system's hyper-parameters and amount of MEC servers to be the same as in the simple test case and create two new test scenarios by increasing the number of UEs to 10 and 20, respectively.

The UEs and MEC servers were randomly distributed in a 2D plane with $x \in [0, 200]$ and $y \in [0, 200]$, resulting in the distribution seen in Figure 4.3 and Figure 4.4.

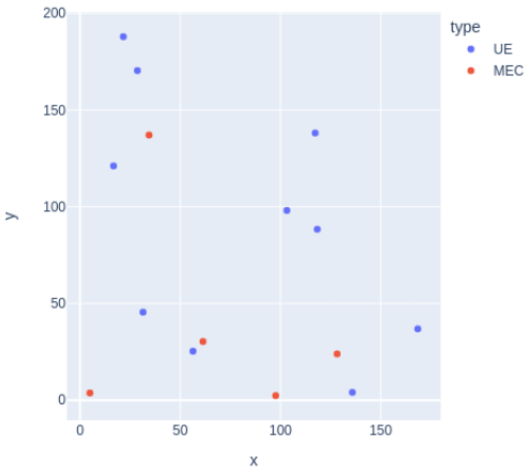

Figure 4.3: Test with 10 UEs.

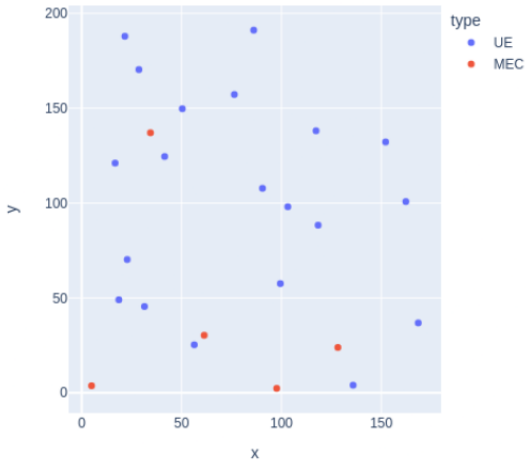

Figure 4.4: Test with 20 UEs.

Each algorithm ran for 100 episodes and an average per episode of offloading decisions is shown in Table 4.4 and Table 4.5.

Algorithm	Average Reward (R)	Reduction
Full Local	-49.50	94.40%
Random Offload	-10.32	73.14%
Full nearest MEC	-3.56	22.17%
Agent (A2C)	-2.77	-

Table 4.4: Average Reward (R) over 100 episodes with 10 UEs

Algorithm	Average Reward (R)	Reduction
Full Local	-49.49	90.42%
Random Offload	-11.67	59.36%
Full nearest MEC	-7.23	34.38%
Agent (A2C)	-4.74	-

Table 4.5: Average Reward (R) over 100 episodes with 20 UEs

As expected, the reward of the Full Local baseline stays almost the same between test environments. This happens because these tests set W_{mean} to 1 and W_{max} to 0, so the cost is only composed by C_{mean}, defined in Equation 3.17. Since C_{mean} is the average cost of each UE's task, this should not scale with the amount of UEs.

The same cannot be said of the other baselines and the agent. Since the number of MECs stays the same and the number of tasks increases, the overall cost of the system increases with the amount of UEs. This represents the computation constraint of the MEC servers.

These results demonstrate that the agent scales with the complexity of the system.

In order to study the data efficiency of the agent, the agent's performance while training must be analysed.

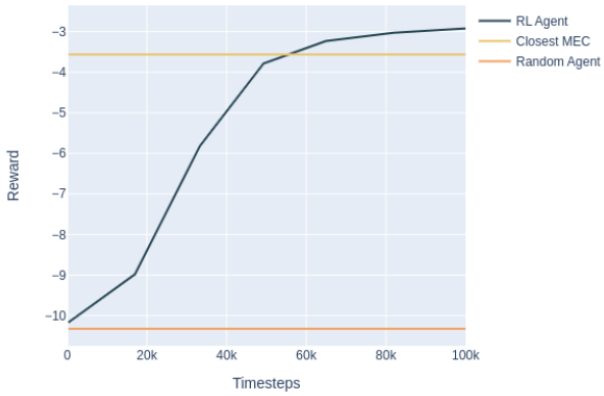

Figure 4.5: Training with 10 UEs.

