NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

PEDRO ALEXANDRE FRANCISCO CAPELO
Degree in Electrical and Computers Engineering Sciences

AI GYM FOR NETWORKS

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

AI GYM FOR NETWORKS

PEDRO ALEXANDRE FRANCISCO CAPELO

Degree in Electrical and Computers Engineering Sciences

Adviser: Pedro Miguel Figueiredo Amaral
Assistant Professor, NOVA University Lisbon

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2022

AI Gym for Networks

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.15) [1].

To my family and friends.

Acknowledgements

First of all, I would like to thank my family. Specifically, my mother and my father for providing me with education, support and conditions to be the person that I am today. They are the principal reason on why I was able to achieve this goal and many others in my life.

Furthermore, I would like to thank my adviser, Dr. Pedro Amaral, for suggesting the thesis topic and for providing the needs to develop the wanted project, by giving me the material, the helpful advices and assistance.

I would also like to thanks NOVA School of Science and Technology, NOVA University of Lisbon and all the teachers who crossed my academic path, for helping me to build the bases to become a better human and professional.

Finally, I would like to thank my colleagues, Alexandre Brito and João Faria, who have become great friends and helped me during the past 5 years. They shared with me the struggles and victories that an academic engineer path brings.

"

Recomeça....
Se puderes
Sem angústia
E sem pressa.
E os passos que deres,
Nesse caminho duro
Do futuro
Dá-os em liberdade.
Enquanto não alcances
Não descanses.
De nenhum fruto queiras só metade.
E, nunca saciado,
Vai colhendo ilusões sucessivas no pomar.
Sempre a sonhar e vendo
O logro da aventura.
És homem, não te esqueças!
Só é tua a loucura
Onde, com lucidez, te reconheças...
" (Sísifo, Miguel Torga)

Abstract

5G Networks are delivering better services and connecting more devices, but at the same time are becoming more complex.

Problems like resource management and control optimization are increasingly dynamic and difficult to model making it very hard to use traditional model-based optimization techniques. Artificial Intelligence (AI) explores techniques such as Deep Reinforcement Learning (DRL), which uses the interaction between the agent and the environment to learn what action to take to obtain the best possible result.

Researchers usually need to create and develop a simulation environment for their scenario of interest to be able to experiment with DRL algorithms. This takes a large amount of time from the research process, while the lack of a common environment makes it difficult to compare algorithms.

The proposed solution aims to fill this gap by creating a tool that facilitates the setting up of DRL training environments for network scenarios. The developed tool uses three open source software, the Containernet to simulate the connections between devices, the Ryu Controller as the Software Defined Network Controller, and OpenAI Gym which is responsible for setting up the communication between the environment and the DRL agent.

With the project developed during the thesis, the users will be capable of creating more scenarios in a short period, opening space to set up different environments, solving various problems as well as providing a common environment where other Agents can be compared.

The developed software is used to compare the performance of several DRL agents in two different network control problems: routing and network slice admission control. A novel DRL based solution is used in the case of network slice admission control that jointly optimizes the admission and the placement of traffic of a network slice in the physical resources.

Keywords: 5G Networks, Deep Reinforcement Learning, Containernet, Ryu Controller

Resumo

As redes 5G oferecem melhores serviços e conectam mais dispositivos, fazendo com que se tornem mais complexas e difíceis de gerir.

Problemas como a gestão de recursos e a otimização de controlo são cada vez mais dinâmicos e difíceis de modelar, o que torna difícil usar soluções de optimização baseadas em modelos tradicionais. A Inteligência Artificial (IA) explora técnicas como Deep Reinforcement Learning que utiliza a interação entre o agente e o ambiente para aprender qual a ação a ter para obter o melhor resultado possível.

Normalmente, os investigadores precisam de criar e desenvolver um ambiente de simulação para poder estudar os algoritmos DRL e a sua interação com o cenário de interesse. A criação de ambientes a partir do zero retira tempo indispensável para a pesquisa em si, e a falta de ambientes de treino comuns torna difícil a comparação dos algoritmos.

A solução proposta foca-se em preencher esta lacuna criando uma ferramenta que facilite a configuração de ambientes de treino DRL para cenários de rede. A ferramenta desenvolvida utiliza três softwares open source, o Containernet para simular as conexões entre os dispositivos, o Ryu Controller como Software Defined Network Controller e o OpenAI Gym que é responsável por configurar a comunicação entre o ambiente e o agente DRL.

Através do projeto desenvolvido, os utilizadores serão capazes de criar mais cenários em um curto período, abrindo espaço para configurar diferentes ambientes e resolver diferentes problemas, bem como fornecer um ambiente comum onde diferentes Agentes podem ser comparados.

O software desenvolvido foi usado para comparar o desempenho de vários agentes DRL em dois problemas diferentes de controlo de rede, nomeadamente, roteamento e controlo de admissão de slices na rede. Uma solução baseada em DRL é usada no caso do controlo de admissão de slices na rede que otimiza conjuntamente a admissão e a colocação de tráfego de uma slice na rede nos recursos físicos da mesma.

Palavras-chave: Redes 5G, Deep Reinforcement Learning, Containernet, Ryu Controller

Contents

List of Figures

List of Tables

Acronyms

3GPP	3rd Generation Partnership Project *(p. 24)*
AI	Artificial Intelligence *(pp. vi, 2–4, 15, 19)*
AMF	Access and Mobility Management Function *(p. 10)*
ANN	Artificial Neural Network *(pp. 18, 50)*
API	Application Programming Interface *(pp. 10, 11, 13, 23, 27)*
AUSF	Authentication Server Function *(p. 10)*
BER	Bit Error Rate *(p. 1)*
BS	Base Station *(pp. 1, 7, 8, 42, 46)*
BWP	Bandwidth Part *(p. 24)*
C	Control Plane *(p. 9)*
CN	Core Network *(pp. 6–8, 10, 11, 55)*
CNF	Core Network Function *(p. 7)*
CNN	Convolutional Neural Network *(p. 18)*
CP	Control Plane *(pp. 10, 11, 14)*
CS	Computing Station *(p. 46)*
CU	Central Units *(p. 9)*
DDoS	Distributed Denial-of-Services *(p. 25)*
DDQL	Double Deep Q-Learning *(pp. 21, 22)*
DL	Deep Learning *(pp. 2, 4, 18, 19)*
DN	Data Network *(p. 10)*
DNN	Deep Neural Network *(p. 20)*
DQN	Deep Q-Network *(pp. 20–22, 37, 40, 45, 51)*
DRL	Deep Reinforcement Learning *(pp. vi, 2–4, 19, 24–26, 34–38, 40, 42, 45, 47, 48, 50, 51, 53, 55, 56)*
DU	Distributed Units *(p. 9)*

E2E	end-to-end *(pp. 6, 7, 10, 24)*
eMBB	Enhanced Mobile Broadband *(p. 6)*
en-gNB	Evolved Universal Mobile Telecommunications System (UMTS) Terrestrial Radio Access -NR-gNB *(p. 8)*
eNB	evolved NodeB *(p. 8)*
EPC	Evolved Packet Core *(pp. 7, 8, 24)*
gNB	next-generation NodeB *(pp. 7–9)*
GPU	Graphics Processing Unit *(p. 34)*
HTTP2	Hypertext Transfer Protocol Version 2 *(p. 11)*
IP	Internet Protocol *(pp. 7, 10)*
JSON	JavaScript Object Notation *(pp. 28, 30)*
KPI	Key Performance Indicator *(p. 6)*
LC-PD	Logically Centralized-Physically Distributed *(p. 14)*
LTE	Long-Term Evolution *(pp. 6–8)*
MAC	Media Access Control *(pp. 24, 28)*
MANO	Management and Orchestration *(p. 11)*
MDP	Markov Decision Process *(pp. 17, 18, 21)*
MECS	Mobile Edge Computing Station *(pp. 45, 46)*
MIMO	Multiple Input Multiple Output *(p. 1)*
ML	Machine Learning *(pp. 2, 15, 16, 55)*
mMTC	Machine Type Communications *(p. 6)*
MN	Master Node *(p. 8)*
NF	Network Function *(pp. 1, 7, 10, 11)*
NFV	Network Function Virtualization *(pp. 1, 4, 6, 12, 24, 31)*
ng-eNB	next generation eNB *(pp. 8, 9)*
NR	New Radio *(pp. 6–9, 24)*
NSA	Non-Standalone Architecture *(pp. 6, 24)*
NSSF	Network Slice Selection Function *(p. 10)*
OFDMA	Orthogonal Frequency-Division Multiple Access *(p. 24)*
OSI	Open Systems Interconnection *(p. 2)*

PCF	Policy Control Function *(p. 10)*
PDU	Protocol Data Unit *(p. 10)*
QoS	Quality of Service *(pp. 1, 10, 24, 55)*
RAN	Radio Access Network *(pp. 6, 7, 9, 10, 23, 24, 55)*
RESTfull	Representational State Transfer Application Programming Interface *(p. 11)*
RL	Reinforcement Learning *(pp. 2–4, 16, 19, 24, 25, 34)*
RNF	Radio Network Function *(p. 7)*
RNN	Recurrent Neural Network *(pp. 18, 19)*
SBA	Service Based Architecture *(p. 11)*
SBI	Service Based Interface *(p. 11)*
SDN	Software-defined networking *(pp. 3, 4, 6, 11, 13–15, 24–26, 55)*
SMF	Session Management Function *(p. 10)*
SN	Secondary Node *(p. 8)*
TD	Temporal-Difference *(p. 22)*
TDMA	Time-Division Multiple Access *(p. 24)*
U	User Plane *(p. 9)*
UDM	Unified Data Management *(p. 10)*
UE	User Equipments *(pp. 1, 8, 10)*
UL	UpLink *(p. 24)*
UP	User Plane *(p. 10)*
UPF	User Plane Function *(p. 10)*
URLLC	Ultra-Reliable and Low Latency Communications *(p. 6)*
VM	Virtual Machine *(pp. 1, 11, 12)*
VNF	Virtual Network Function *(pp. 12, 24)*

1 Introduction

1.1 Motivation

Over the years, we have witnessed a growth in the number of connected mobile devices and the transmitted data in modern networks. Furthermore, the necessity to support new use cases and services, such as the Internet of Things, Industry 4.0, and Connected Vehicles, that were not possible to implement with older wireless network generations, led to the investigation and implementation of future generation wireless networks.

5G networks and beyond have numerous characteristics. Starting with the physical network advancements, technologies like Multiple Input Multiple Output (MIMO) allow each Base Station (BS) to transmit high-speed data streams to multiple User Equipments (UE)s, simultaneously, reducing the Bit Error Rate (BER), minimizing fading effects, and offering high Quality of Service (QoS) by increasing spectral efficiency and data rates. Another technology, known as millimeter-wave (mm-wave) communications, provides a higher transmission rate, more immunity to interference, and enables multiple short distance usages because of its higher frequencies [2].

An significant characteristic of 5th generation technology is network softwarization and Network Function Virtualization (NFV). These technologies bring the advantage of not depending on dedicated hardware. NFV brings the possibility of virtualizing network services by packing the services in Virtual Machine (VM)s or containers on commodity hardware.

These technologies provide significant benefits to handle the challenges discussed above, such as the possibility to run multiple functions on a single server, leading to use fewer physical hardware and to have a cost reduction. An extra benefit is the flexibility to run Network Function (NF)s in different services with the capacity to change their locations when demand changes, accelerating the delivery of services [3].

5G networks with these characteristics have more UEs and BSs of different types and different QoS requirements, creating large-scale, heterogeneous, and decentralized networks, with a high level of complexity, dynamism, and uncertainty, that difficults the goal of optimizing these networks in aspects such as data rate, energy consumption, latency, resources, management, and policy definitions.

There are conventional approaches, such as convex optimization and linear programming that can be used for optimizing and planning networks, but due to the complexity, scalability, and uncertainty of 5G networks modelling the problems is very complex. These approaches have intractable complexity when facing challenging problems to model and are not easily applicable in highly dynamic scenarios [4]. An alternative is the use AI based approaches for network control, especially in the Reinforcement Learning (RL) domain, that are model-free and are demonstrating great potential for control based problems in communication systems.

RL is a class of Machine Learning (ML) that is suited for handling problems related to real-time dynamic-decisions-making. The advantage of RL is that it does not rely on a system mathematical model and it can adapt and learn in dynamic conditions, making It useful in scenarios such as 5G networks. RL algorithms can update decision policies to obtain better systems performance, using feedback based on previous decisions. Solutions based on RL can be an alternative to solve resource management problems in extensive networks.

When the complexity and the scale of networks increases, RL algorithms have a slow convergence speed because of the increasing number of space and action spaces. Another problem is related to the amount of stored information, for example, state-action pair information, making the amount of data too big, increasing the problem of finding optimal policies in a reasonable time. A better solution than RL is DRL, which belongs to the AI field and can be described as a combination of RL with Deep Learning (DL) [5].

The use of DRL allows handling larger scale dynamic systems, by eliminating the need to store action-state pairs [6]. DRL has been used to optimize and solve issues related to 5G networks, as data offloading, data rate control, dynamic network access and network security.

The paper [6] presents a survey focus on the applications of DRL in communications and networking. The applications included subjects related with dynamic network access, data rate control, wireless caching, data offloading, traffic routing, resource sharing and others.

DRL agents need to be trained to achieve a good performance in a specific scenario. The problem is the necessity of having a high volume of trial and error interactions with the environment to learn an useful policy. The simulators' utilization leads to better results in a cost-effective way. Researchers need to use simulations or mathematical models to obtain the results from a given action on the environment.

On other hand, there are many tools available that can be use to simulate network environments. The issue is that each of these tools is for specific scenarios or specific Open Systems Interconnection (OSI) Layers. The complexity of each layer and the different types of architectures adopted by them, make it difficult to find simulators capable of replicating all the network's layers behaviour, including the links between them.

This leads to a myriad of simulation implementations in the research literature, with each author building its custom environment to train a specific DRL for a specific scenario. Tools that can streamline this process are needed in order to expedite experimentation, but also to more easily provide , common environment implementations for benchmarking DRL algorithms. The lack of a tool that facilitates and speeds up the development of environments to train, test, and compare DRL agents in 5g network scenarios is the motivational basis that supports this thesis.

1.2 Objectives

The Thesis is focused on developing a simulated DRL training environment that can simulate 5G networks scenarios. In other words, the goal is to create a tool for simulation environments that can accelerate the experimentation and validation of DRL models and at the same time facilitate the comparison of different algorithms in a common environment simulation platform. The followed approach is based on implementing a solution that integrates simulators used in different 5G network domains, including, Transport, and Software-defined networking (SDN) controller.

Beyond that, the solution needs to have the capacity to provide a standardized interface, allowing access to the executed actions and the state in the environment for communication with the DRL agent.

OpenAIGym will be used to integrate the environment with the DRL agent providing a known interface, that allows the execution of actions and the observation of states in an environment. OpenAIGym is a python toolkit for developing and comparing RL algorithms.

The software to emulate the transport and computing parts of the environment is called Containernet, a fork of Mininet. Containernet provides relevant features, such as using Docker containers as hosts in emulated network topologies. This tool is used by the research community, focusing on experiments in fog computing, cloud computing, multi-access edge computing, and network function computing, which it is one of the most important aspects of 5G networks [7].

Finally, the software used to implement the Software Defined Network controller is Ryu Controller, which is an open-source software-defined networking controller written in python.

After the development of the environment simulation tool, the final step is to benchmark algorithms proposed in the literature for three different scenarios to demonstrate the use of the "AI Gym for Networks" solution.

1.3 Contributions

The contributions achieved by the present thesis are the creation of a tool that facilitates the setup of network environments, decreasing the amount of time the user needs to create a simulation environment, opening space to create different scenarios and facilitating the comparison of different proposals in the same simulation environment.

The different DRL based solutions for three separate problems/scenarios were implemented to demonstrate the simplicity in the development of the different environments using of the developed software.

The first scenario, named "Network Path Selection", is a DRL based routing scenario in which the DRL agent has to decide what is the best path to be installed in the Software Defined Network Controller, taking into consideration the network bandwidth utilization state.

The second scenario, named "Dynamic Network Slicing", is a scenario to train and test the performance of DRL agents for network slice admission control.

The third scenario, named "Dynamic Network Slicing and Path Selection", is a scenario where DRL has the responsibility of considering both slicing admission and choose the best path for each slice. Being a novel contribution of the thesis, extends the model used in the second scenario to jointly consider path selection for a slice in order to optimize both profit and network utilization.

Finally, it is compared the "Dynamic Network Slicing" scenario and the "Dynamic Network Slicing and Path Selection" scenario to conclude what it is the influence of path selection by agent and path selection by a predefined rule in the Dynamic Network Slicing scenario.

1.4 Outline

Besides the introduction, this document is structured into five more chapters.

The second chapter describes the state of the art.

It begins by describing basic concepts related to 5G Networks, SDN, NFV, AI, DL, RL and DRL.

There is also a section on Related Work that provides an overview of research papers that use DRL for network control and management problems with focus on how the environments were simulated.

The third chapter, named "Environment Architecture", describes the software used to develop the proposed solution and what is the architecture behind, in other words, how the different blocks of the "AI Gym for Networks" communicate and works with each other.

Furthermore, the chapter has a sub-section named "Environment Setup" focus on describing from a tutorial perspective, how a user can download and setup his environment to run and test his scenarios.

The fourth, chapter is named "Developed Scenarios" and describes the setup design used to build each of the three example scenarios developed to prove the advantages of the thesis solution.

The fifth chapter, named "Results", is focused on presenting the agents' train for each example, the comparison between the "Dynamic Network Slicing" example and the "Dynamic Network Slicing and Path Selection" example and the conclusions based on the obtained results.

The sixth and final chapter describes the conclusions achieved with the work developed.

2

Fundamentals and Literature Review

2.1 5G Networks

The first discussions about the successor of 4G Long-Term Evolution (LTE) were about analyzing the different options that could take 5G to be a next-generation system far beyond 4G LTE, and then these Technical Reports were turned into Technical Specifications. The areas that were improved by 5G are the following use case scenarios.

Enhanced Mobile Broadband (eMBB) which defines a minimum level of data transfer rate, and delivers increased bandwidth and decreased latency in comparison to 4G, Machine Type Communications (mMTC), that can support extremely high connection density of devices and Ultra-Reliable and Low Latency Communications (URLLC).

One of the most important Technical Specifications is a global view and description of the 5G New Radio (NR), also known as the 5G Radio Access Network (RAN), which can be high-level related to LTE.

The beginning 5G implementation started by re-using the existing LTE radio and core network, these deployments are referred to as 5G Non-Standalone Architecture (NSA). The purpose is to replace 4G networks with 5G components and support many different scenarios. Along 5G NR, 5G networks are also composed of 5G Core Network (CN)s, which is desirable for scenarios such as 5G end-to-end (E2E) networks, bringing new capabilities such as very high reliability and very low latency.

SDN and NFV have a huge importance in terms of the possible 5G applications and flexible network implementations. The goal of implementing 5G technology is to create connections with ultra-low latency, with more connected devices, more speed in transmissions, and network slicing [8] [9].

In addition to improving broadband services, 5G networks also improve Key Performance Indicator (KPI)s, such as user experience data rate, E2E latency, reliability, communications efficiency, availability, and energy consumption. These KPIs have different requirements for different. The solution to support the diverse requirements of 5G use cases it's by using Network Slicing.

To enable the multiplexing of virtualized independent logical networks on the same physical network infrastructure. With this technology, it is possible to have multiple logical networks in parallel on a single physical platform.

With Network Slicing, it's possible to have multiple verticals supported by dedicated logical networks, which are composed of Core Network Function (CNF)s and Radio Network Function (RNF)s, running on top of the physical 5G infrastructures.

Network slicing has many advantages, the logically separated networks can be assigned to different types of service providers, which can administer and manage them. It is also possible to handle the resulting networks in an isolated way, such as isolation of the configurations, isolation of the NFs, isolation of the specifications settings hardware or virtual resources and security isolation [9].

5G networks can be divided into two main networks, the RAN and the CN.

The RAN is related to radio, such as scheduling access concerning aspects, radio-resource handling, transmission protocols, and multi-antenna schemes.

The CN is responsible for E2E connections, authentication, and charging functionalities. The CN should have a service-based architecture with support for slicing, and control-plane/user-plane split.

The next chapters will introduce concepts, such as Network Slicing and the domains that compose a 5G network, i.e., RAN, CN, and the aspects related to the CN, such as Transport, Computing, and Software Defined Network [10].

2.2 Radio Access Network

The RAN is the network that establishes the connection between wireless host and the CN.

With the development of 5G networks, it has been crucial to develop a RAN that can handle a wide range of frequency bands with variable characteristics, such as bandwidths and propagation conditions. It also needs to scale in terms of throughput, the number of devices, and connections [11].

The 5G NR has its radio transmission technology based on LTE concepts, but it optimizes them due to the necessity of delivering better performance and flexible handling.

The implementation of the NR RAN takes into count the existence of the LTE, the first standardized RAN technology that uses Internet Protocol (IP) as the transport protocol, also known as the transition between 3G and 4G. So, the NR has the capacity of connecting to the Evolved Packet Core (EPC), the LTE CN, which can be called a non-standalone operation [10].

The New Generation RAN can operate in Standalone operations as in Non-Standalone operations, which means that it can have 4G and 5G technology working in the same network.

There are different combinations between CNs and RANs, and these combinations, besides having the EPC and 5GCN components, have four distinct elements related to the radio access domain, next-generation NodeB (gNB), which is a 5G BS with a direct interface to the 5G core, Evolved Universal Mobile Telecommunications System (UMTS) Terrestrial Radio Access -NR-gNB (en-gNB), which is a 5G BS with an interface to the 4G core (EPC) via a 4G BS evolved NodeB (eNB) with LTE functionality, eNB which is an LTE BS with an EPC interface and finally next generation eNB (ng-eNB) which is an LTE BS with an interface to 5G CN [8].

The combination of these elements derived to different configuration options, being each of them a viable deployment option of network operators. The RAN options are represented in Figure 2.1.

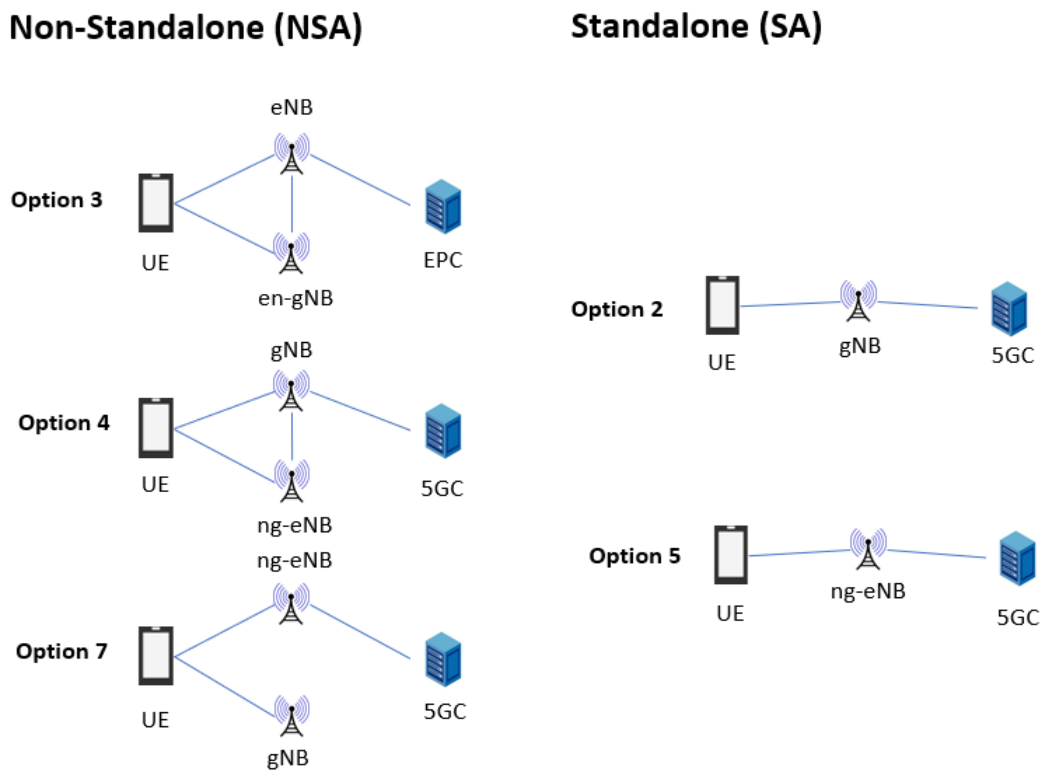


Figure 2.1: RAN Options

In option 2, the gNBs are connected to the 5G CN, using the NR interface, in option 3, the UE is connected to an eNB that works as a Master Node (MN) and to an en-gNB that works as a Secondary Node (SN). The eNB is linked to EPC and en-gNB is linked to eNB and also can be linked to EPC.

In option 4, the UE is linked to a gNB that works as an MN and to an ng-eNB that works as an SN. The gNB is connected to the 5G Core and the ng-eNB is connected to the gNB.

In option 5, the ng-eNBs are connected to the 5GC network, but it also allows the existence of an LTE radio infrastructure if the node eNB gets an upgrade and then it will be able to connect to the 5GC.

Finally, option 7, is when the UE is linked to an ng-eNB that works as an MN and to a gNB that works as an SN. The ng-eNB is linked to the 5GC and the gNB is linked to the ng-eNB.

In summary, we have 5 options, two (option 2 and option 5) are Standalone and the other three are Non-Standalone.

The 5G NR logical node, gNB, brings new concepts to the RAN field. By splitting up the gNB into Central Units (CU)s and Distributed Units (DU)s, it's possible to have important benefits such as flexible hardware implementation allowing coordination performance features, scalable cost-effective solutions, real-time performance optimization, and load management. The gNB Architecture is represented in Figure 2.2.

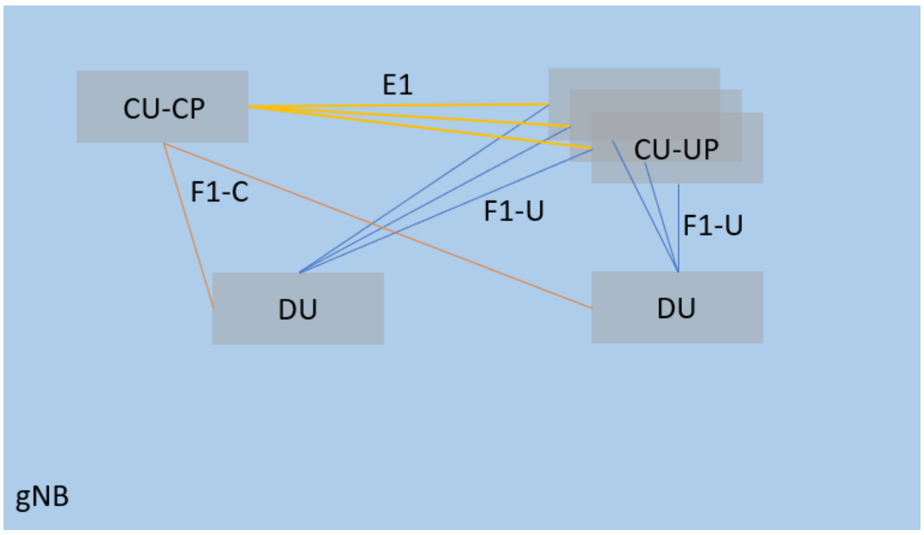


Figure 2.2: gNB Architecture

When implementing the gNB, the important aspect is to know that one gNB-DU only connects to one gNB-CU and can support one or more cells, but one gNB-CU can connect to many gNB-DU. The interface that connects them is called the F1 interface and supports signalling exchange and data transmission. Its functions are separated into F1-Control Plane (C) which handles with Management Functions and F1-User Plane (U), which is responsible for transferring user data and flow control functions [12]. The RAN architecture is represented in Figure 2.3.

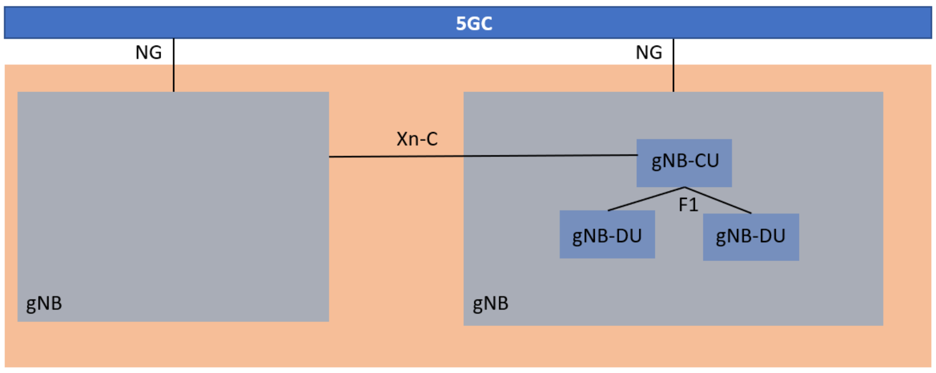


Figure 2.3: RAN Architecture

2.3 5G Core Network

To fulfil the requirements required in 5G network, the CN must follow some design principles that include, network softwarization, cloudification, openness for 3rd-party providers, multi-tenant capability, modularization, and support of a wide range of wireless and wired access network technologies.

It is also important to have a functional design's modularization to provide aspects like flexibility and efficiency on network slicing, to mapping processes between NFs and services, to reuse system functionalities, traffic steering between NFs and to uniform authentication system among others.

Another important feature is the separation between the User Plane (UP) and the Control Plane (CP), which brings advantages, such as independent scalability and evolutionary development.

The 5G CN is split into two main modules, the UP and the CP.

The UP is responsible for the functions of user data transport, routing, forwarding of data packets, traffic control, provision of required QoS, service continuity ensurance, transmission, and recording of billing data.

On other hand the CP is responsible for authentication, authorization, mobility management, roaming, monitoring, and policy definitions.

The functions that are provided by these two main modules are called NFs and they are hosted in a repository and called via Application Programming Interface (API)s.

Some important examples of network functions are the Access and Mobility Management Function (AMF) located in the CP that is responsible for the UE registration, connection, accessibility, mobility management, authentication and authorization and the User Plane Function (UPF) that represents the connection between the RAN and the Data Network (DN) and it is responsible for data handling routing, traffic control, redirection, QoS handling, packet inspection and collection and provision of usage data.

The Policy Control Function (PCF) is responsible for providing network policies related with QoS, traffic forwarding, Access Network priorities and it also make it available to the others NFs, the Unified Data Management (UDM) is responsible to process authentication and identification of data based on user profiles.

The Authentication Server Function (AUSF) is the representation of an authentication server, the Network Slice Selection Function (NSSF) is responsible for associating a network slice with a UE and determines the AMF instance.

And finally, the Session Management Function (SMF) is responsible for establishing, modifying and terminating Protocol Data Unit (PDU) sessions, providing E2E UP connectivity between the UE and a specific DN, and controlling IP address management. The 5G Core Network Functions are represented in Figure 2.4.

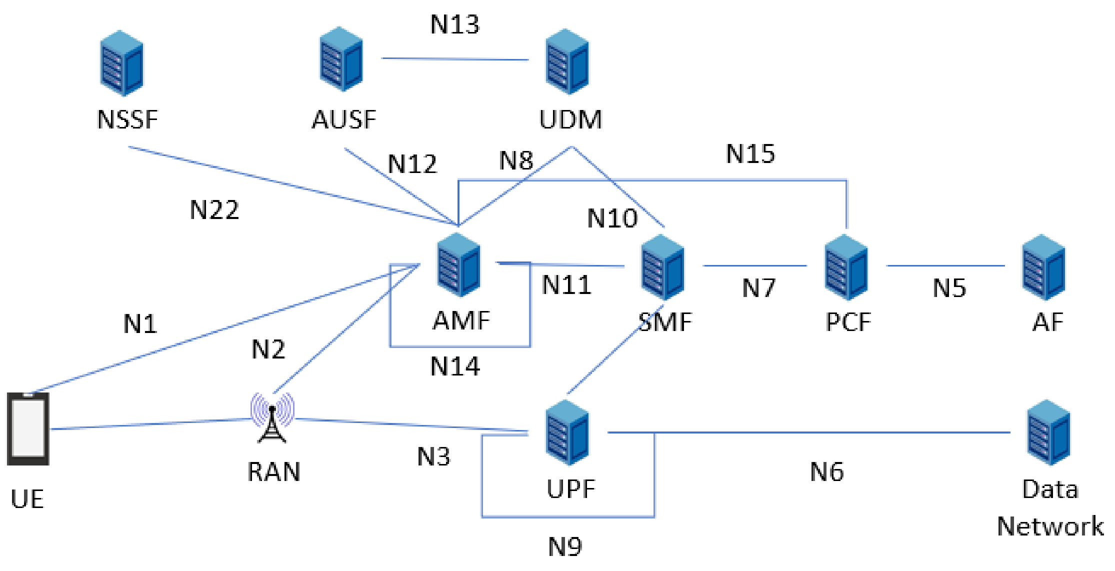


Figure 2.4: 5G Core Network Functions

The architecture responsible for the planning is called Service Based Architecture (SBA) [8]. In SBA, the Network Function Service is a set of NFs that provide the services that are accessed via the Service Based Interface (SBI).

The Services on the CP communicate with each other via Hypertext Transfer Protocol Version 2 (HTTP2) Representational State Transfer Application Programming Interface (RESTfull) APIs. It is possible to create independent , and reusable manageable communications, by defining the NF Services [13].

The hardware used to build the CN consists of SDN switches or high-performance routers with dedicated hardware, the servers for the 5G Core, and the Management and Orchestration (MANO) functions. The Transport Network is based on SDN switches, controlled by SDN controllers, and connects the data centers with Edge Cloud location and Access Networks. The 5G Deployment over an SDN Architecture is represented in Figure 2.5.

To use Virtual Network Functions, a virtualization layer in the hardware is needed to provide virtual computing, storage, network resources, and SDN controllers in form of VMs or Containers [8].

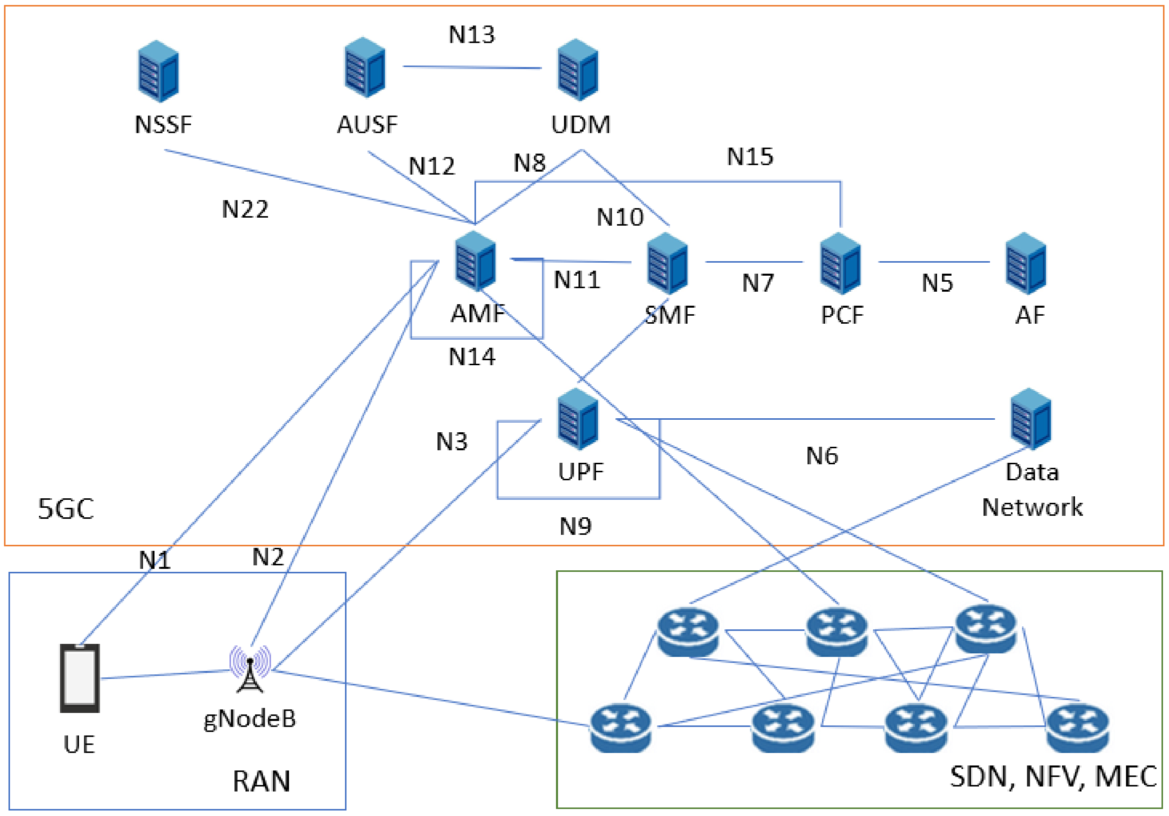


Figure 2.5: 5G Deployment over an SDN architecture

2.4 Network Function Virtualization

NFV is a technology that was created to overcome different problems related to the launching of services inside large and dynamic networks.

NFV aims to overcome problems, such as the difficulty of finding space, power, and people skilled enough to deploy, integrate and operate network services, which need to be installed in specific hardware-based appliances.

Using this kind of hardware that has a short period of life, increases, the costs and the time spent to re-implement the services executed in those hardware-based appliances.

NVFs technology, which is based on standard IT virtualisation technology, brings to the market the possibility of installing network services in a virtualized way, that has no need to depend of the type of hardware that it is being used.

This characteristic comes from installing the Virtual Network Functions inside of VMs or containers that are running in computing devices. In this way it is possible to bring more flexibility and agility to adapt the network to overcome future problems and deliver different types of services and solutions.

Some advantages of using Virtual Network Function (VNF)s are the reduction of equipment costs and energy consumption, the adaptability to the market requirements, enabling a large variety of different ecosystems and the rapid creation or scaling of services.

2.5 Software Defined Network

Software Defined Network allows new approaches in terms of orchestration mechanisms that contribute to the automation of the management and service deployment process [14].

The SDN architecture is split into three main layers, the infrastructure layer is composed of switches, routers, and access points, and it is where the data transportation occurs, the control layer controls the network devices, i.e., which policies are adopted and how the devices should behave, responsible for the network's automation, and management. Finally, the application layer is responsible for getting the network information and for defining the desired requirements and behavior for the network.

The communication between the controller and the network devices is made via Southbound APIs and the communication between the application layer and the controller is made via Northbound APIs.

The network performance is directly affected by the location of the SDN controller, so it is important to have an idea about some SDN control plane architectures.

The centralized control plane architecture has only one centralized controller connected to the infrastructure layer. The advantage of this approach is the overall network overview that it provides leading to better decision-making and knowledge about what is happening in the network. The problems with this type of architecture are the lack of redundancy, not preventing failures, and the bad performance when the network's size increases, so it is a better architecture for small-scale networks. The Centralized Control Plane Architecture is represented in Figure 2.6.

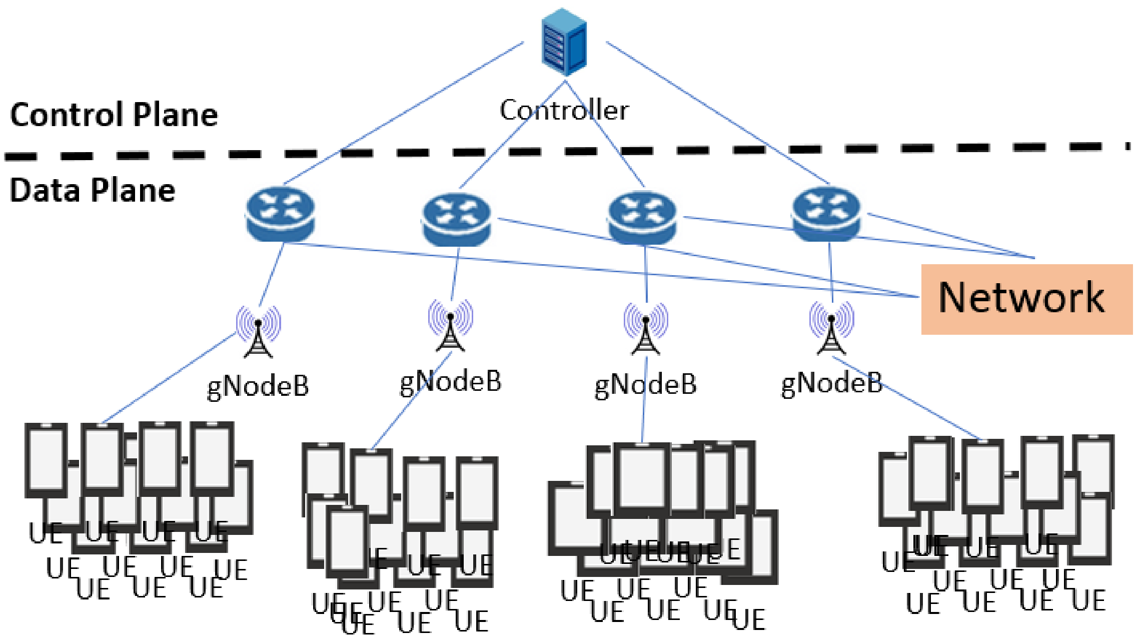


Figure 2.6: Centralized Control Plane Architecture

The distributed control plane architecture tries to overcome the centralized control plane architecture problems by having multiple distributed controllers, creating sets of clusters structured each with its own responsible controller. Having many SDN controllers increases the response time compared to the previous architecture and the cost to maintain the architecture is also higher. The Distributed Control Plane Architecture is represented in Figure 2.7.

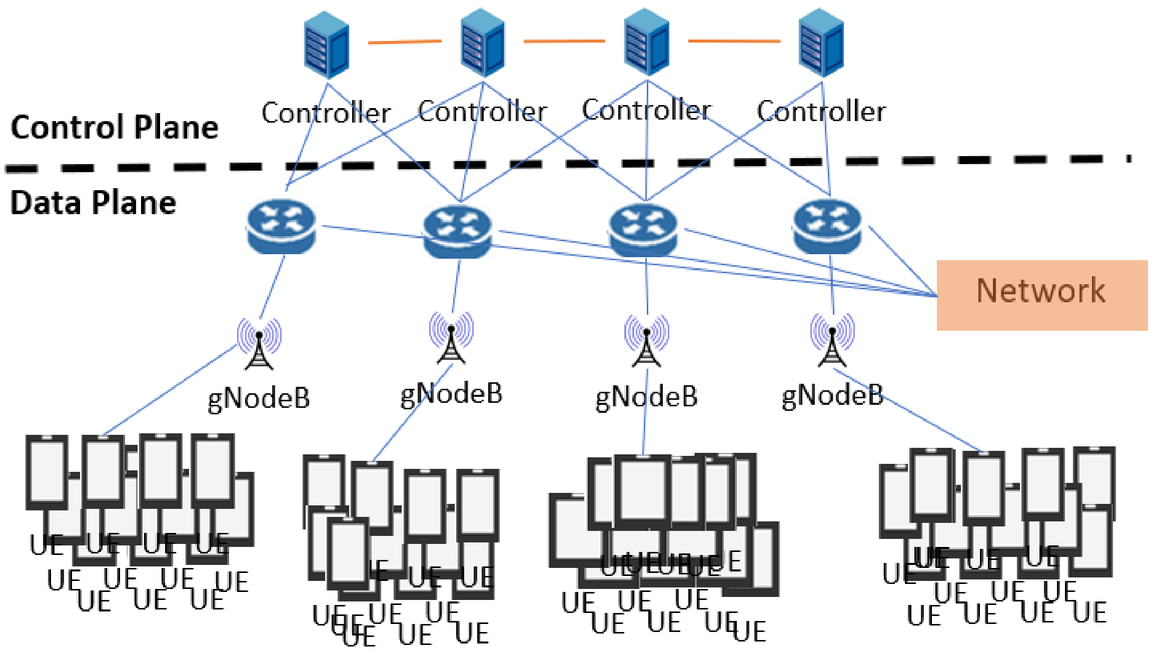


Figure 2.7: Distributed Control Plane Architecture

The Logically Centralized-Physically Distributed (LC-PD) CP architecture combines the two previous approaches, by having distributed controllers spread across the network. The idea is to have controllers distributed physically all over the network, but they act as a centralized one. The LC-PD Control Plane Architecture is represented in Figure 2.8. The distributed controllers are connected and aware of any network change, and all of them share the same information [15].