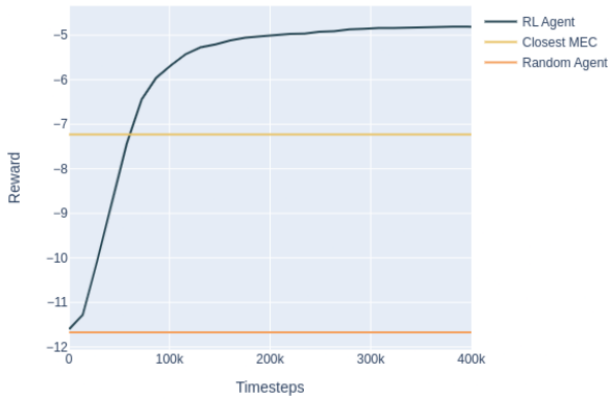

Figure 4.6: Training with 20 UEs.

The same trend as the simple test can be observed with the two new tests in Figure 4.5 and Figure 4.6. The agent starts with an equivalent reward to the Random Offload, then it quickly learns the system and

the reward increases rapidly in the first iterations. After about 55K steps in both environments, the agent surpasses the best offloading baseline, Full nearest MEC. As expected, the reward starts to stabilize as the agent learns the final details of the system increasing slightly the final performance.

This confirms that the agent is data efficient, meaning that even when the system's state space increases linearly and the action space increases exponentially, the agent is able to generalize concepts achieving a good performance. It does this with almost the same number of interactions with the system, proving it is finding a strategy without brute forcing a solution.

4.4 Robustness and Stability

In this section, the robustness and stability of our agent is explored. Robustness in this context can be defined as the ability of the agent to learn and outperform the baselines in environments with different network conditions and heterogeneous computation capabilities.

Stability in this context can be defined has the ability of the agent to not diverge or collapse while training. This means that its performance should improve as it interacts with the environment, and after learning a strategy it should not forget it, collapsing the system's performance.

In order to test the system's robustness to changing network conditions, a new test case is devised by setting the system hyper-parameters to the values present in Table 4.6.

Variable	Value	Variable	Value
B	10×10^6	f_n^l	0.75×10^9
n	10	I_n^t	0.25
β	-4	I_n^e	0.75
$h_u l$	50	P_m	100
$h_d l$	50	P_n	250×10^{-3}
$g_u l$	0.5	P_n^i	50×10^{-3}
$g_d l$	0.5	P_n^d	100×10^{-3}
N_0	3×10^{-5}	F_m	2.5×10^9
W_{mean}	1	W_{max}	0

Table 4.6: New system hyper-parameters.

By setting the number of UEs to 10 and the number of MECs to 5, they were randomly distributed in a 2D plane with $x \in [0, 200]$ and $y \in [0, 200]$, resulting in the distribution seen in Figure 4.7.

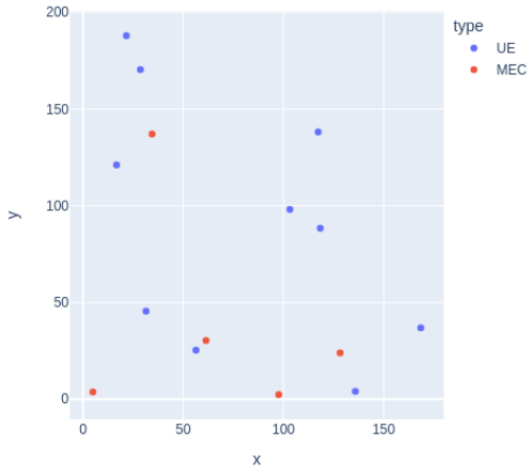

Figure 4.7: Robustness test layout.

Each algorithm ran for 100 episodes and an average per episode of offloading decisions is shown in Table 4.7.

Algorithm	Average Reward (R)	Reduction
Full Local	-31.31	84.92%
Random Offload	-9.14	48.35%
Full nearest MEC	-6.81	30.62%
Agent (A2C)	-4.72	-

Table 4.7: Average Reward (R) over 100 episodes of robustness test.

As expected, the reward values of all the algorithms differ from previous tests since the system hyper-parameters that are used to calculate the reward function are different.