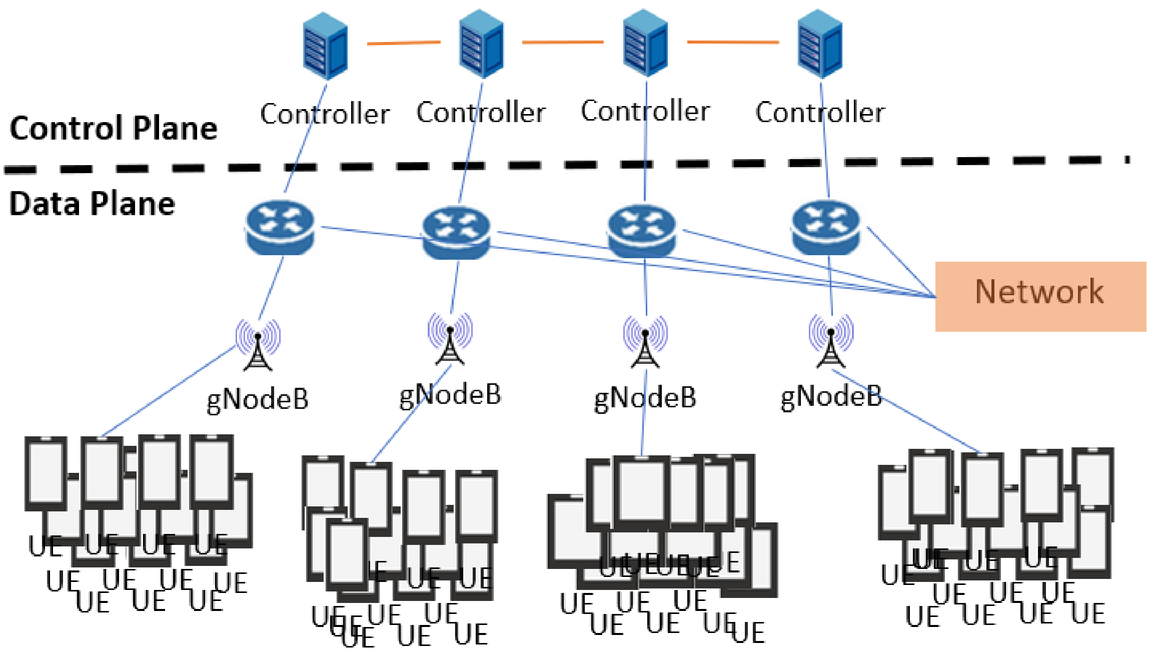


Figure 2.8: Logically Centralized-Physically Distributed Control Plane Architecture

The communication between the Controller and the network switches can be accomplish by using a communication protocol named OpenFlow.

2.5.1 OpenFlow

The OpenFlow protocol is one of the possible protocols for the SouthBound API and is supported by most of the existing virtual switching software.

To use the OpenFlow protocol a secure channel communication with the SDN controller must be established. The switches forwarding behaviour is then determined by a Flow table were forwarding rules are installed that determine how to handle specific packets.

The forwarding rules are defined using flow entries that are composed by three main fields: Match, instructions and priority. Flow entries statistics are provided by counters and each flow entry has a priority value.

The Match field is used to define to which packets the rule is applied.

Each match field has a corresponding action/instruction, which defines how the switch has to handle a specific packet that is sent to the switch. The Counters contain flow statistics and the Priority defines the organization of flow entries inside the flow table. Finally there are Timeouts to control the lifetime of a flow entry before expiring.

2.6 Artifial Intelligence

AI consists of many branches that use different techniques to solve many problems. Some of these branches are Natural Language Processing, Expert Systems, Fuzzy Systems, and ML.

Natural Language Processing has the task of linking the human language and the computer understanding, i.e., the study of how a computer can process and analyze text and documents written in human language.

Expert Systems are computer systems that can solve complex problems and give advice at a level comparable to an expert in the problem's field.

Fuzzy Systems aims to solve problems with a certain level of imprecision and uncertainty. ML is the science of giving a machine the capacity to learn and act based on the data that we offer [16] [17].

Acquiring knowledge has some aspects in consideration. First, we need to understand the processed information and then make decisions. ML can be described as a system that can learn from its experience, respecting a set of tasks, and with that experience obtain knowledge to improve its performance.

ML solves many different tasks. We can use it for classification to find the preset category belonging to the respective input. Regression to predict numeric values depending on the given input, Transcription to observe unstructured data and transform it into a textual form, Machine Translation to translate information, Anomaly Detection to detect unusual behaviors when analyzing events, etc.

The algorithm performance is specific to the task attributed to the system. We can use performance measurements such as accuracy for classification tasks or error rate to measure if the model creates an incorrect output. When we have known data, we usually split the data set into training sets and test set to evaluate the accuracy. Relatively to the kind of experience that a ML algorithm can have, it is possible to distinguish three ML categories.

In Supervised Learning, the system has access to a data set that is known previously, and it's labeled. This solution is useful for classification, regression tasks, and calculating outcomes.

In Unsupervised Learning, the system must learn from unknown data, observe many examples of random data, and try to find properties of that data without any guidance. It's useful to deal with clustering and consists in slipping the data-set into clusters of similar examples, mining associative rules, and discovering patterns in new data sets [18] [19].

2.7 Reinforcement Learning

RL is a ML area concerned with the methods and algorithms used to make an agent learn how to act when facing a specific situations in the environment around him. This agent does not know which actions to take, so it will have a try and error behaviour, until finding out what action generates the best reward.

Distinct from the ML branches described before, RL aims to maximize rewards, improving the agent's actions instead of classifying or clustering data sets.

The agent needs to have the ability to evaluate the state of its environment, take actions that can change the state, and have goals depending on the environment's state.

The policy defines the behaviour of an agent depending on the state, in other words, the action that should be executed in a specific situation.

The reward signal defines what is good and bad for the agent, in other words, this element defines if a reward sent by the environment indicates that the agent took a good action or not.

The value function is present in each state and indicates to the agent if the choice of that specific state will have more rewards in a long term, even if the specific state doesn't have a good reward the value function indicates if the next states, the future ones, will give good rewards to the agent.

And finally, the environment generates the result and reward of the action depending on the agent state [20]. The Reinforcement Learning Overview is represented in Figure 2.9.

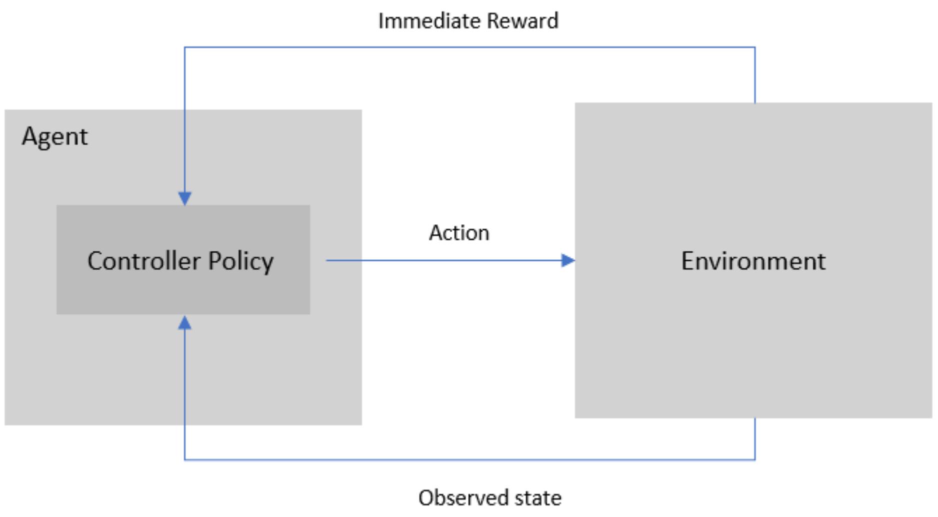


Figure 2.9: Reinforcement Learning Overview

2.7.1 Markov Decision Process

Modeling a control task as an Markov Decision Process (MDP) is fundamental in Reinforcement Learning. The current state of the system is used to choose the optimal actions that maximize the rewards, there is no memory requirement.

The MDP is associated to evaluative feedback and it is structured by sequential decision making, actions influence, immediate rewards, subsequent events, and future rewards.

In MDPs, we have an agent who learns with the experience acquired from acting in the environment. The goal is to define an optimal policy behaviour by giving a set of states, a set of actions, a probability for an agent to jump from one state to another, and an immediate reward by some action [21].

The outcomes of MDPs are partially random and controlled by an agent or a decision-maker and used in the studying of optimization problems.

Normally, MPDs are tuples with a finite set of actions, a finite set of states, a transition probability from one state to another after a specific action, and the instant reward obtained after the action is made. The MDPs' goal focused on discovering the best policy that can maximize the reward function [6].

2.8 Deep Learning

DL algorithms are built to extract knowledge from complex data representations using a hierarchical learning process in Artificial Neural Network (ANN) with multiple layers. Each of the middle layers depends on the previous one, which means that the input of each layer will be calculated having in attention the weights of the links that are connecting two layers and a nonlinear function that can be chosen according to the problem.

After the computation, the error value between the output and the wanted result is used in a backpropagation algoritm to update and adjust the weights.

This approach can transform a complex dataset into a combination of more elementary components. The Deep Neural Network Overview is represented in Figure 2.10.

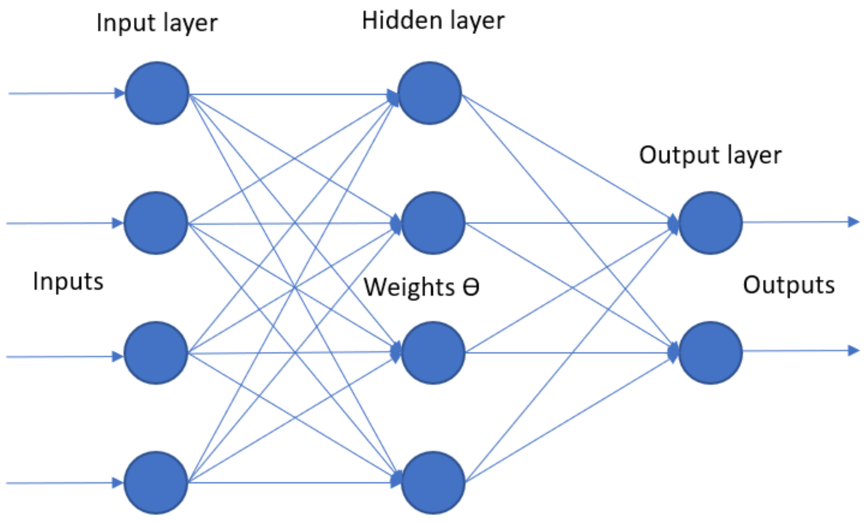


Figure 2.10: Deep Neural Network Overview

There are many different models used in DL, the most known are the Convolutional Neural Network (CNN) and the Recurrent Neural Network (RNN).

The purpose of CNNs is to process data that come in form of multiple arrays and are structured in a series of layers. Convolutional layers extract features from the data, pooling layers mapping the previous features, reducing the data dimension, and fully connected layers predict classes for the input data. The problem with this algorithm is failing in the interpretation of the temporal and sequential information. But for this type of problem, we can use Recurrent Neural Networks.

RNNs process input at a time, but they have a vector in their hidden units that has the history of all past elements of the sequence. This is due to the existence of direct cycles in these networks that allow the information to circulate in the network, making the output related not just with the present input but also related to the previous steps.

DL it's the best way to solve classification problems but it's limited, we can't use it in a task where we need to learn how to act in the environment and not just to classify or make a prediction [22] [23].

2.9 Deep Reinforcement Learning

DRL tries to combine the benefits of DL and RL to build AI Systems. We use deep neural networks to improve RL elements, such as the action value or the policy. The Deep Reinforcement Learning Overview is represented in Figure 2.11.

DRL is an alternative to traditional methods that are limited and dependent on models, network traffic and others data sets [6].

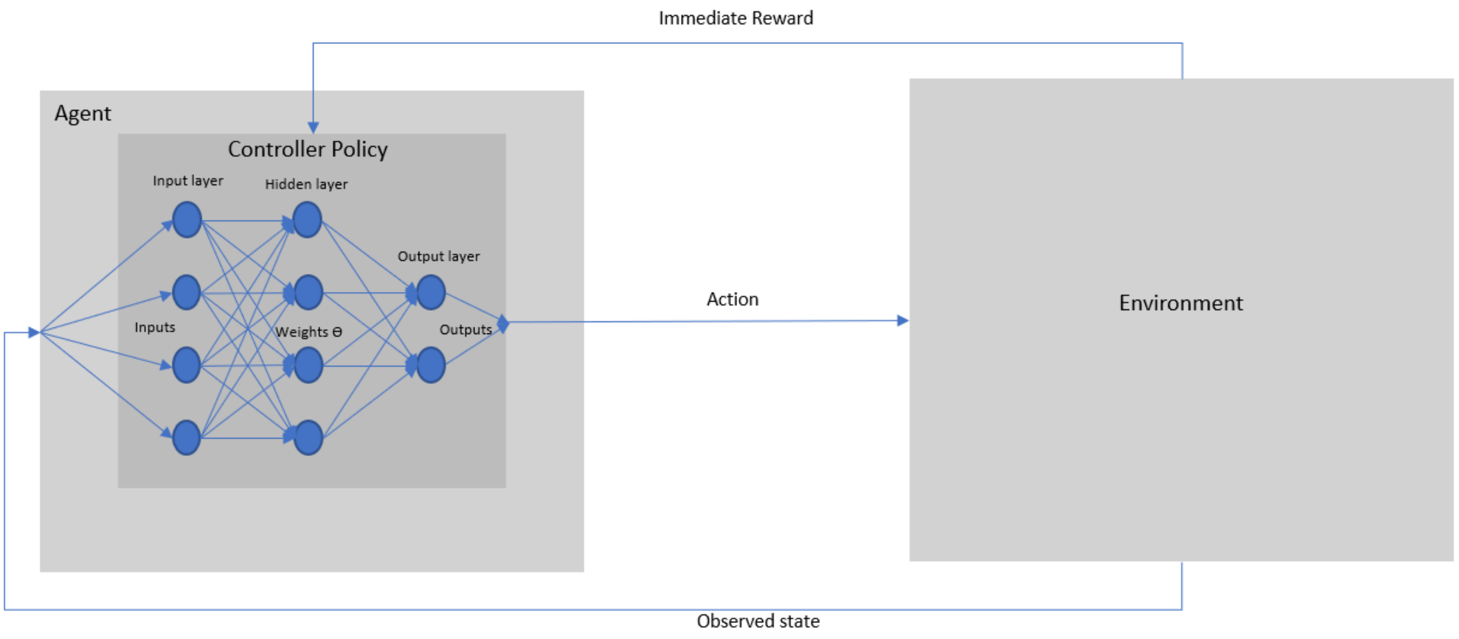


Figure 2.11: Deep Reinforcement Learning Overview

2.9.1 Deep Q-Network

The algorithm Q-Learning, has been proven to converge to an optimal solution when using a tabular case or a linear function approximation, but when using a non-linear function becomes unstable. Q-Learning algorithms, also suffers when applied to complicated system models with large spaces [6].

With the evolution and progress of deep neural networks, it has become possible to overcome this issue, igniting the research of DRL [24].

The Deep Q-Network (DQN) algorithm can be explained as a combination of a Q-learning algorithm with Deep Neural Network (DNN).

The algorithm uses a function called Q-function to return the Q-value considering the received input state-action pair. The Q-values are stored in the Q-Table, which is represented by a DNN.

To optimize the function, the value is calculated by multiplying each reward by a reward coefficient that measures the reward importance and compares the result to the observed accumulated rewards. The process is described by the equation 3.1.

$$Q\left(S_t, A_t\right) = Q\left(S_t, A_t\right) + \alpha\left[R\left(S_t, A_t\right) + \gamma max Q\left(S_{t+1}, a'\right) - Q\left(S_t, A_t\right)\right] \tag{2.1}$$

The updated Q-value is the current Q-value (Q(S_t, A_t)) plus the Temporal Difference, which is the amount of value expected in the future. In this equation, α is the learning rate, γ is the discount factor.

The Q-value dependency on the observed reward, make it unstable, and because of that, the same state-action pair input can have different outcomes which lead to constant function and policy updates. There are two important mechanisms to overcome the instability problem.

The first mechanism is known as target network or fixed target Q-network and aims to increase the learning stability. Instead of using the Q-network, a target network is used to slowly update the primary Q-networks' values, and with that the dependence between the target and estimated Q-values are significantly reduced, stabilizing the algorithm. The usage of a Target Network, which is a copy of the main DQN Q, will help stabilize the back-propagation training process because it will hold off on updating its parameters. By using the Q-values of the target network to train the main one, the previously mentioned updates will decrease [6].

Another important technique is called Experience Replay mechanism or Replay Buffer which consists in saving at each time step, the agent's experience into a replay buffer. A set of samples from this buffer is then uniformly sampled to apply the Q-learning update.

The advantage of using a replay buffer is that the experience in each step can be reused to learn the Q-function, smoothing out learning and reducing oscillations and divergence in the parameters [24]. Figure 2.12 presents an overview of Deep Q-Network algorithm.

The Experience Replay method has a problem related to the extraction of samples from the data structured randomly. So a method called Prioritized Experience Replay was implemented, which samples data by priority criteria, characterized by their recent addition to the replay buffer or low error verified in past transitions.

2.9.2 Dueling Deep Q-Network

In Q-learning Algorithms, the Q-value has the function to express the evaluation of taking a specific action at a provided state, which can be split into two principal values.

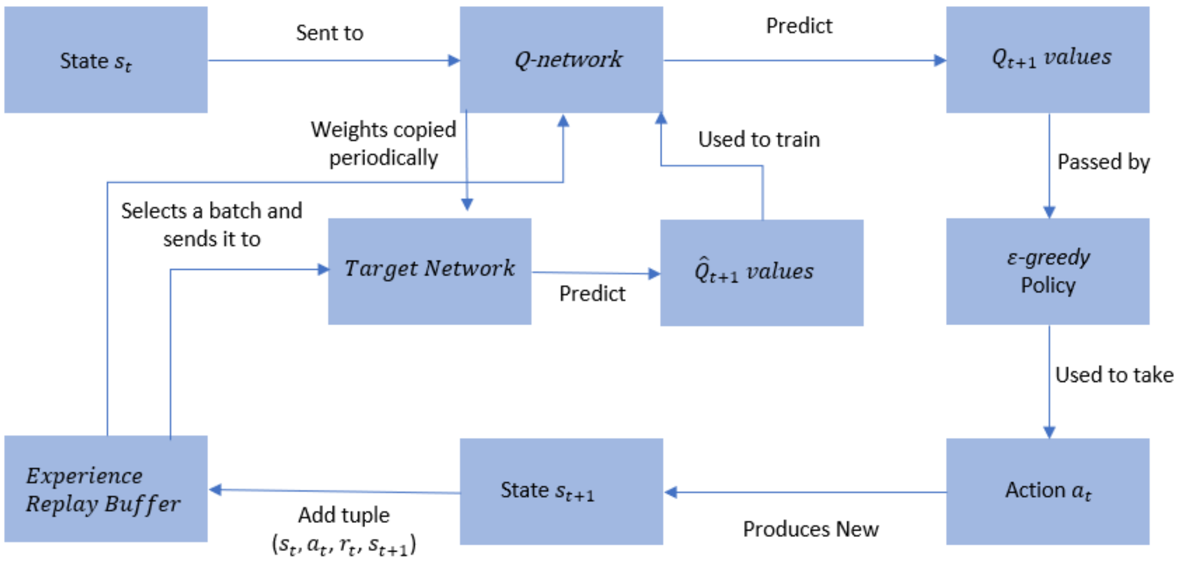


Figure 2.12: Deep Q-Network Overview

One of the values is known by state-value function, V(s), and it's used to calculate the relevance of being in a specific state, the second value is called action-value function, A(a), and it is used to calculate which is the best action to choose.

These two functions represent different layers in the DQN, and by summing them, we will get the Q-Value. The process is described by the equation 3.3.

$$Q(s, a, \alpha, \beta) = V(s; \beta) + \left(A(s, a; \alpha) - \frac{\sum_{a'} A(s, a'; \alpha)}{|A|} \right) \tag{2.2}$$

Where, α and β are function parameters, and |A| represents the total number of actions. This method makes it easier for DQN to learn what action valuers will get a better result. In Figure 2.13 shows the Dueling DQN agent architecture.

In some MDPs, there is no need to calculate the two values that composed the Q-value function at the same time, so it was developed an idea of using two sequences of fully connected layers. These two sequences are constructed to be able to give independent estimations on the state value function and action, and finally can be combined to generate the Q-value output. This approach has been proved that can outperform DQNs, but only have clear benefits for MDPs when there is large action space.

2.9.3 Double Deep Q-Learning

Q-learning uses the maximum action value as an approximate value for the maximum expected action value, influenced by the samples used to choose which action is the best and to calculate the action-value, which are the same. This procedure creates an overestimation problem, and the Double Deep Q-Learning (DDQL) model tries to solve it.

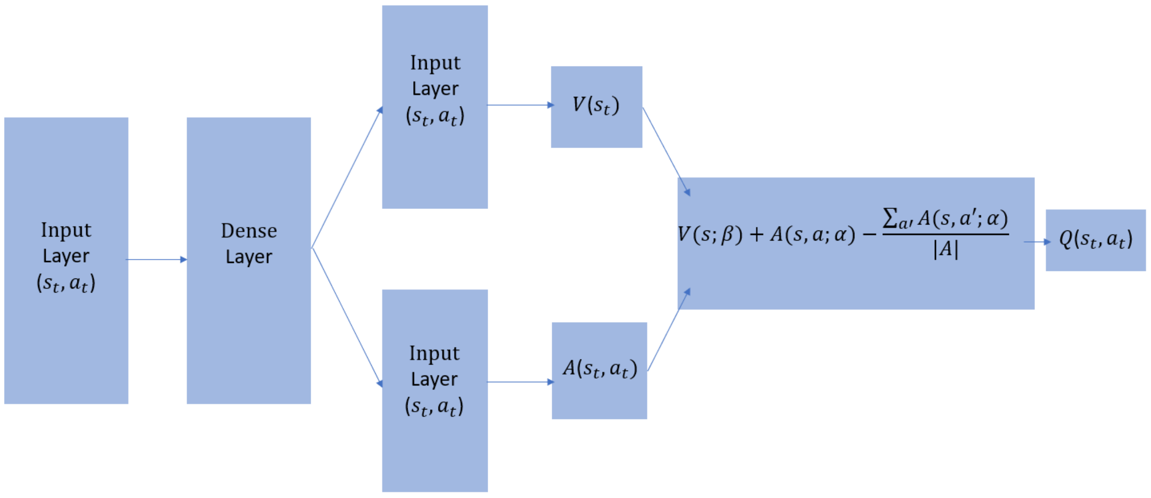


Figure 2.13: Dueling Deep Q-Network Architecture Overview

The DDQL algorithm is based on a solution that uses two Q-functions to simultaneously select and evaluate action values. This means that the selection of action is still due to the online weights, but the second set of weights is used to evaluate fairly the value of the policy.

In DDQL, the weights of the second network are replaced with the weights of the target networks for the evaluation of the current greedy policy, The update of target networks stays unchanged from DQN, and remains a periodic copy of the online network [6].

2.9.4 Deep Q-Networks with Actor-Critic

As seen before, DQN algorithm is an off-policy method, which means that it is independent of the agent's action, figuring out the optimal policy despite the agent's motivation, and integrating the Q-learning algorithm with a deep neural network to get knowledge from visual inputs and response with decision outputs.

The problem is that, while the inputs can be raw and observed in a high-dimensional space, DQNs just handle discrete and low-dimension action spaces. The actor-critic is an extension of the Reinforce algorithm and converts the policy gradient update of Monte Carlo into a Temporal-Difference (TD) update, by incorporating a critic.

The multi-step method, make it possible to have a degree of bootstrapping flexibly selected, and with that, the updated policy doesn't have the necessity of waiting until the end of the game. The problem with this method is being an on-policy algorithm, making it have a lower sample efficiency.

With the combination of these two methods, it is possible to take advantage of both, because with DQNs, the actor-critic methods are turned into off-policy methods. So, now using this combination, networks can be trained using samples from a replay buffer (seen in previous chapters) that improves sample efficiency, and can also minimize correlations between samples, making it possible to learn value function in a stable and consistent way, without losing the advantage obtained by the actor-critic method of easily solving problems by using continuous action spaces to learn the policy [24].

2.10 NS3

All of the tools described in the present section, were studied and tested to be used for simulating a possible RAN. The problem with these tools were the difficulty to implement due to the fact of being deprecated in some functions that were needed to build the wanted solution, "AI Gym for Networks".

Even though they were not used, it is important for the reader to know there were studies and tests focus on creating a RAN simulator, which couldn't be executed properly.

NS3 is an open-source project that aims to deliver a network simulation platform that can assist the community in networking research and education. Some of the advantages of using ns-3 are its capacity to develop studies that are more challenging or not possible to perform in real systems, to analyse systems behaviour in an extremely controlled, reproducible environment, and to understand how networks work.

There are many simulation tools, but ns-3 has some distinguishing features, which is the advantage of being a modular simulator regarding the possibility for the user to use external tools with ns-3 to complement the study/research, such as data analysis and visualization tools and external animators.

Along with ns-3, it's important to introduce two ns-3 modules (NS3-gym and 5g-Lena) that were studied and tested for simulate the wanted environment [25].

2.10.1 NS-Gym

Ns3-gym is an ns-3 module to connect the ns-3 network simulator and OpenAI Gym framework. This middleware is responsible for transferring state and control between the Gym agent and the simulated environment. It is structured by two modules, the Environment Gateway written in C++, and the Environment Proxy written in Python.

The Environment Gateway is located inside the simulator and has the responsibility for converting the environment state into structured numerical data and translating the received actions from OpenAI Gym into function calls with appropriated arguments to the environment.

The Environment Proxy receives the environment state and sends it to the agent via the Gym API. The state and actions are transferred as numerical values and the user is who must define their semantics.

The ns3-gym toolkit is responsible for simplifying the development of networking environments and training RL-based agents, it also enables the collection and exchange of information between frameworks, handles the managing of the simulation process life cycle, and freezing the simulation's execution of the agent's interaction [26].

2.10.2 5G-Lena

5G-LENA is an ns-3 module to simulate 3rd Generation Partnership Project (3GPP) 5G networks. Some of the features that this module supports are NSA, architecture: 5G RAN and 4G EPC, flexible and automatic configuration of the NR frame structure through multiple numerologies, realistic beamforming based on SRS-based channel estimates, Time-Division Multiple Access (TDMA) and Orthogonal Frequency-Division Multiple Access (OFDMA)-based access with variable transmission time intervals and single beam capability, etc.

The NR module is based on the ns-3 mmWave simulation tool, and it is a pluggable module for ns-3 that can be used to simulate 5G NR networks. The features that were added and modified are the flexibility and automatic configuration of the NR frame structure through multiple numerologies, the OFDMA based access with variable TTIs, the restructuration and redesign of the Media Access Control (MAC) layer, the UpLink (UL) grant-based access scheme, the NR-compliant processing timings, the new Bandwidth Part (BWP) managers, etc. The NR module was developed to perform E2E simulations of 3GPP- oriented cellular networks.

2.11 Related Work

The role of DRL in trying to solve network problems is a common solution nowadays.

The paper [27], focuses on solving a problem that decreases the QoS requirements and deny reliability criterion in latency-critical applications.

This problem has its roots related to the difficult of having good dynamic resource management in virtualized environments and at the same time finding a balance between efficiency and reliability parameters.

And for this problem, the paper proposes a model for an Asynchronous DRL enhanced Graph Neural Network for topology-aware VNF resource prediction in dynamic NFV environments, which solves the need by predicting scaling decisions to balance the provisioning time sink.

The software used to simulate the DRL agent environment was the VIM-Emulator, which is a platform that was created to emulate realistic E2E multi-PoP scenarios and is based on Containernet.

The paper [28] starts to present a problem related to the need of having faster and more robust network services that lead to the use of technologies, such as NFV and SDN.

These technologies create the necessity of having solutions based on self-adaption and automation, capable of managing resource allocation. So the researchers present a DRL algorithm, more specifically, a Deep Deterministic Policy Gradient RL algorithm, to automate the Virtual Network Functions deployment process between edge and cloud network nodes.

It was used Containernet and a Network Level POX controller, to create the environment for their research.

The paper [29], starts to present the necessity of having ways and techniques to coordinate detection and mitigation, considering that are always anomalies that need to be found and solved in computer networks.

The solution found by the researchers was to collect network metrics, group them into profiles, and use RL algorithms to detect and handle anomalies.

To create the environment for their research, they choose to work with Containernet and POX to develop the SDN Controller.

The paper [30] identifies that the Distributed Denial-of-Services(Distributed Denial-of-Services (DDoS)) flooding attack is a threat that exists for more than two decades and it is always difficult to protect from it, because of the fact that malicious traffic can be mixed with benign traffic.

In another hand, using the technology SDN which is a centralized controller that has an overview of the network, brings the necessity of having better defenses against network attacks.

As a solution, the researchers proposed a DRL algorithm that reduces DDoS flooding attacks by learning the optimal mitigation policies under different attacks.

To train and test their solution, the researchers used Mininet as a network emulator and Ryu Controller as SDN Controller to set up the environment.

The papers described previously have three things in common. All of them try to solve a network problem, all of them present solutions based on DRL and all of them have to build a simulator ad-hoc using different simulators.

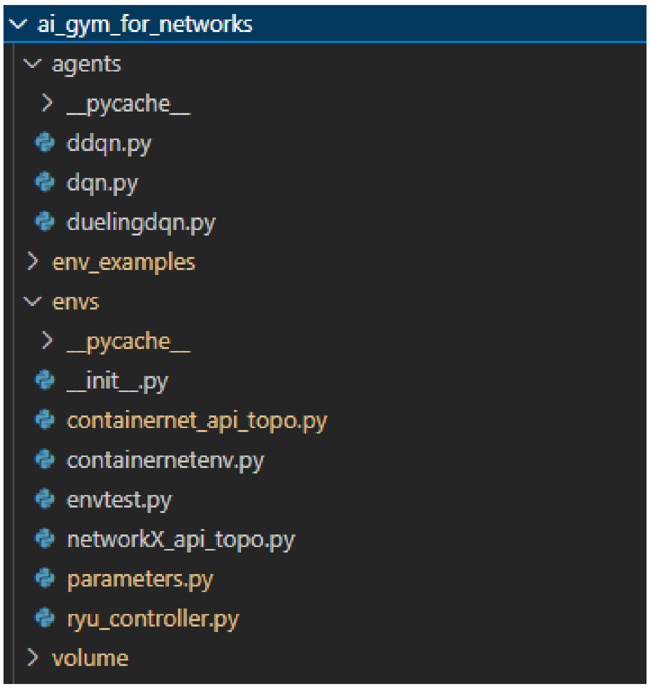
Because of these and other examples, the proposed thesis theme presents a tool in which there is no need to create an ad-hoc environment. The user just has to understand the architecture and follow a tutorial to set up his own environment.

3 Environment Architecture

3.1 Architecture Overview

The present section focus on describing the developed project architecture and what is the interaction between the different modules.

The complete solution developed during the thesis can be found on `https://github.com/pedrocapelo10/ai_gym_for_networks` and has the name "AI Gym for Networks". The Project folder structure is represented in Figure 3.1.



```
ai_gym_for_networks
  agents
    __pycache__
    ddqn.py
    dqn.py
    duelingdqn.py
  env_examples
  envs
    __pycache__
    __init__.py
    containernet_api_topo.py
    containernetenv.py
    envtest.py
    networkX_api_topo.py
    parameters.py
    ryu_controller.py
  volume
```

Figure 3.1: "AI Gym for Networks" Folder

The project architecture is built by using four important modules. The SDN Controller built using Ryu Controller, the network topology, composed of containers and Open VSwitches, and emulated by Containernet, the framework that connects the DRL agent to the environment, that is built using OpenAI gym and "ContainernetAPI", which is a class that is defined in the "containernet_api_topo.py" file, enables the communication between the Ryu Controller, Containernet and the OpenAI Gym framework. This class gives access to Containernet functions, enables the creation and use of Containernet components, and offers auxiliary functions to extract information from Containernet and to create the files that will be described below. Figure 3.2 illustrates the interaction between the Ryu Controller, Containernet and OpenAI Gym via the "ContainernetAPI" class.

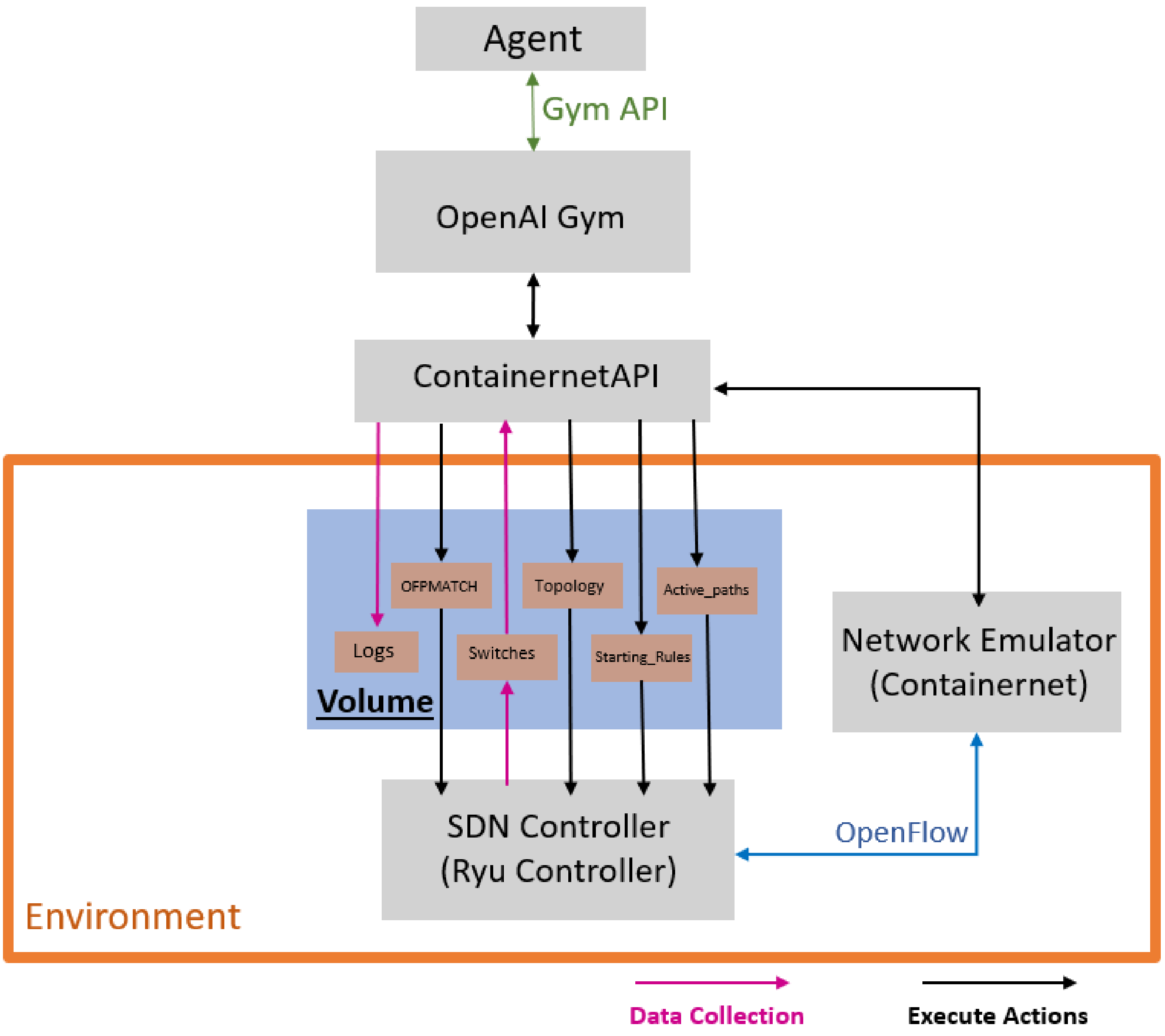
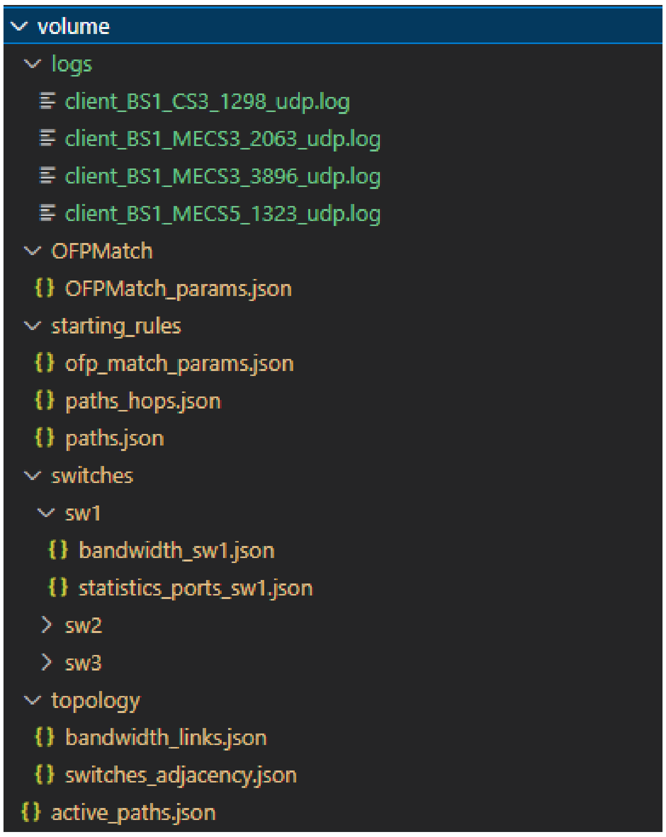


Figure 3.2: "AI Gym for Networks" Communication Architecture Overview

Information is shared between the "ContainernetAPI" class and the Ryu Controller by using a folder named "volume" where the shared data shared is stored. The "volume" folder was created because the Ryu Controller API was not able to get all the information related to the topology created in Containernet. The content of the "volume" folder is represented in Figure 3.3



```
volume
  logs
    client_BS1_CS3_1298_udp.log
    client_BS1_MECS3_2063_udp.log
    client_BS1_MECS3_3896_udp.log
    client_BS1_MECS5_1323_udp.log
  OFPMatch
    OFPMatch_params.json
  starting_rules
    ofp_match_params.json
    paths_hops.json
    paths.json
  switches
    sw1
      bandwidth_sw1.json
      statistics_ports_sw1.json
    sw2
    sw3
  topology
    bandwidth_links.json
    switches_adjacency.json
  active_paths.json
```

Figure 3.3: Volume Folder

In all the shared files, containers are identified by using their MAC address while switches are identified by the label ""S"+switch identifier".

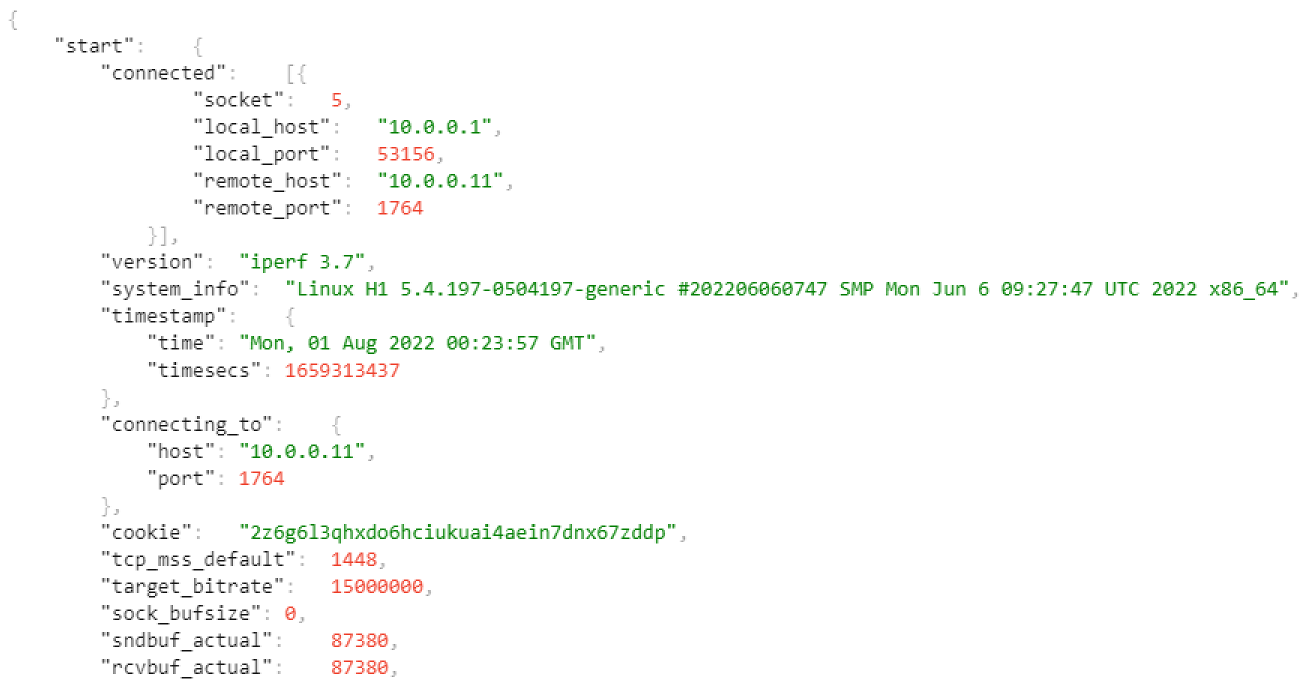
The shared information between the "ContainernetAPI" class and the Ryu Controller is related with the Open VSwitch rules. "ContainernetAPI" class creates four folders: "logs", "starting rules","topology and "OFPMatch".

The "logs" folder is used to save statistics about the generated traffic. Figure 3.4 shows an example of a log file.

The "starting_rules" folder contains the information about the initial rules that need to be installed in the Open VSwitches by the Ryu Controller.