The same cost reduction trends can be observed in this new environment, showing that the agent is able to learn the system's behaviour independently of the network conditions.

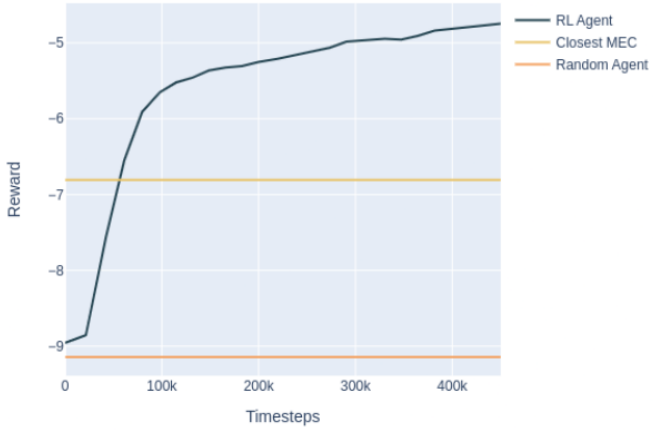

Figure 4.8: Training in robustness test.

As shown in Figure 4.8, the agent's performance while training follows a similar trend to previous tests. The agent starts with an equivalent reward to the Random Offload, then it quickly learns the system and the reward increases rapidly in the first iterations. After about 55K steps, the agent surpasses the best offloading baseline, Full nearest MEC. As expected, the reward starts to stabilize as the agent learns the final details of the system, slightly increasing the final performance.

The other component of robustness that must be tested is robustness to heterogeneous computation capabilities of UEs and MEC servers. To test this, a new test case is devised by resetting the system hyper-parameters to the values present in Table 4.1 but instead of setting f_n^l and F_m to a fixed value, they are set to a value randomly sampled from a set of values, $f = \{0.25, 0.5, 0.75, 1\} \times 10^9$ and $F = \{2.5, 5, 7, 10\} \times 10^9$, respectively.

By setting the number of UEs to 10 and the number of MECs to 5, they were randomly distributed in a 2D plane with $x \in [0, 200]$ and $y \in [0, 200]$, resulting in the distribution seen in Figure 4.9.

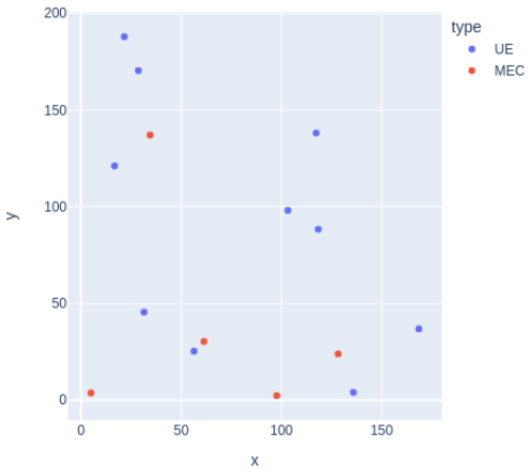

Figure 4.9: Heterogeneous test layout.

Each algorithm ran for 100 episodes and an average per episode of offloading decisions is shown in Table 4.8.

Algorithm	Average Reward (R)	Reduction
Full Local	-33.56	89.55%
Random Offload	-8.80	60.16%
Full nearest MEC	-5.04	30.39%
Agent (A2C)	-3.51	-

Table 4.8: Average Reward (R) over 100 episodes of heterogeneous test.

As presumed, the reward values of all the algorithms differ from previous tests since the system computation capabilities that are used to calculate the reward function are different.

Although the ordering of the algorithm performances stays the same in this test case, the cost reduction of our agent in comparison to the Full nearest MEC is higher. With a reported 30.39% improvement,

while the equivelant homogeneous test had an improvement of 22.17%. This makes sense given that the Full nearest MEC only takes into account the distance of UEs to MECs and not their computation capabilities. This, in turn, leads to worse load balancing when the computation capabilities of the UEs and MECs are not all the same.

The agent on the other hand is able to learn the system's network topology and make offloading decisions that better load balance the required tasks.

By analysing all training curves, it is clear that the agent is stable and its reward does not diverge or collapse while training, even when ran for millions of iterations.