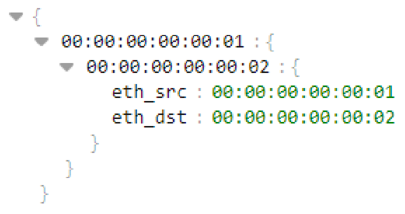
This folder contains three different types of JavaScript Object Notation (JSON) files. The "ofp_match_params.json" file stores information about the OFPMatches (Match part of the Rules) that need to be used when installing the rules, an example is illustrates in Figure 3.5.

The "paths.json" file, illustrated in Figure 3.6, stores information about the input port and output port that belong to a given path in each switch. Finally, the "paths_hops.json" file stores information about the sequence of nodes in a path and is illustrated in Figure 3.7.



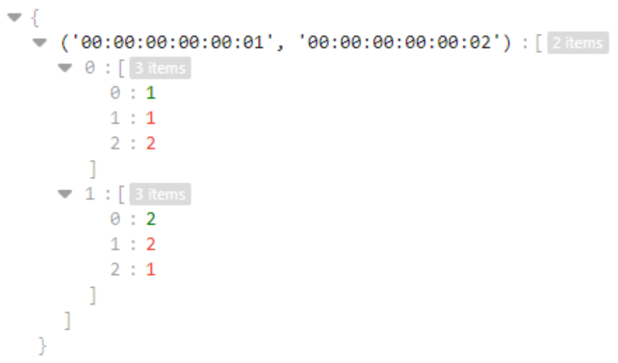
```
{
    "start":    {
        "connected":    [{
                "socket":   5,
                "local_host":   "10.0.0.1",
                "local_port":   53156,
                "remote_host":  "10.0.0.11",
                "remote_port":  1764
            }],
        "version":  "iperf 3.7",
        "system_info":  "Linux H1 5.4.197-0504197-generic #202206060747 SMP Mon Jun 6 09:27:47 UTC 2022 x86_64",
        "timestamp":    {
            "time": "Mon, 01 Aug 2022 00:23:57 GMT",
            "timesecs": 1659313437
        },
        "connecting_to":    {
            "host": "10.0.0.11",
            "port": 1764
        },
        "cookie":   "2z6g6l3qhxdo6hciukuai4aein7dnx67zddp",
        "tcp_mss_default":  1448,
        "target_bitrate":   15000000,
        "sock_bufsize": 0,
        "sndbuf_actual":    87380,
        "rcvbuf_actual":    87380,
```

Figure 3.4: Log File Example



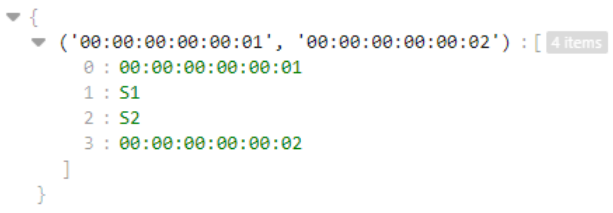
```
{
  00:00:00:00:00:01 : {
    00:00:00:00:00:02 : {
      eth_src : 00:00:00:00:00:01
      eth_dst : 00:00:00:00:00:02
    }
  }
}
```

Figure 3.5: OFPMatch File for Starting Rules Example



```
{
  ('00:00:00:00:00:01', '00:00:00:00:00:02') : [ 2 items
    0 : [ 3 items
      0 : 1
      1 : 1
      2 : 2
    ]
    1 : [ 3 items
      0 : 2
      1 : 2
      2 : 1
    ]
  ]
}
```

Figure 3.6: Paths File for Starting Rules Example

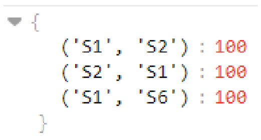


```
{
  ('00:00:00:00:00:01', '00:00:00:00:00:02') : [ 4 items
    0 : 00:00:00:00:00:01
    1 : S1
    2 : S2
    3 : 00:00:00:00:00:02
  ]
}
```

Figure 3.7: Paths and Hops File for Starting Rules Example

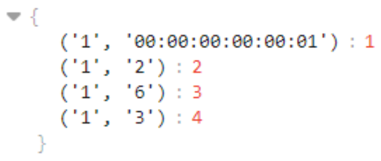
The "topology" contains of two JSON files. The "bandwidth_links.json" file depicted in Figure 3.8 stores information about the bandwith of the topology links that connect the paths between containers.

The second file, named "switches_adjacency.json", is used to store information about the output port that connects to a specific neighbor. Figure 3.9 shows an example.



```
{
    ('S1', 'S2') : 100
    ('S2', 'S1') : 100
    ('S1', 'S6') : 100
}
```

Figure 3.8: Bandwidth Links File Example

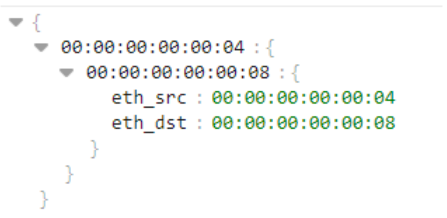


```
{
    ('1', '00:00:00:00:00:01') : 1
    ('1', '2') : 2
    ('1', '6') : 3
    ('1', '3') : 4
}
```

Figure 3.9: Switches Adjacency File Example

The folder "OFPMatch" has a file named "OFPMatch_params.json" that stores the OPFMatch information that needs to be installed between a pair of containers, during the agents training. Figure 3.10

The "ContainernetAPI" class also creates a file named "active_paths.json" that stores the information about the path and the hops that need to be installed to connect a pair of containers, which has the structure depicted in Figure 3.7



```
{
  00:00:00:00:00:04 : {
    00:00:00:00:00:08 : {
        eth_src : 00:00:00:00:00:04
        eth_dst : 00:00:00:00:00:08
    }
  }
}
```

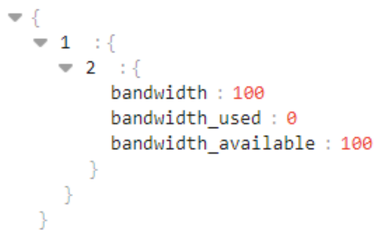
Figure 3.10: OFPMatch File Example

The shared information that the Ryu Controller sends to the "ContainernetAPI" class is stored inside a folder called "switches". This folder has a sub-folder with the identifier of each switch named "s+identifier" and each of these sub-folders contain two JSON files.

One of the files is named "bandwidth_sw+identifier.json" and contains information about the initial bandwidth, define when creating the links between the switchers, the used bandwidth, and the available bandwidth between the switch and its neighbors. Figure 3.11 shown an example of the file structure.

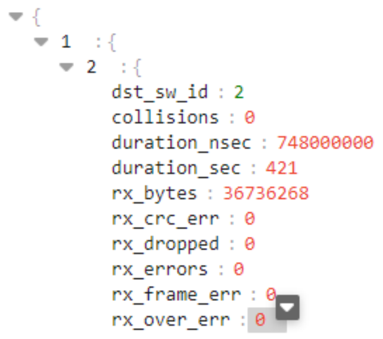
The second file, named "statistics_ports_sw+identifier.json" illustrated in Figure 3.12, stores switch port statistics obtained by the Ryu Controller.

The next section will describe all the functions used by the "ContainernetAPI" class.



```
{
  1  :{
    2  :{
      bandwidth : 100
      bandwidth_used : 0
      bandwidth_available : 100
    }
  }
}
```

Figure 3.11: Switch Bandwidth File Example



```
{
  1  :{
    2  :{
      dst_sw_id : 2
      collisions : 0
      duration_nsec : 748000000
      duration_sec : 421
      rx_bytes : 36736268
      rx_crc_err : 0
      rx_dropped : 0
      rx_errors : 0
      rx_frame_err : 0
      rx_over_err : 0
```

Figure 3.12: Switch Statistics Port File Example

3.2 Containernet

Containernet is a fork of Mininet, i.e., it is a copy of Mininet with relevant additional features, allowing to use of Docker containers as hosts in emulated networks. Some of Containernet's most relevant features include adding and removing Docker containers to Mininet topologies, connecting containers to switches, or Mininet hosts and allowing the execution of commands inside containers.

Containernet is used to build networking emulators and testbeds, focusing on research related to cloud computing, fog computing, multi-access edge computing, and NFV [7].

Mininet has been used for research, testing, and educational purposes and it is a network emulation orchestration system, in other words, with Mininet it is possible to create a virtual network composed by hosts, switches, routers, and connections between them, using lightweight virtualization. This network emulator system provides a set of features built into Linux allowing a single system to be split into a set of smaller containers.

A Mininet network consists of three components: hosts, links and switches. Hosts are implemented using Linux name-space isolation to enable the possibility to have, for example, two web servers in two network namespaces coexisting independently in one system. The Emulated Links connect the devices inside the Mininet network and are connected to virtual interfaces or virtual switch ports and enable the sending of packets between interfaces with a configured data rate. Finally, the Emulated Switches are implemented using either the Linux bridge or Open vSwitch to switch packets across interfaces.

With this tool is possible to have a real network environment emulated on a single Linux kernel and run and test our programs in it with the advantage of having a trustworthy mirror from real networks.

The benefits of using Mininet are the fast and flexible network creation, the trustfulness in running programs in the emulated network and getting reliable responses, the utilization of OpenFlow protocol in switches for customized packet forwarding, it's open-source, and under active development.

On other hand, Mininet has its limitations, such as the sharing of machine process resources among the virtual hosts and switches, and the need to use a separate SDN controller.

The "ContainernetAPI" class is defined in the "containernet_api_topo.py" file inside of the "envs" folder and is were the user can implement control functions, set up the topology, use functions to obtain information from the Containernet environment and send information to the Ryu Controller and to OpenAI Gym.

More specifically, it is inside of the class "ContainernetAPI" that are functions with particular roles implemented. One of the main functions is named "load_topology", this function creates the containers, switches, and links between the nodes according to the topology file description. This function makes use of functions imported from the Containernet library.

The "add_arps" unction defines the initial ARP rules. The "generate_traffic_with_iperf" function is responsible for creating iperf traffic and saving the statistics related to the specific traffic into a log file inside the "logs" folder. There is also a function that gets information from the logs files called "json_from_log" and another that can return specific statistics from the log files called "get_traffic_stats".

A function named "get_bw_used_bw_available" updates the used bandwidth values and the available bandwidth from the links between the switchers. The "upload_sw_adjacency" function updates the "switches_adjacency.json" file.

A function named "send_path_to_controller" adds information about the path that needs to be installed for a specific containers pair to the file "OFPMatch_params.json" and to the file "active_paths.json". Another important function is the "define_ofp_match_params" function that defines the parameters that can be used when defining the OFPMatch.

A function named "update_container_cpu_limits" to change a container CPU definitions and another function named "add_container_to_topo" which is used to add new containers to the topology. It is also possible to get the container statistics by calling the function "get_container_stats".

There are also other auxiliar functions that write and read data in files to enable the sharing of information between the "ContainernetAPI" class and the Ryu Controller, i.e., "get_data_from_json" and "upload_data_in_json_file".

3.3 Ryu Controller

The Ryu Controller is a open source SDN controller that supports the OpenFlow protocol.

This software is implemented in python in the "ryu_controller.py" file that can be found in the "envs" folder. The auxiliary functions created in this file are responsible to load information from "ContainernetAPI" side and to organize the information to create the rules to be installed by the Controller.

The functions to get information from "ContainernetAPI" are "load_paths", "upload_ topology_information", "upload_bw_links", "get_data_from_json" and the function to adapt the information to be used by the Controller is called "convert_json_ with_key_tuples_into_dict". All of these functions are described in the "ryu_controller.py" file.

In order to obtain the current network status, it is necessary to monitor the network on a regular basis, there are two functions that implement a traffic monitoring system. The "_monitor" function is called every three seconds and uses the "_request_stats" function to send an OFPPortStatsRequest message to each datapath registered in the controller to obtain the port and flow statistics of each datapath.

Before starting receiving rules to install in the switches, during the agent's training, the Ryu Controller is programmed to install the starting rules defined by the user, after all the switches are registered. The function responsible for this is called "install_starting_rules".

The function "add_flow" is responsible for installing a forwarding rule using a Flow Modification OpenFlow message. The "remove_flow" function removes installed rules.

During the training of a DRL agent a function called "update_paths" can be used to install new rules to forward packets and uninstall old ones that are not used, by calling the methods "install_path" and "uninstall_path".

Besides the functions to upload the information about the topology that is set up by the Containernet side, the "ryu_controller.py" file has defined call back functions that are called in specific events.

The "_state_change_handler" call back function, is called in the connection and disconnection of switches from the network when all the switchers are connected, a function to install the starting rules is called.

The "switch_features_handler" function is called, after the handshake between the controller and the switch. In this function a table-miss flow entry in the switch, tells the switch how to handle packets that do not match ant Flow Entry.

Two other callback functions are the "_port_stats_reply_handler" function and the "_flow_stats _reply_handler" function, which receive statistics from the ports and the flow of each switch.

The function "_port_stats_reply_handler" is also responsible to create the folders and files related to the switches' statistics, as mentioned previously. And it is also responsible to download the more recent paths from the "active_paths.json" file and installing them.

3.4 OpenAI Gym

OpenAI Gym is a framework that provides an interface that gives to an DRL agent the access to execute actions and to observe the state in an environment.

The Gym toolkit is used for developing and comparing RL algorithms. The Gym library provides many types of environments to be used for the algorithm tests. Some of these environments are for simulating robots, classic control scenarios, Atari games, continuous control tasks, etc.

Considering the advantages of RL as a tool to solve decision making problems in hard to model complex networking scenarios, there is a need for simulation tools that can easily provide a simulated environment to train DRL agents.

Such tools can also provide a common benchmarking environment to allow the direct comparison of different DRL architectures.

The OpenAI Gym was used on the "containernetenv.py" file and can be found in the "envs" folder.

It is in this file that the user will define all the functions that it will use to communicate with the DRL agent, along with an object from the "ContainernetAPI" class to have access to Containernet and the features created by the author.

The "Environment Setup" section describes the contents of this file.

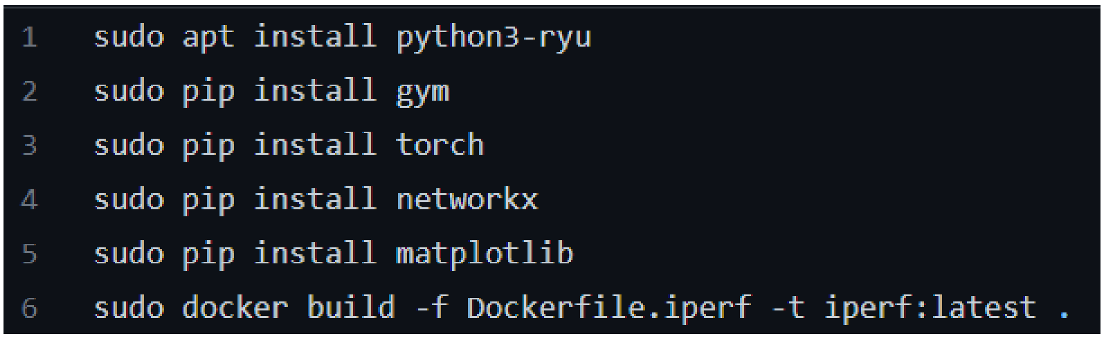
3.5 Environment Setup

This section describes the setup of all the software needed to use the developed environment simulation tool.

Containernet and Python 3 should be installed previously, and the instructions can be found on Containernet Github. The installation of Ryu Controller is then made by running the "setup.sh" bash file that can be found on the projects folder "ai_gym_for_networks" in Github Figure 3.13, and it is also used to install the dependencies related to the python libraries, i.e., "gym", "torch", "networkx" and "matplotlib".

The gym library is used to compare RL algorithms, the torch library is used for tensor computation with strong Graphics Processing Unit (GPU) acceleration and to build deep neural networks, the networkX library is used for graph analysis and finally matplotlib is used for plotting the results obtained in the simulation.

The bash file last line creates a Docker container labeled "iperf:latest" that will be used to create the different hosts for the simulated scenarios.



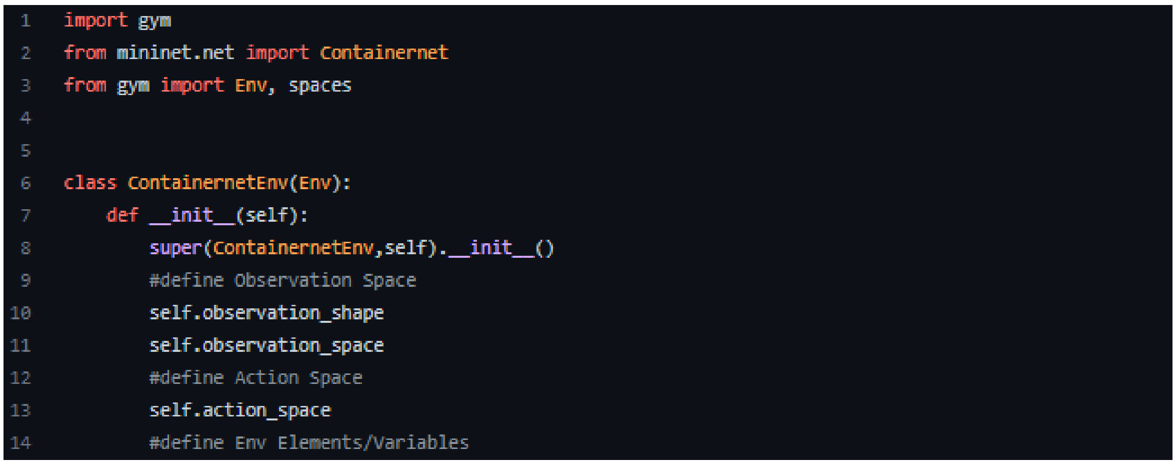
```
sudo apt install python3-ryu
sudo pip install gym
sudo pip install torch
sudo pip install networkx
sudo pip install matplotlib
sudo docker build -f Dockerfile.iperf -t iperf:latest .
```

Figure 3.13: Setup Bash File

After having all the dependencies installed, the user will open the "containernetenv.py" file that can be found inside of "envs" folder and starts to define the variables and functions that the DRL agent will need to communicate with the training environment. Figure 3.14.

The observation space defines the structure to observe the state of the environment before the agent chooses an action.

The action space defines what are the possible actions that can be taken by the agent to interact with the environment.



```python
import gym
from mininet.net import Containernet
from gym import Env, spaces


class ContainernetEnv(Env):
    def __init__(self):
        super(ContainernetEnv,self).__init__()
        #define Observation Space
        self.observation_shape
        self.observation_space
        #define Action Space
        self.action_space
        #define Env Elements/Variables
```

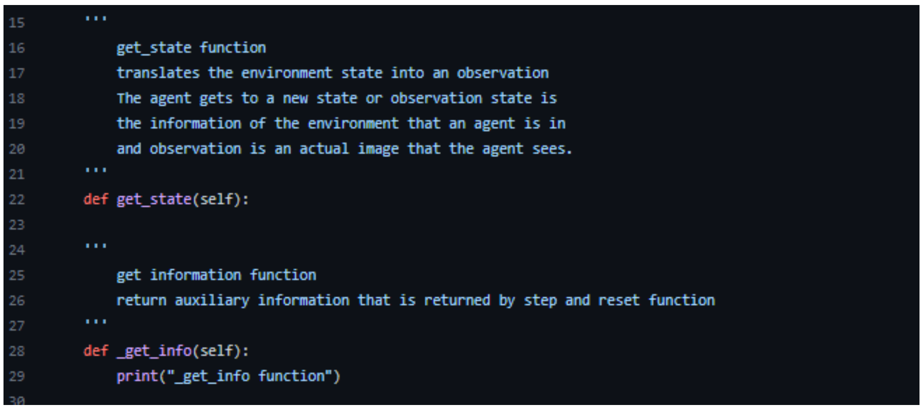
Figure 3.14: Constructor - ContainernetEnv Class

It is necessary to instantiate the "ContainernetAPI" class object that receives as input the topology file name of the file that contains the topology the user wants to implement.

Then the user needs to develop a function that creates the files where the information about the starting rules that will be installed in Ryu Controller is described.

After that, the user needs to define a "get_state" function to return the environment state, the "_get_info" function which returns auxiliary information from the environment and finally needs to define the most important functions. Figure 3.15.

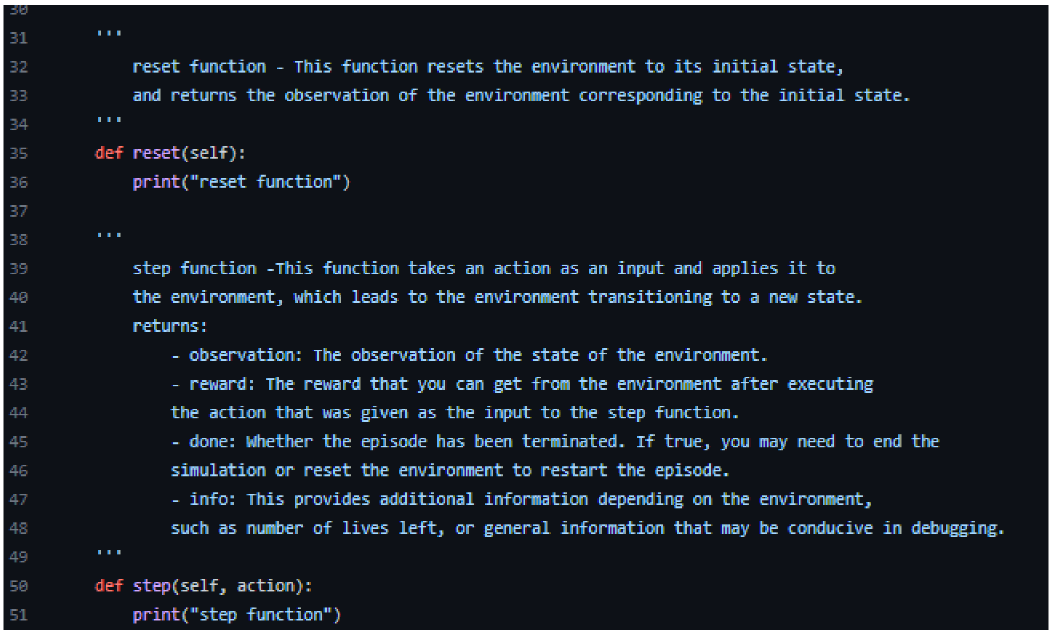
The "reset" function is where the user will define how the environment will be reset to its initial state, returning the environment's observation to the initial state.



```python
    '''
        get_state function
        translates the environment state into an observation
        The agent gets to a new state or observation state is
        the information of the environment that an agent is in
        and observation is an actual image that the agent sees.
    '''
    def get_state(self):

    '''
        get information function
        return auxiliary information that is returned by step and reset function
    '''
    def _get_info(self):
        print("_get_info function")
```

Figure 3.15: Get State and Get Information Functions - ContainernetEnv Class

The "step" function receives an action as input and uses it to change the state of the environment. By doing that, the environment will return a "reward" that indicates the performance of the specific action, an "observation" or, in other words, the next environment state, and the variable "done" indicating if the step terminated the epoch. There is also a fourth variable called "information" that can be added to provide additional information depending on the environment that the user built.Figure 3.16.



```python
    '''
        reset function - This function resets the environment to its initial state,
        and returns the observation of the environment corresponding to the initial state.
    '''
    def reset(self):
        print("reset function")

    '''
        step function -This function takes an action as an input and applies it to
        the environment, which leads to the environment transitioning to a new state.
        returns:
            - observation: The observation of the state of the environment.
            - reward: The reward that you can get from the environment after executing
            the action that was given as the input to the step function.
            - done: Whether the episode has been terminated. If true, you may need to end the
            simulation or reset the environment to restart the episode.
            - info: This provides additional information depending on the environment,
            such as number of lives left, or general information that may be conducive in debugging.
    '''
    def step(self, action):
        print("step function")
```

Figure 3.16: Reset and Step Functions - ContainernetEnv Class

After setting up the environment, the DRL algorithm the user wants to test must be defined. If the user wants to have some examples of DRL agents or environments, they can be found in the folders named "agents" and "env_examples".

The next chapters describe how three different scenarios were created in order to demonstrate how different scenarios can be set up using the same code base.

4

Developed Scenarios

The following chapter focuses on describing application examples of the developed simulator, illustrating the use of the simulator developed during the thesis and simultaneously explaining how each example was implemented. The first two examples were taken from previous works and the last one was developed in the thesis.

It was necessary to develop two DRL algorithms. Respectively, DQNs and Dueling DQNs, to be trained in the environment examples that will be presented in the this chapter.

The code implementation of the agents was inspired by the implementation of the work reported in [31] and can be found in the "agents" folder of the "AI Gym for Networks" project.

4.1 Network Path Selection - Example 1

4.1.1 System Design

The first example replicates an experience from the paper reported in [31] that applies a DRL algorithm to select the best path to transport a video stream between a pair of hosts. This example can be found in our repository inside of the "network_path_selection" folder in the "env_examples" folder. Figure 4.1.

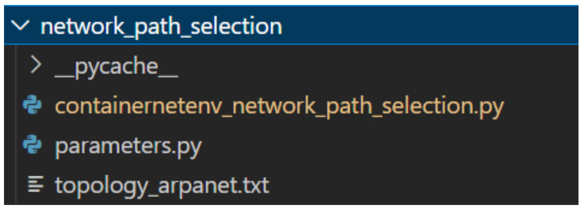

```
network_path_selection
  __pycache__
  containernetenv_network_path_selection.py
  parameters.py
  topology_arpanet.txt
```

Figure 4.1: Network Path Selection Folder - Example 1

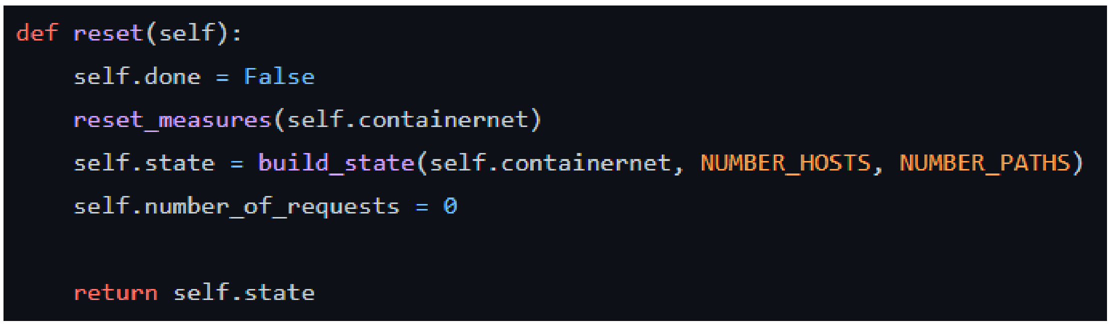
The implemented "Network Path Selection - Example 1" can be found on "containernetenv_network_path_selection.py", and contains all the functions and variables referred in section 3.5 and also specific functions needed to create the example.

The environment is composed of a state space which is a four-dimensional tensor with dimensions [N, N, k, 1], where N represents the number of hosts and k represents the pre-computed number of paths per host pair. The last value represents the available bandwidth in the bottleneck link of each path, i.e., the link with the smallest available bandwidth.

The action space is defined by discrete values with a range of the number of pre-computed paths per host pair. The reward results are obtained based on the available path bandwidth in the resulting state and are influenced by the action used in a specific environment state. The bandwidth is calculated on the Ryu Controller side, by receiving the port statistics from each datapath that composed the topology.

The program starts with the installation in the Ruy Controller, then an algorithm that obtains the k shortest paths is used to calculate the paths that make up the action space of the DRL agents for each host pair. The bandwidth values are then changed by the traffic, created using the iperf tool via Containernet.

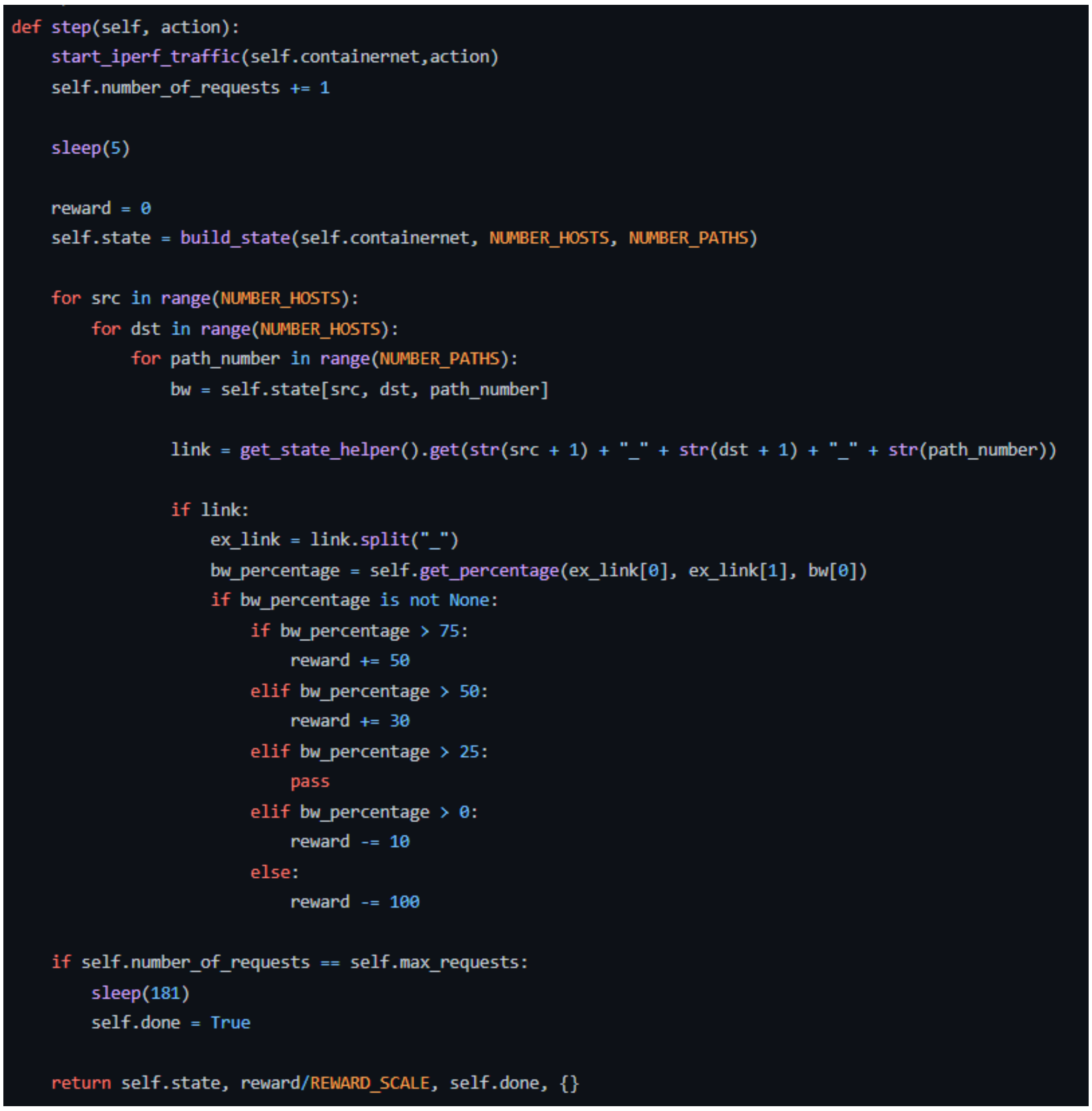
In the "reset" function the "reset_measures" function to reset the variables related with the active paths and the available bandwidth, and then the initial state is calculated. Figure 4.2 shows the code.



```python
def reset(self):
    self.done = False
    reset_measures(self.containernet)
    self.state = build_state(self.containernet, NUMBER_HOSTS, NUMBER_PATHS)
    self.number_of_requests = 0

    return self.state
```

Figure 4.2: Reset Function - Example 1

The "step" function gets the available bandwidth between each active communicating pair and calculates the reward resulting from the percentage of the available bandwidth. Figure 4.3.



```
def step(self, action):
    start_iperf_traffic(self.containernet,action)
    self.number_of_requests += 1

    sleep(5)

    reward = 0
    self.state = build_state(self.containernet, NUMBER_HOSTS, NUMBER_PATHS)

    for src in range(NUMBER_HOSTS):
        for dst in range(NUMBER_HOSTS):
            for path_number in range(NUMBER_PATHS):
                bw = self.state[src, dst, path_number]

                link = get_state_helper().get(str(src + 1) + "_" + str(dst + 1) + "_" + str(path_number))

                if link:
                    ex_link = link.split("_")
                    bw_percentage = self.get_percentage(ex_link[0], ex_link[1], bw[0])
                    if bw_percentage is not None:
                        if bw_percentage > 75:
                            reward += 50
                        elif bw_percentage > 50:
                            reward += 30
                        elif bw_percentage > 25:
                            pass
                        elif bw_percentage > 0:
                            reward -= 10
                        else:
                            reward -= 100

    if self.number_of_requests == self.max_requests:
        sleep(181)
        self.done = True

    return self.state, reward/REWARD_SCALE, self.done, {}
```

Figure 4.3: Step Function - Example 1

The "parameters.py" file is where the parameters of the DRL algorithms, that define the characteristics related to the paths to store, the needed files and also the number of hosts, switches and paths used in the scenario are defined.

The parameters related to the agent's neural network used to define the layers of each agent were defined by experimentation and can be found in the table 4.1 for the DQN agent and in table 4.2 for the Dueling-DQN agent. The DRL algorithms used to train the agents for this scenario are described in section 2.9 and their results will be discussed in the "Results" chapter.

Table 4.1: Agent's networks layers DQN - Example 1

Type	Input Size	Output Size	Activation Function
Linear	845	1500	ReLU
Linear	1500	700	ReLU
Linear	700	200	ReLU
Linear	200	5	ReLU

Table 4.2: Agent's networks layers Dueling DQN - Example 1

Type	Input Size	Output Size	Activation Function
Linear	845	1500	ReLU
Linear	1500	700	ReLU
Linear (Value)	700	200	ReLU
Linear (Value)	200	5	-
Linear (Advantage)	700	200	ReLU
Linear (Advantage)	200	5	-

Table 4.3: Agent's parameters - Example 1

Parameter	Value
Epochs	2500
ϵ	0.5
γ	0.99
Replay Buffer Size	50000
Batch Size	256
Learning Rate	0.001

The topology implemented in Containernet has thirteen hosts and twenty switchers and is shown in Figure 4.4.

The topology is described in file "topology_arpanet.txt" that is store on "env_examples/ network_path_selection" folder. The file specifies the configuration links between hosts and switches and the characteristics from each link, in this case the link bandwidth capacity.

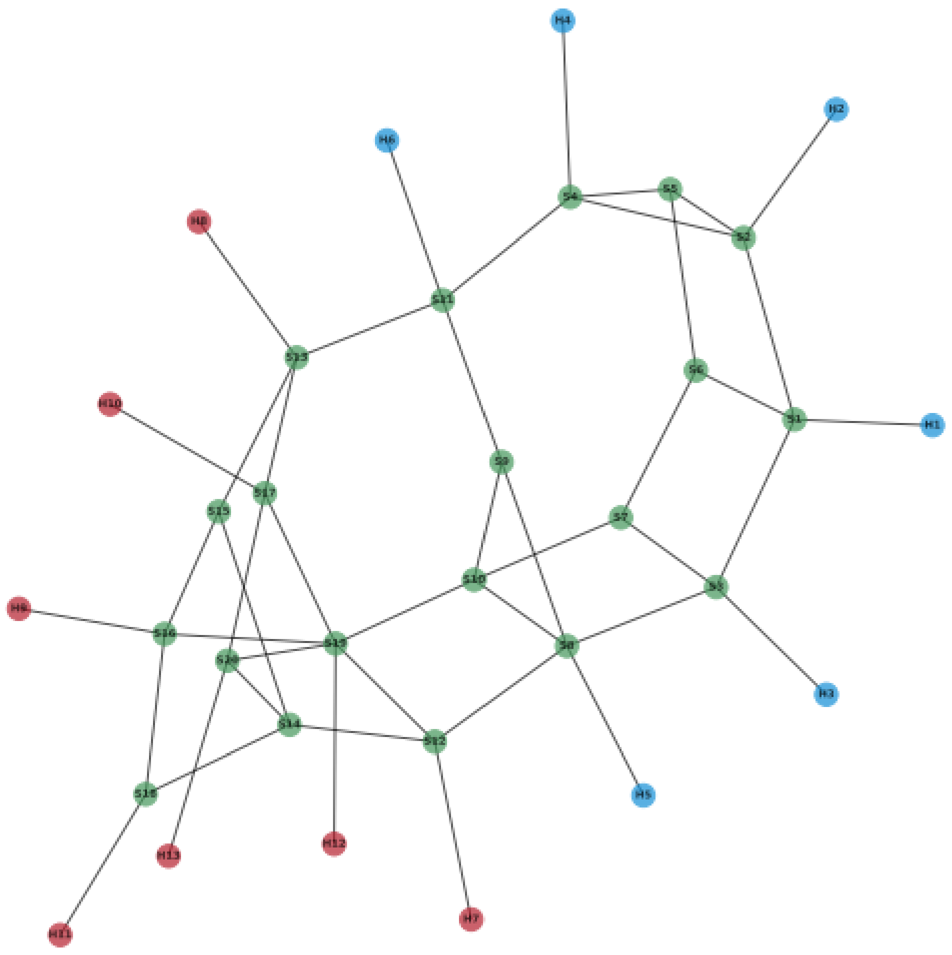
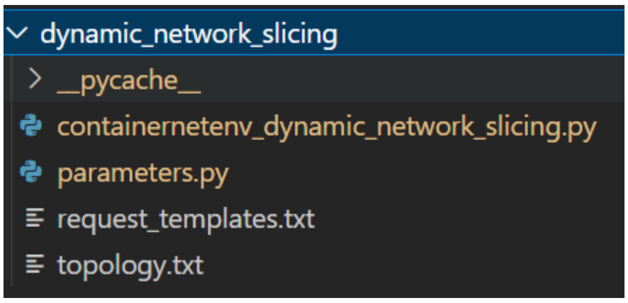


Figure 4.4: Topology - Example 1

4.2 Dynamic Network Slicing - Example 2

4.2.1 System Design

The second example is based on the problem addressed in the work reported in [32]. It concerns the use of a DRL algorithm to address the problem of optimal slices' admission in transport networks. This example can be found in the "dynamic_network_slicing" folder, on the "env_examples" folder. Figure 4.5.



```
dynamic_network_slicing
  > __pycache__
  containernetenv_dynamic_network_slicing.py
  parameters.py
  request_templates.txt
  topology.txt
```

Figure 4.5: Dynamic Network Slicing Folder - Example 2

The implementation can be found on the "containernetenv_dynamic_network_slicing.py" file. The scenario has two main components, the tenants represented by the requests generator, and the operator that defines the prices for each slice request. The DRL algorithm chooses if it accepts or not the slice admission in the network.

First, we have the request, composed of the slice type, the slice duration, the bandwidth, the price, and the needed connectivity between the BSs and the computing stations. The information about the slice type, the bandwidth and the price per second is downloaded from the "request_templates.txt" file.

The number of elastic and inelastic slice requests follow probability distributions, to regulate the frequency of requests for each slice type. The request duration and the number of needed connections are obtained through exponential distributions. After that, the connections are picked randomly to know between each host pairs the creation of a slice will be made.

The goal of this model is to use a DRL agent and optimize the operator profit by responding to the slice requests sent by the tenants, deciding when to accept or not the request based the decision on profit and the network state.

The paths installation is decided by choosing the path with a higher bottleneck, which is the path with the link with lower bandwidth. These bandwidth values are obtain by the Ryu Controller, using the port statistics of the typology's datapaths.

The use of DRL agents creates the need to set up an environment with specific characteristics.

The state space is obtained by having the request information, the number of elastic slices, the number of inelastic slices, and the value of the bottleneck of each calculated path for each pair of BSs and computing stations of the slice.

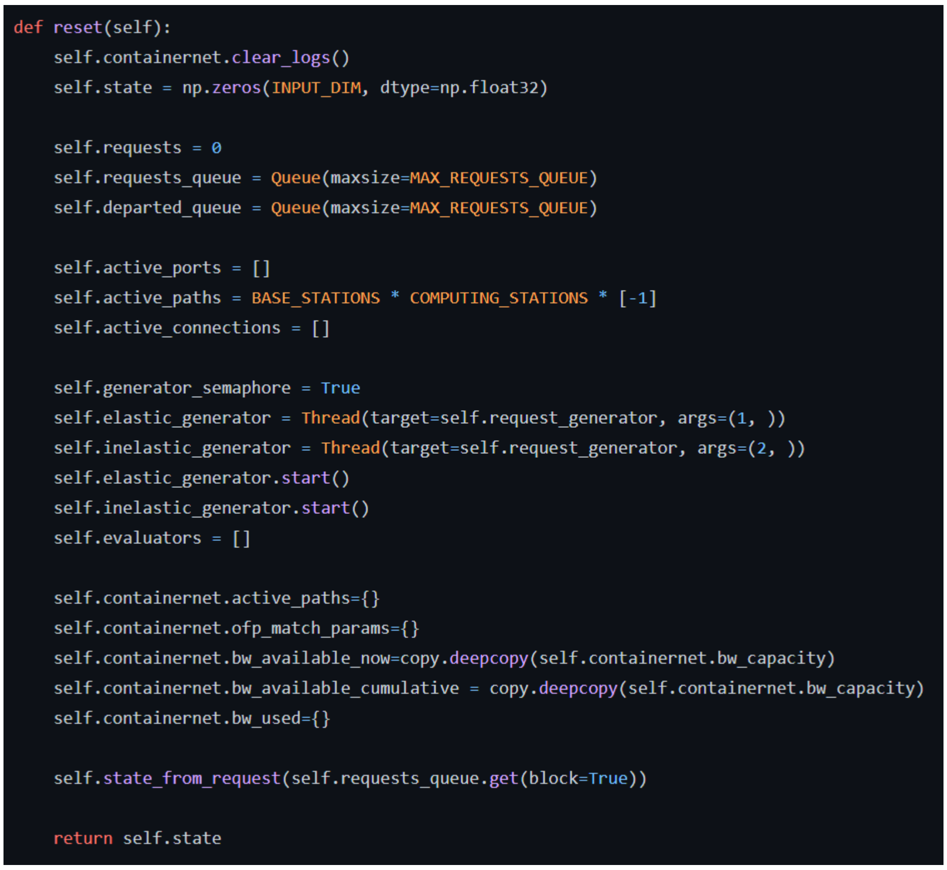
The action space is composed of two discrete numbers representing accepting or rejecting the received request.

The reward is obtained by multiplying the price of a slice by its duration. When a slice ends, it is evaluated and the result will influence the reward.

For elastic slices, the average of the measured bandwidth needs to be equal to or higher than the requested bandwidth, if not the reward will be half of its value.

For inelastic slices, all the bandwidth measurements need to be equal to or higher than the requested bandwidth. If not, the reward will be zero. If all the requirements are respected the reward remains intact.