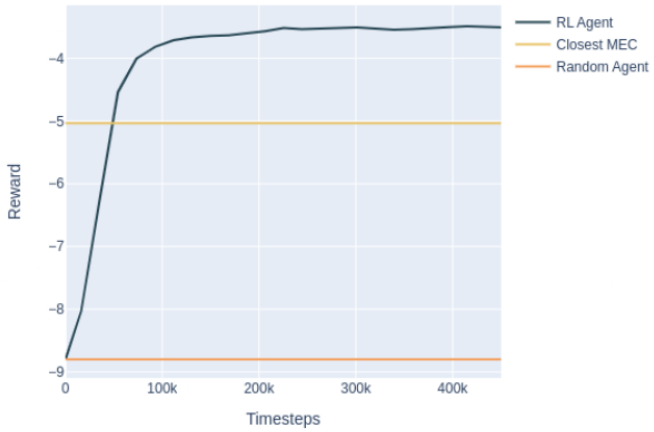

Figure 4.10: Training in heterogeneous environment.

As shown in Figure 4.10, the agent's performance while training follows a similar trend to previous tests. The agent starts with an equivalent reward to the Random Offload, then it quickly learns the system and the reward increases rapidly in the first iterations. After about 50K steps, the agent surpasses the best offloading baseline, Full nearest MEC. As expected, the reward starts to stabilize as the agent learns the final details of the system increasing slightly the final performance.

4.5 Reward function weights

This section focuses on studying the effect of the cost function importance weights on the agent's overall and worst-case performance.

To test this, five new test cases were created by setting the system hyper-parameters to the values present in Table 4.1 but instead of setting W_{mean} to 1 and W_{max} to 0, they are set to $(1, 0.75, 0.5, 0.25, 0)$ and $(0, 0.25, 0.5, 0.75, 1)$, respectively.

By setting the number of UEs to 10 and the number of MECs to 5, they were randomly distributed in a 2D plane with $x \in [0, 200]$ and $y \in [0, 200]$, resulting in the distribution seen in Figure 4.11.

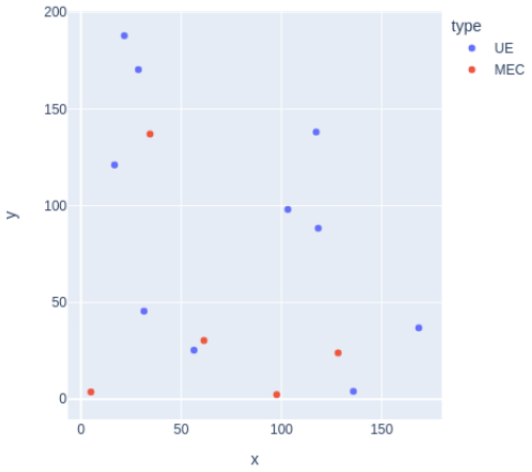

Figure 4.11: Weight test layout.

Each combination of the cost function ran for 100 episodes and an average per episode of offloading decisions is shown in Table 4.9.

$\{W_{mean}, W_{max}\}$	Overall Cost	Worst Cost	Difference
$\{1, 0\}$	2.72	4.04	1.32
$\{0.75, 0.25\}$	2.97	3.84	0.87
$\{0.5, 0.5\}$	3.22	3.53	0.31
$\{0.25, 0.75\}$	3.52	3.81	0.29
$\{0, 1\}$	3.81	3.78	-0.03

Table 4.9: Agent performance over 100 episodes of weight tests.

As the weights shift from prioritizing overall performance to prioritizing worst-case performance, the agent's overall cost increases while the worst-case cost decreases. While the agent outperforms the best baseline with every weight distribution, it learns to make offloading decisions that prioritize the worst-case performance at the expense of overall system performance. With this in mind, the process of picking the best weight distribution would involve trying to determine the maximum cost allowed per UE and shifting the weight distribution in favour of W_{max}, trying to maximize W_{mean} while still meeting the maximum allowed cost.

Chapter 5

Conclusions

5.1 Summary

As presented in Chapter 2 Section 2.1 several architectures have been proposed in the MEC context that present complex optimization problems of managing how to distribute tasks between UEs and MEC servers. Due to the high dimensional complexity and uncertain networking conditions, classical offline optimization algorithms fail to effectively manage these types of problems. As a solution to this, a review of the state of the art in DRL and its application to these MEC challenges was made in Chapter 2 Sections 2.2, 2.3 and 2.4. Based on the work done by papers [27] and [31] a more complete system that takes into account the possibility of offloading between a network of heterogeneous UEs to a network of heterogeneous MEC servers is proposed. As far as the candidate knows, this is the first time this extended optimization problem is addressed. In order to deal with the increased complexity of taking into account computation, battery, delay and communication constraints in an *N* to *N* problem, a network manager agent is proposed in Section 3.1.3. In order to evaluate the performance of the proposed agent, several baselines are described in Section 4.1. Finally, the agent's capabilities are put to the test in Sections 4.2, 4.3, 4.4 and 4.5.

In Section 4.2, a simple test case is used to demonstrate the simulator, baselines and the agent's capacity to learn. As expected, the Full Local baseline performs the worst, then the Random Offload baseline and finally the Full nearest MEC. The proposed agent is shown to learn, overperforming the baselines by 96.65%, 82.89% and 24.06%, respectively. The agent achieves this while not requiring any information of the network topology by simply interacting with the environment.

In Section 4.3, the agent's scalability and data efficiency is put to the test by increasing the complexity of the system with two new tests. Given that the system complexity is tightly related with the number of UEs in the network, the agent was tested in an environment with 10 and 20 UEs. As showcased, the agent learned to overperform the best baseline with a 22.17% and 34.38% improvement, respectively, showcasing its ability to scale with problem complexity. The agent also achieved this in 55K steps, demonstrating it's data efficiency.

Section 4.4 serves to test the agent's robustness and stability. In order to test robustness, two

new test cases were implemented. First, the agent's robustness to changing network conditions was tested and proven by overperforming the baseline with a 30.62% improvement. Secondly, the agent's robustness to an environment with heterogeneous computation capabilities of UEs and MEC servers was put to the test. The agent not only was able to learn and overperform the best baseline, but it did so in a greater faction than with simpler tests, demonstrating a reduction of 30.39% over the Full nearest MEC baseline, compared with the 22.17% of the same test without heterogeneous computation capacity. Stability was analysed by looking at the different learning curves of the agent. Over all previous tests, the agent was shown to be stable independently of the changing conditions, never showing regression or collapse of its reward after it starts learning the environment.

In the end, Section 4.5 presents the effects of changing the importance weight distribution from the overall performance to the worst-case performance. The adjustability of the reward function is shown to allow the improvement of the worst-case cost at the expense of the overall system cost.

All the simulator, baselines and agent code, as well as environment configurations, can be found in the following code repository: https://github.com/Carlos-Marques/rl-MEC-scheduler.

5.2 Future Work

Future steps of research should include:

- Expanding the proposed simulator to include more dynamic environments, implementing the ability to change the following as simulation progresses:
  - UE locations to simulate mobility;
  - The number of UEs and MEC servers;
  - Network conditions.
- Creating a more complete test set, testing environments for specific scenarios. For example, a Smart city environment or an autonomous vehicle environment.
- Exploring the use of this orchestration agent with choreography algorithms like the one proposed in [30], by simulating an environment where several groups of MEC servers would be orchestrated by agents like the one proposed in this work, while choreographing with each other.
- Exploring the use of model based RL algorithms to take advantage of the extensive domain knowledge of telecommunication physics. By leveraging this knowledge, the agent could converge to a better solution more quickly without having to learn this through experimentation.

Bibliography

[1] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Computer Networks*, vol. 182, p. 107496, 2020.

[2] M. Patel, B. Naughton, C. Chan, N. Sprecher, S. Abeta, A. Neal, *et al.*, "Mobile-edge computing introductory technical white paper," *White paper, mobile-edge computing (MEC) industry initiative*, vol. 29, pp. 854–864, 2014.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[4] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[5] G. A. McGilvary, A. Barker, and M. Atkinson, "Ad hoc cloud computing," in *2015 IEEE 8th International Conference on Cloud Computing*, pp. 1063–1068, 2015.

[6] O. Consortium, "Openfog reference architecture for fog computing." Online, 2017. Last access on 28/05/2021.