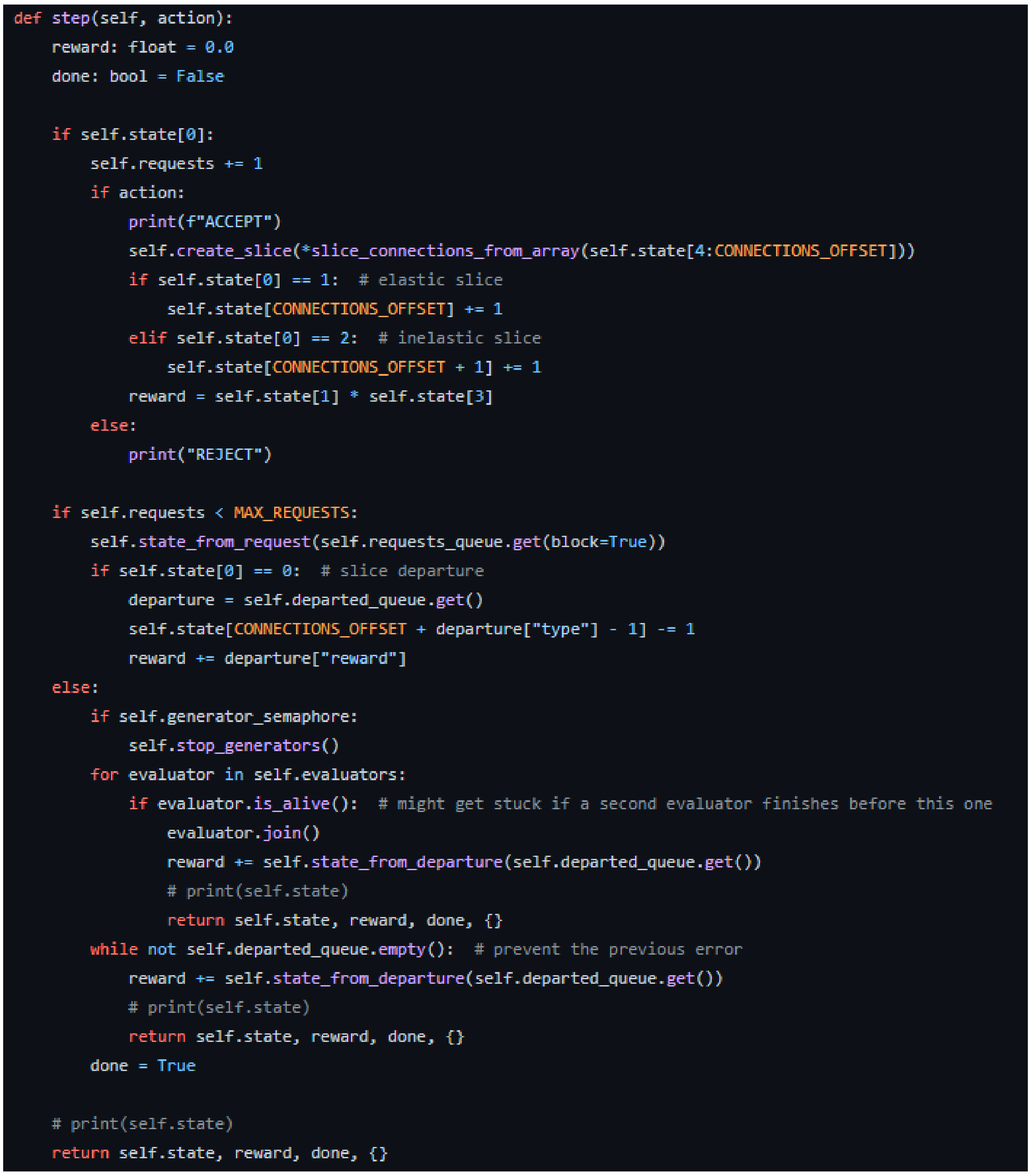
The "reset" function cleans the "logs" folder from the last epoch and then restarts the variables needed for the next step: the queues to save the requests; the threads to generate and validate the slices; the variables related with the bandwidth values; the active paths and the connections. Finally, the next environment state is generated and returned it. The "reset" function can be seen in Figure 4.6.



```
def reset(self):
    self.containernet.clear_logs()
    self.state = np.zeros(INPUT_DIM, dtype=np.float32)

    self.requests = 0
    self.requests_queue = Queue(maxsize=MAX_REQUESTS_QUEUE)
    self.departed_queue = Queue(maxsize=MAX_REQUESTS_QUEUE)

    self.active_ports = []
    self.active_paths = BASE_STATIONS * COMPUTING_STATIONS * [-1]
    self.active_connections = []

    self.generator_semaphore = True
    self.elastic_generator = Thread(target=self.request_generator, args=(1, ))
    self.inelastic_generator = Thread(target=self.request_generator, args=(2, ))
    self.elastic_generator.start()
    self.inelastic_generator.start()
    self.evaluators = []

    self.containernet.active_paths={}
    self.containernet.ofp_match_params={}
    self.containernet.bw_available_now=copy.deepcopy(self.containernet.bw_capacity)
    self.containernet.bw_available_cumulative = copy.deepcopy(self.containernet.bw_capacity)
    self.containernet.bw_used={}

    self.state_from_request(self.requests_queue.get(block=True))

    return self.state
```

Figure 4.6: Reset Function - Example 2 and Example 3

The "step" function receives the action value representing the agent's acceptation or rejection for the request. If the agent accepts the request, the slice is created and evaluated based on that request. Then the reward is stored and this procedure occurs until the requests reach the max requests value, after that the program will wait until all the remain evaluation threads are finished and then send their rewards to the agent, along with the environment state and the value of the done variable. The "step" function code can be found on Figure 4.7.



```python
def step(self, action):
    reward: float = 0.0
    done: bool = False

    if self.state[0]:
        self.requests += 1
        if action:
            print(f"ACCEPT")
            self.create_slice(*slice_connections_from_array(self.state[4:CONNECTIONS_OFFSET]))
            if self.state[0] == 1:  # elastic slice
                self.state[CONNECTIONS_OFFSET] += 1
            elif self.state[0] == 2:  # inelastic slice
                self.state[CONNECTIONS_OFFSET + 1] += 1
            reward = self.state[1] * self.state[3]
        else:
            print("REJECT")

    if self.requests < MAX_REQUESTS:
        self.state_from_request(self.requests_queue.get(block=True))
        if self.state[0] == 0:  # slice departure
            departure = self.departed_queue.get()
            self.state[CONNECTIONS_OFFSET + departure["type"] - 1] -= 1
            reward += departure["reward"]
    else:
        if self.generator_semaphore:
            self.stop_generators()
        for evaluator in self.evaluators:
            if evaluator.is_alive():  # might get stuck if a second evaluator finishes before this one
                evaluator.join()
                reward += self.state_from_departure(self.departed_queue.get())
                # print(self.state)
                return self.state, reward, done, {}
        while not self.departed_queue.empty():  # prevent the previous error
            reward += self.state_from_departure(self.departed_queue.get())
            # print(self.state)
            return self.state, reward, done, {}
        done = True

    # print(self.state)
    return self.state, reward, done, {}
```

Figure 4.7: Step Function - Example 2

The "parameters.py" file is where the DRL algorithms parameters of the table 4.6 are defined together with the paths to store and consult the needed files and also the number of hosts, switches and paths used in the scenario.

The parameters related to the agent's neural network used to define the layers of each agent were defined by experimentation and can be found in the table 4.4 for the DQN agent and in table 4.5 for the Dueling-DQN agent.

Table 4.4: Agent's networks layers DQN - Example 2 and 3

Type	Input Size	Output Size	Activation Function
Linear	790	1800	ReLU
Linear	1800	1200	ReLU
Linear	1200	800	ReLU
Linear	800	1	ReLU

Table 4.5: Agent's networks layers Dueling DQN - Example 2 and 3

Type	Input Size	Output Size	Activation Function
Linear	790	1800	ReLU
Linear	1800	1200	ReLU
Linear (Value)	1200	800	ReLU
Linear (Value)	800	1	-
Linear (Advantage)	1200	800	ReLU
Linear (Advantage)	800	1	-

Table 4.6: Agent's parameters - Example 2 and 3

Parameter	Value
Epochs	1300
ϵ	0.3
γ	0.99
Replay Buffer Size	1000
Batch Size	200
Learning Rate	0.001

The chosen topology to be used in this example is represented in Figure 4.8 and it is described in file "topology.txt" that is stored on the "env_examples/dynamic_network_slicing" folder. The file specifies the configuration links between hosts and switches and the characteristics for each link, in this case the link bandwidth capacity, the link delay and the link loss. For this example there are three types of hosts, the Mobile Edge Computing

Station (MECS)s, the Computing Station (CS)s and the BSs. The MECSs are used to give computation and storage power to the access network layer. The CSs are used for more demanding computation and the BSs simulates a Radio Network Acess point.

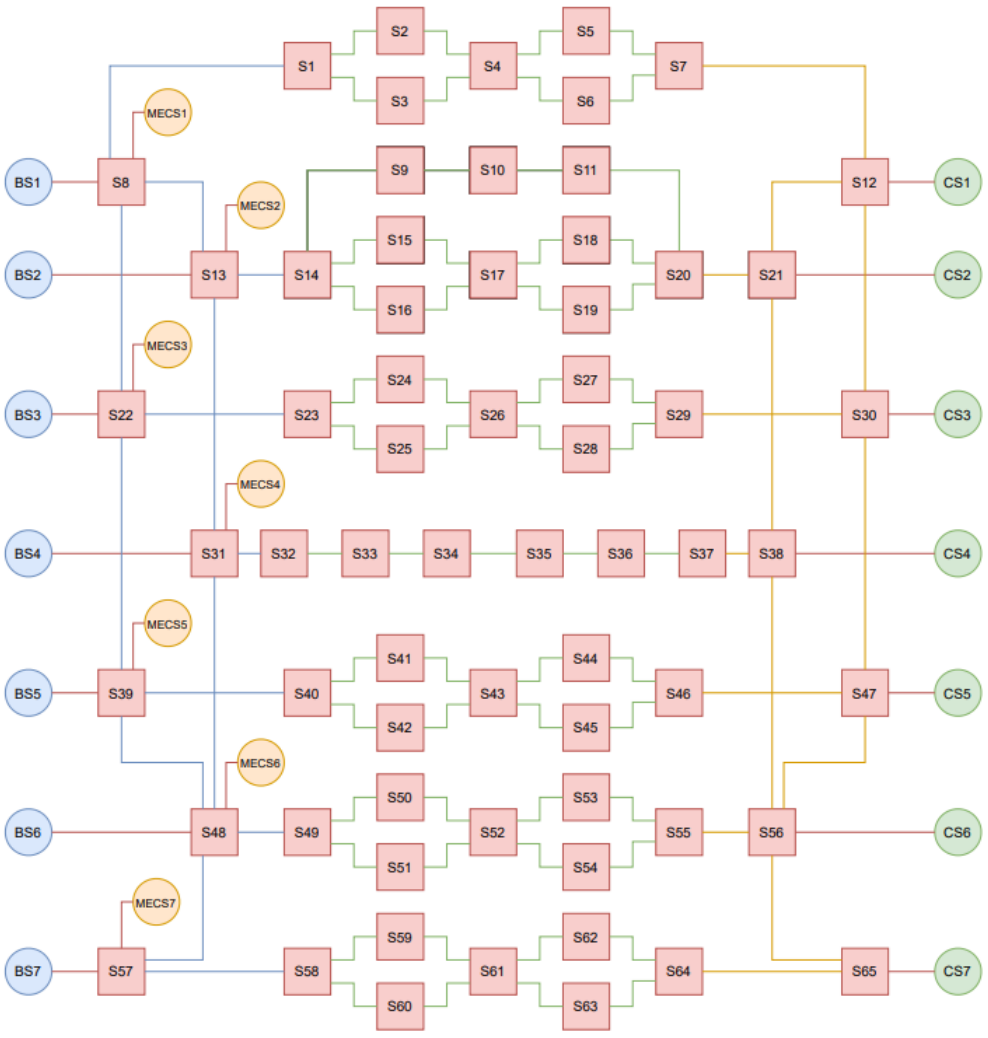
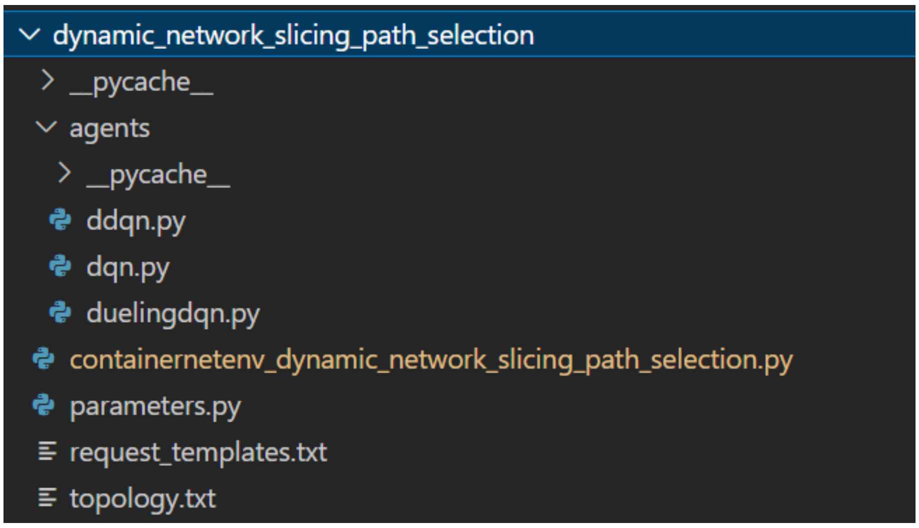


Figure 4.8: Topology - Example 2 and Example 3

4.3 Dynamic Network Slicing and Path Selection - Example 3

4.3.1 System Design

In the third example a new model was developed to jointly optimize network slice admission control and slice traffic routing. In other words, this example has the main goal of presenting a possible solution to solve a problem related to the slices' admission to transport networks and at the same time optimize the selection path algorithm, by using DRL agents. This example is inside of the "dynamic_network_slicing_path_selection" folder, which can be found on "env_examples" folder. Figure 4.9.



```
dynamic_network_slicing_path_selection
  __pycache__
  agents
    __pycache__
    ddqn.py
    dqn.py
    duelingdqn.py
  containernetenv_dynamic_network_slicing_path_selection.py
  parameters.py
  request_templates.txt
  topology.txt
```

Figure 4.9: Dynamic Network Slicing and Path Selection Folder - Example 3

In order to allow a comparison between the agents performance in example 2 and example 3 the same topology was chosen. It is represented in Figure 4.8 and it is described in file "topology.txt" on the "env_examples/dynamic_network_slicing" folder.

The model differs from the one of the previous section in an important aspect. In the problem addressed in example 2, only the admission control is considered, with the slice traffic being always placed by following a rule that define the best path between two containers as the one with the best bottleneck. In this model with also consider the routing of slice traffic, that is also decided by the DRL agent. This leads to a different agent architecture. In the two previous examples, the DRL agents only had the capacity to deliver one action per step, in other words, these agents just influenced one aspect of the environment.

The DRL agents developed for this new model have the capacity to influence two aspect of the environment by choosing two actions for two different tasks. One to select the paths to transport slice traffic, and the other to choose to accept new slices admissions or not.

The new action space must include both the action of accepting or rejecting network slices as well as the choice of the paths that carry the traffic of each slice. This means that the state space is now a tensor with dimensions [N P] were | N | = 2, accept or reject a slice, and P is the number of considered paths.

Two different approaches could be followed to deal to the action space. The first one is to have a single agent with an output layer corresponding to the action space size of N * P. This size increases exponentially with the increase of the considered paths P and in our case is 2*6 = 12.

Another option is to divide the problem in to different tasks, each with its own action space. In this later case we need two agent networks, one with an output layer of size N that calculates the action values for accepting or rejecting slices, and another with output of size P to calculate the value of choosing each of the P paths. This later option is more complex having more neural networks but it scales better with the increase of the number of paths in P. We opted for this later configuration and therefore, the structure of each DRL algorithm described in section 2.9 was duplicated, to deal with the the accepting/rejecting requests actions and selection of the best path to be installed in the controller in to separate agent networks.

The implementation of the two algorithms can be found inside of the "agents" folder on the "dynamic_network_slicing_path_selection" folder.

Each agent network receive in the input layer the environment state, the reward and the respective action for the controller policy and return the next action to be executed on the environment. So one agent network will calculate the weights to choose the better path and the other will calculate the weights to decide if accepts the requests.

Algorithm 1 illustrates the training loop in the DQN agent with two separate tasks. For the Dueling-DRL the same logic is followed but taking into account its particularities, which are described in section 2.9.

First, there are initialized two replay buffers, two Q-Networks, and two target networks for the respective agents. Then each agent network returns an action that will affect the environment state. The environment reacts by returning a new environment state and a reward. These values are stored in the respective replay buffers, to later be used to generate new samples to update the agent networks and periodically, the target networks. This process is repeated until the defined requests to the environment are reached.

Algorithm 1 Training environment

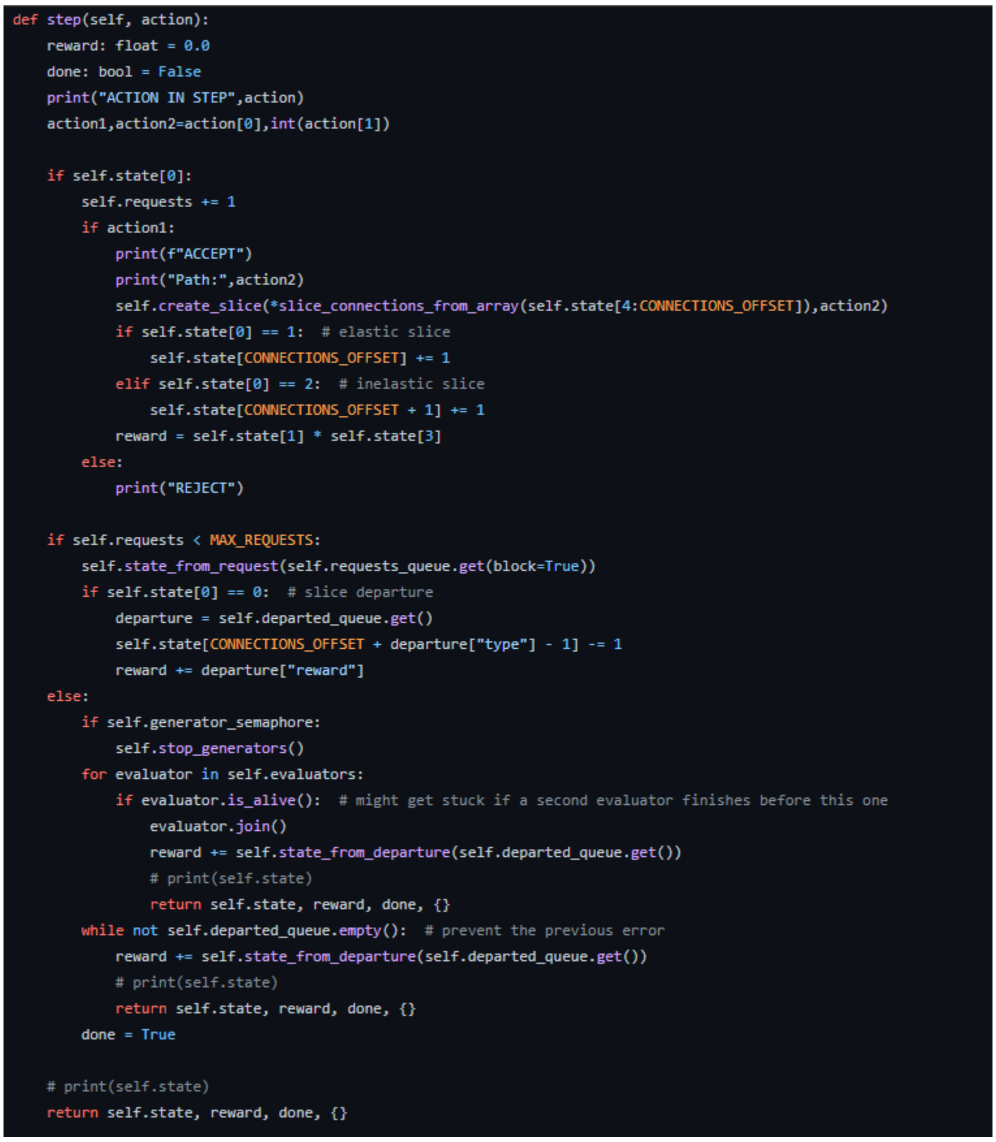
```
$replayBuffer1 \leftarrow ReplayBuffer()$
$replayBuffer2 \leftarrow ReplayBuffer()$
$model1 \leftarrow Q_Network()$
$model2 \leftarrow Q_Network()$
$targetModel1 \leftarrow model1$
$targetModel2 \leftarrow model2$
for $i$ episodes do
    $requests \leftarrow 0$
    $done \leftarrow False$
    $maxRequests \leftarrow N$
    while $notdone$ do
        $action1 \leftarrow argmax(model1, state)$
        $action2 \leftarrow argmax(model2, state)$
        $transformedState \leftarrow makeReservation(action1, action2)$
        $reward \leftarrow evaluate(transformedState)$
        $requests \leftarrow requests + 1$
        $replayBuffer1 \leftarrow Add(state, action1, reward, transformedState)$
        $replayBuffer2 \leftarrow Add(state, action2, reward, transformedState)$
        $model1 \leftarrow updateModel(sample(replayBuffer1))$
        $model2 \leftarrow updateModel(sample(replayBuffer2))$
        if $i == updatefrequency$ then
            $targetModel1 \leftarrow model1$
            $targetModel2 \leftarrow model2$
        end if
        if $requests == maxRequests$ then
            $done \leftarrow True$
        end if
    end while
end for
```

The "parameters.py" file contains the specifications of each DRL agent and their ANN layers as in the previous example.

The "reset" function is equal to the Example 2 "reset" function and it is represented in Figure 4.6, however the "step" function has some differences.

In this case the "step" function receives the action which represents the agent's acceptation or rejection for the request and also receives the action that will choose what will be the path to be installed for the requested slice.