[7] F. Lobillo, Z. Becvar, M. A. Puente, P. Mach, F. Lo Presti, F. Gambetti, M. Goldhamer, J. Vidal, A. K. Widiawan, and E. Calvanesse, "An architecture for mobile computation offloading on cloud-enabled lte small cells," in *2014 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pp. 1–6, 2014.

[8] S. Wang, G.-H. Tu, R. Ganti, T. He, K. Leung, H. Tripp, K. Warr, and M. Zafer, "Mobile micro-cloud: Application classification, mapping, and deployment," in *Proc. Annual Fall Meeting of ITA (AMITA)*, 2013.

[9] K. Wang, M. Shen, J. Cho, A. Banerjee, J. K. V. der Merwe, and K. Webb, "MobiScud: a fast moving personal cloud in the mobile network," 2015.

[10] A. Aissioui, A. Ksentini, and A. Gueroui, "An efficient elastic distributed sdn controller for follow-me cloud," in *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 876–881, 2015.

[11] J. Liu, T. Zhao, S. Zhou, Y. Cheng, and Z. Niu, "Concert: a cloud-based architecture for next-generation cellular systems," *IEEE Wireless Communications*, vol. 21, no. 6, pp. 14–22, 2014.

[12] H. E. Project, "Small cells coordination for multi-tenancy and edge services (sesam).." Online, 2015. Last access on 28/05/2021.

[13] J. Baek and G. Kaddoum, "Heterogeneous task offloading and resource allocations via deep recurrent reinforcement learning in partial observable multifog networks," *IEEE Internet of Things Journal*, vol. 8, no. 2, pp. 1041–1056, 2021.

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

[15] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1, pp. 99–134, 1998.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb 2015.

[17] A. Choudhar, "A hands-on introduction to deep q-learning using openai gym in python." Online, 2019. Last access on 28/05/2021.

[18] H. v. Hasselt, "Double q-learning," in *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2*, NIPS'10, (Red Hook, NY, USA), p. 2613–2621, Curran Associates Inc., 2010.

[19] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[20] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015.

[21] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016.

[22] C. Yoon, "Understanding actor critic methods and a2c." Online, 2019. Last access on 31/05/2021.

[23] Y. Wu, E. Mansimov, S. Liao, R. B. Grosse, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," *CoRR*, vol. abs/1708.05144, 2017.

[24] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," *CoRR*, vol. abs/1611.05763, 2016.

[25] N. Cesa-Bianchi, C. Gentile, G. Lugosi, and G. Neu, "Boltzmann exploration done right," *CoRR*, vol. abs/1705.10257, 2017.

[26] X. Qiu, L. Liu, W. Chen, Z. Hong, and Z. Zheng, "Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 8, pp. 8050–8062, 2019.

[27] Y. Gong, C. Lv, S. Cao, L. Yan, and H. Wang, "Deep learning-based computation offloading with energy and performance optimization," *EURASIP Journal on Wireless Communications and Networking*, vol. 2020, p. 69, Mar 2020.

[28] S. Yu, X. Wang, and R. Langar, "Computation offloading for mobile edge computing: A deep learning approach," in *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pp. 1–6, 2017.

[29] J.-y. Baek, G. Kaddoum, S. Garg, K. Kaur, and V. Gravel, "Managing fog networks using reinforcement learning based load balancing algorithm," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–7, 2019.

[30] L. P. de Matos Morgado Ferreira, "Fog computing task offloading optimization based on deep reinforcement learning," Master's thesis, Instituto Superior Técnico, 2021.

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

[32] Y. Wen, W. Zhang, and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones," in *2012 Proceedings IEEE INFOCOM*, pp. 2716–2720, 2012.

[33] GitHub, "The 2020 state of the octoverse." Online, 2020. Last access on 3/06/2021.

[34] GitHub, "The state of the octoverse: machine learning." Online, 2019. Last access on 3/06/2021.

[35] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[37] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2018.

[38] M. Plappert, "Deep reinforcement learning for keras." Online, 2016. Last access on 28/05/2021.

[39] T. McNally, “Deep reinforcement learning for tensorflow 2 keras.” Online, 2019. Last access on 31/05/2021.