Then the reward from the evaluation is stored and this procedure occurs until the requests reach the max requests value, after that the program will wait until all the remain evaluation threads are finished and then send their rewards to the agent, along with the environment state and the value of the done variable. The "step" function code can be found on Figure 4.10.



```python
def step(self, action):
    reward: float = 0.0
    done: bool = False
    print("ACTION IN STEP",action)
    action1,action2=action[0],int(action[1])

    if self.state[0]:
        self.requests += 1
        if action1:
            print(f"ACCEPT")
            print("Path:",action2)
            self.create_slice(*slice_connections_from_array(self.state[4:CONNECTIONS_OFFSET]),action2)
            if self.state[0] == 1:  # elastic slice
                self.state[CONNECTIONS_OFFSET] += 1
            elif self.state[0] == 2:  # inelastic slice
                self.state[CONNECTIONS_OFFSET + 1] += 1
            reward = self.state[1] * self.state[3]
        else:
            print("REJECT")

    if self.requests < MAX_REQUESTS:
        self.state_from_request(self.requests_queue.get(block=True))
        if self.state[0] == 0:  # slice departure
            departure = self.departed_queue.get()
            self.state[CONNECTIONS_OFFSET + departure["type"] - 1] -= 1
            reward += departure["reward"]
    else:
        if self.generator_semaphore:
            self.stop_generators()
        for evaluator in self.evaluators:
            if evaluator.is_alive():  # might get stuck if a second evaluator finishes before this one
                evaluator.join()
                reward += self.state_from_departure(self.departed_queue.get())
                # print(self.state)
                return self.state, reward, done, {}
        while not self.departed_queue.empty():  # prevent the previous error
            reward += self.state_from_departure(self.departed_queue.get())
            # print(self.state)
            return self.state, reward, done, {}
        done = True

    # print(self.state)
    return self.state, reward, done, {}
```

Figure 4.10: Step Function - Example 3

5 Results

5.1 Results

This section presents the results obtained by training the agents DQN and Dueling DQN in the three examples. The training results will be present in the form of plots, of the reward as a function of epochs. The reward per epoch is the sum of all the step rewards occurred in a epoch. To smooth the plots, averages of batches of epochs were used, so the real horizontal axis value of each plot is obtained by multiplying it by the size of the batch indicated of the figure, in this case batches of 100.

The DRL agents were trained on a machine with the following specifications:

- System:
  - Host: compute-Standard-PC-Q35-ICH9-2009
  - Kernel: 5.15.0-46-generic x86_64
  - bits: 64
  - Desktop: Xfce 4.14.2
  - Distro: Ubuntu 20.04.3 LTS (Focal Fossa)
  - RAM: 9.96 GiB
  - CPU:
    - 2x Single Core: Intel Core (Haswell no TSX IBRS)
    - type: SMP
    - speed: 1898 MHz

The first two examples discussed in the last chapter took an average of 2 days to be implemented. The third example took three days due to the fact that the DRL agents have a more complex architecture, as described in chapter 4.

The training time did not vary much in the case of the built examples. The DQN agent took an average of 4 days to run 1000 epochs, while the Dueling-DQN agent took 5 days to do 1000 epochs on average.

It is important to highlight that the results obtained by the examples 1 and 2, built using the software developed for the thesis are similar to the scenarios that served as inspiration.

The graphs presented in the figures 5.1 for example 1,5.2 for example 2 and 5.3 for example 3 demonstrate that the Dueling-DQN agent can obtain better results than the DQN agent over the trained epochs. It can also be observed that, in the example 1 and example 3, the Dueling-DQN agent needs less epochs to obtain higher rewards, concluding that it is a faster learner.

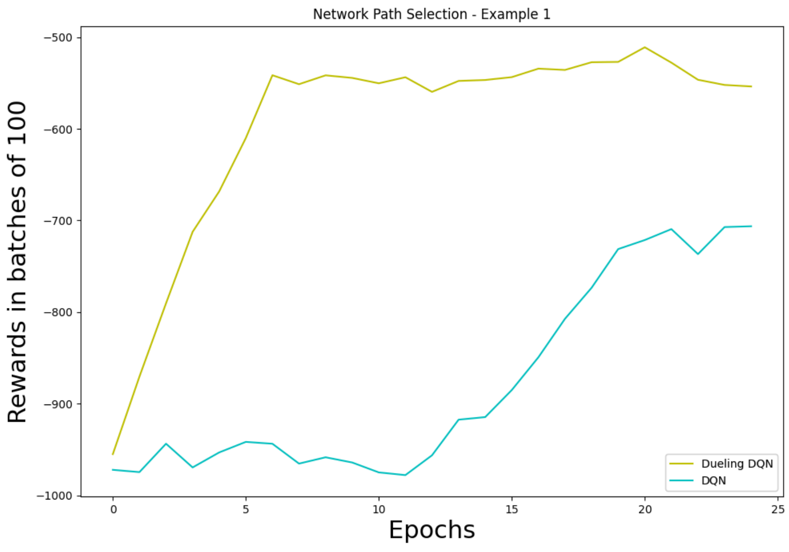


Figure 5.1: Results from Network Path Selection - Example 1

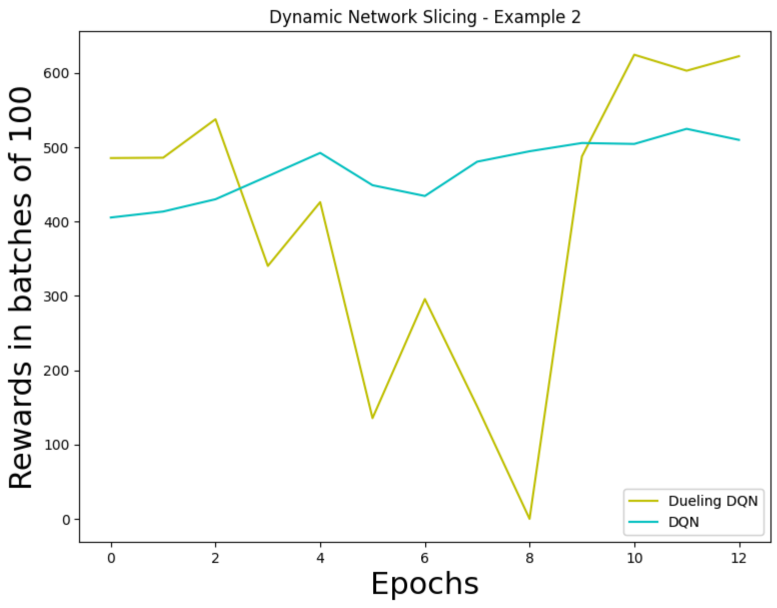


Figure 5.2: Results from Dynamic Network Slicing - Example 2

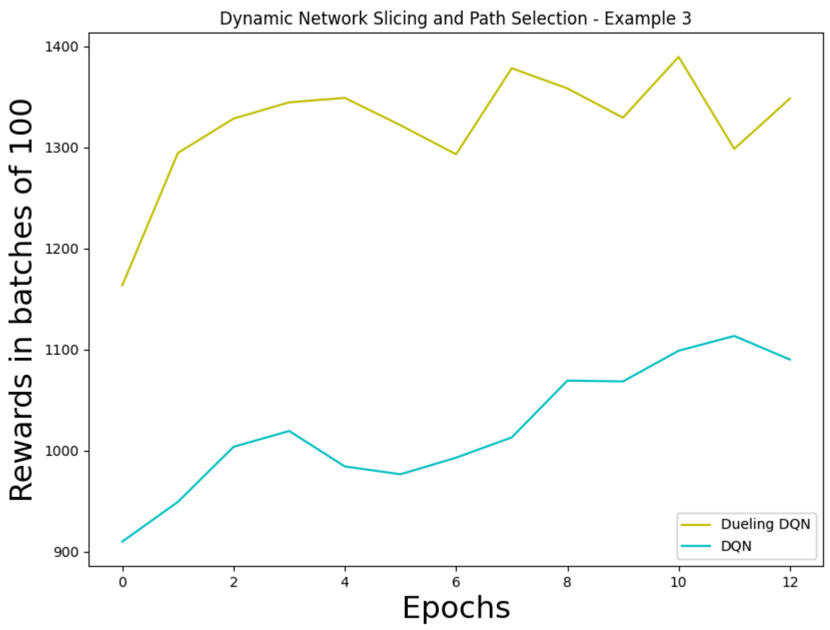


Figure 5.3: Results from Dynamic Network Slicing and Path Selection - Example 3

5.2 Comparison between Example 2 and Example 3

One of the objectives of the thesis is to demonstrate the advantages of creating different examples in a short period of time, making it possible to compare results from different scenarios using the same development environment.

The comparison of the results generated by the examples " Dynamic Network Slicing" and "Dynamic Network Slicing and Path Selection" is possible because they were trained and evaluated using the same environment. The third example uses a DRL algorithm, developed during the thesis, to jointly optimize Network Slice Admission and slice traffic routing, while in the model in example 2 only admission control is considered and the Traffic is always forwarded using the least cost path.

When looking at the figures 5.4 and 5.5, we can see that there is a great gain in terms of reward in the joint optimization model, were the developed DRL agents choose the better path to take instead of following a predefined rule that chooses the path with lower bottleneck.

In figure 5.5, we can also see that the joint model is quicker to convergence in the same period of time.

It appears that considering joint routing optimization in network slice admission can lead to an increase in the operator profit. A performance test using the trained agents is needed to confirm that the increase in reward translates to measurable gains by testing the amount of accepted and satisfied slices using both trained agents in the same conditions.

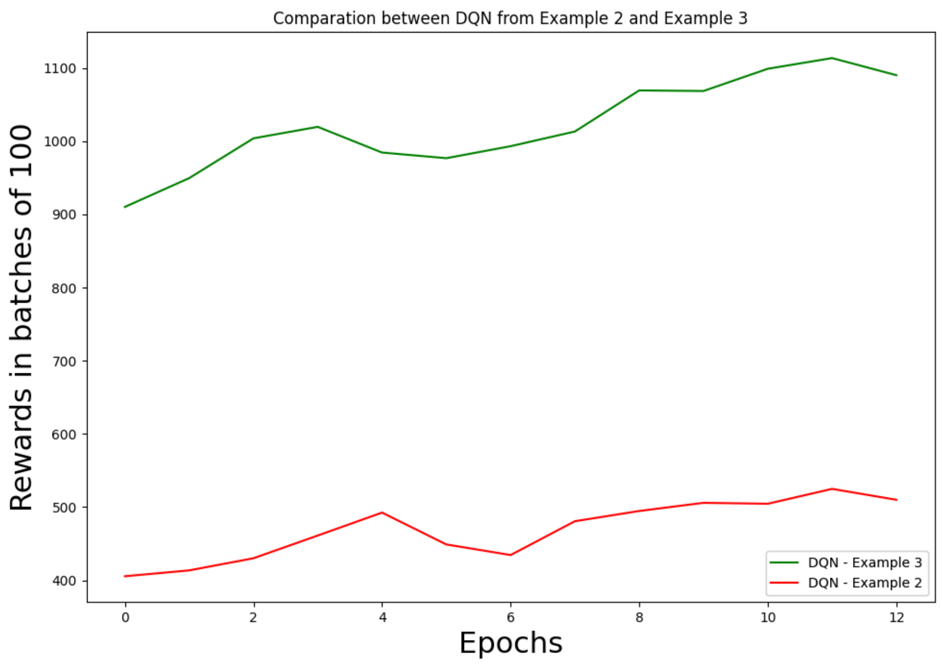


Figure 5.4: Comparison between Example 2 and Example 3 - DQN agent

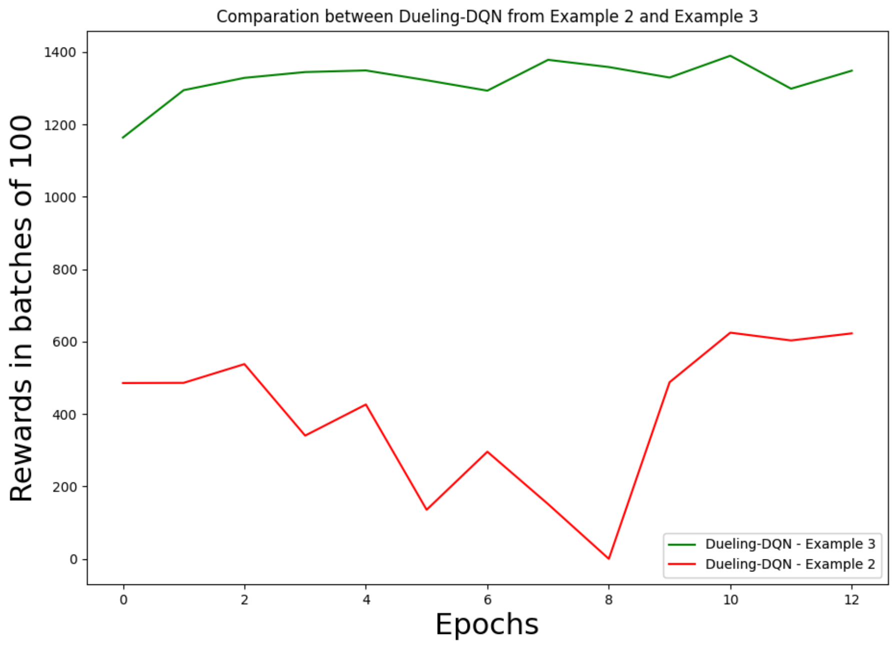


Figure 5.5: Comparison between Example 2 and Example 3 - Dueling-DQN agent

6 Conclusion

The need to connect devices all over the world and to increase the QoS and communication speed between them led to the evolution of 5G Networks and Beyond. New technologies always bring new challenges and 5G Networks are no exception.

With the emerging number of connected devices with different software, hardware, and interfaces. The complexity to organize, orchestrate and handle with such diversity increased, and solutions based on heuristic optimization techniques can't handle all the problems that 5G brings to the table.

That's why DRL solutions are being introduced in the networks research area. The capability to solve extremely complex problems basing the solution on the agent's experience makes these solutions more simple to achieve than the heuristic optimization techniques.

There are software solutions to simulate and emulate networks and create DRL agents, but there is no software solution that facilitates the creation of a network environment able to communicate with a DRL agent.

The project developed during the thesis aims to reduce the difficulty of setting up a network environment to train and compare DRL algorithms to solve network problems.

To reach the final solution, first, there was a research about the state of the art, i.e., the concepts and fundamentals of the 5G Networks, the 5G CN, the RAN, the SDN Controller, the ML, the DRL algorithms and also a section dedicated to understand what are the tools that are being used in this area of research and the importance of creating a software solution to help future researches.

Then, it is described what were the tools used to build the thesis solution, i.e., Containernet, Ryu Controller, and OpenAI gym, and how the communication between them was made. It was also explained how to set up the solution in order to test the wanted scenario.

After explaining the solution developed, three examples are presented to demonstrate the true value of the thesis. The two first examples focus on different problem related to routing and network slice admission control. The third example is the merge between the previous ones, which led to the development of DRL agents with the capability of choosing two actions instead of one. Each example was presented with its building blocks.

The last chapter is focused on showing the agents' training results from each example and comparing the second and third examples, reaching the conclusion that the DRL solution developed in the third example is better than the second one, which could not be possible without the thesis developed solution.

The thesis fulfilled the main goal of creating a tool to facilitate the fast development of scenarios for networks research field. Now on, the students and researches that want to explore, search and compare different DRL algorithms to solve network problems will have to spend much last time on creating the needed environment and move faster in the research process.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[2] Z. Xiong et al. "Deep Reinforcement Learning for Mobile 5G and beyond: Fundamentals, applications, and challenges". In: *IEEE Vehicular Technology Magazine* 14.2 (2019), pp. 44–52. DOI: 10.1109/mvt.2019.2903655 (cit. on p. 1).

[3] *What is NFV?* Accessed: 2022-01-30. URL: https://www.redhat.com/en/topics/virtualization/what-is-nfv (cit. on p. 1).

[4] X. YOU et al. "AI for 5G: Research directions and paradigms". In: *SCIENTIA SINICA Informationis* 48.12 (2018), pp. 1589–1602. DOI: 10.1360/n112018-00174 (cit. on p. 2).

[5] X. You et al. *AI for 5G: Research directions and paradigms - science china information sciences*. 2018-10. URL: https://link.springer.com/article/10.1007/s11432-018-9596-5 (cit. on p. 2).

[6] N. C. Luong et al. "Applications of deep reinforcement learning in Communications and Networking: A Survey". In: *IEEE Communications Surveys; Tutorials* 21.4 (2019), pp. 3133–3174. DOI: 10.1109/comst.2019.2916583 (cit. on pp. 2, 18–20, 22).

[7] M. Peuster, H. Karl, and S. van Rossem. "Medicine: Rapid prototyping of production-ready network services in multi-pop environments". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2016). DOI: 10.1109/nfv-sdn.2016.7919490 (cit. on pp. 3, 31).

[8] U. Trick. *5G an introduction to the 5th generation mobile networks*. De Gruyter Oldenbourg, 2021 (cit. on pp. 6, 8, 11).

[9] P. Marsch and P. Marsch. *5G system design: Architectural and functional considerations and Long Term Research*. Wiley, 2018 (cit. on pp. 6, 7).

[10] "5g nr: the next generation wireless access technology$_2$018". In: (2018). DOI: 10.1016/c2017-0-01347-2 (cit. on p. 7).

[11] K. Ibrahim and S. B. Sadkhan. "Radio Access Network Techniques Beyond 5G Network: A brief overview". In: *2021 International Conference on Advanced Computer Applications (ACA)* (2021). DOI: `10.1109/aca52198.2021.9626804` (cit. on p. 7).

[12] B. Bertenyi et al. "Ng Radio Access Network (NG-ran)". In: *Journal of ICT Standardization* 6.1 (2018), pp. 59–76. DOI: `10.13052/jicts2245-800x.614` (cit. on p. 9).

[13] S. Rommer et al. *5G Core Networks: Powering Digitalization*. Academic Press, an imprint of Elsevier, 2020 (cit. on p. 11).

[14] P. R. S. Kumar M. C. Trivedi and A. Punhani. "Evolution of software-defined networking foundations for IOT and 5G Mobile Networks". In: *Advances in Wireless Technologies and Telecommunication* (2021). DOI: `10.4018/978-1-7998-4685-7` (cit. on p. 13).

[15] C. N. Tadros, M. R. Rizk, and B. M. Mokhtar. "Software defined network-based management for Enhanced 5G network services". In: *IEEE Access* 8 (2020), pp. 53997–54008. DOI: `10.1109/access.2020.2980392` (cit. on p. 14).

[16] K. Chowdhary. *Fundamentals of Artificial Intelligence*. Springer, India, Private Ltd, 2020 (cit. on p. 16).

[17] S. J. Russell, P. Norvig, and E. Davis. *Artificial Intelligence: A modern approach*. Pearson Educación, 2022 (cit. on p. 16).

[18] V. François-Lavet et al. "An introduction to deep reinforcement learning". In: *Foundations and Trends® in Machine Learning* 11.3-4 (2018), pp. 219–354. DOI: `10.1561/2200000071` (cit. on p. 16).

[19] Y. Li. *Deep Reinforcement Learning: An Overview*. 2017. DOI: `10.48550/ARXIV.1701.07274`. URL: `https://arxiv.org/abs/1701.07274` (cit. on p. 16).

[20] M. L. Littman. "Reinforcement learning improves behaviour from evaluative feedback". In: *Nature* 521.7553 (2015), pp. 445–451. DOI: `10.1038/nature14540` (cit. on p. 17).

[21] R. S. Sutton. "Introduction: The Challenge of Reinforcement Learning". In: *Reinforcement Learning* (1992), pp. 1–3. DOI: `10.1007/978-1-4615-3618-5_1` (cit. on p. 17).

[22] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444. DOI: `10.1038/nature14539` (cit. on p. 19).

[23] X. Du et al. "Overview of deep learning". In: *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)* (2016). DOI: `10.1109/yac.2016.7804882` (cit. on p. 19).

[24] H. Dong, Z. Ding, and S. Zhang. *Deep reinforcement learning: Fundamentals, research and applications*. Springer Singapore, 2020 (cit. on pp. 19, 20, 23).

[25] *NS-3 Tutorial*. Accessed: 2022-01-30. URL: https://www.nsnam.org/docs/tutorial/ns-3-tutorial.pdf (cit. on p. 23).

[26] P. Gawłowicz and A. Zubow. "NS-3 meets Openai Gym". In: *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems - MSWIM '19* (2019). DOI: 10.1145/3345768.3355908 (cit. on p. 23).

[27] N. Jalodia, S. Henna, and A. Davy. "Deep reinforcement learning for topology-aware VNF resource prediction in NFV Environments". In: *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2019). DOI: 10.1109/nfv-sdn47374.2019.9040154 (cit. on p. 24).

[28] A. Dalgkitsis et al. "Dynamic resource aware VNF placement with deep reinforcement learning for 5G networks". In: *GLOBECOM 2020 - 2020 IEEE Global Communications Conference* (2020). DOI: 10.1109/globecom42002.2020.9322512 (cit. on p. 24).

[29] L. S. Sampaio et al. "Using NFV and reinforcement learning for anomalies detection and mitigation in SDN". In: *2018 IEEE Symposium on Computers and Communications (ISCC)* (2018). DOI: 10.1109/iscc.2018.8538614 (cit. on p. 25).

[30] Y. Liu et al. "Deep reinforcement learning based smart mitigation of ddos flooding in software-defined networks". In: *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)* (2018). DOI: 10.1109/camad.2018.8514971 (cit. on p. 25).

[31] P. Amaral and D. Simoes. "Deep reinforcement learning based routing in IP Media Broadcast Networks: Feasibility and performance". In: *IEEE Access* 10 (2022), pp. 62459–62470. DOI: 10.1109/access.2022.3182009 (cit. on p. 37).

[32] P. Oliveira. "Dynamic Network Slicing using Deep Reinforcement Learning". In: *NOVA School of Science and Technology, NOVA University Lisbon. Master in Electrical and Computer Engineering* (2022) (cit. on p. 42).

2022 AI Gym for Networks Pedro Capelo

NOVA SCHOOL OF SCIENCE & TECHNOLOGY