Faculdade de Engenharia da Universidade do Porto

Using Machine Learning to Improve Performance of Flying Networks

Baltasar de Vasconcelos Dias Aroso

Master's degree in Electrical and Computers Engineering

Supervisor: Rui Lopes Campos

Co-Supervisor: Eduardo Nuno Almeida

July 26, 2019

Resumo

A utilização acentuada de dispositivos móveis, as exigências de uma boa Qualidade de Serviço (QoS) e o aumento do número de *Temporary Crowded Events* nos dias de hoje remetem para a utilização de redes aéreas para fornecer conectividade à Internet nestes cenários. Estas redes aéreas são construídas por *Unmanned Aerial Vehicles (UAVs)* que atuam como pontos de acesso Wi-Fi ou estações de base celular.

A falta de informação que mapeie algoritmos de *Machine Learning (ML)* e o seu uso em problemas de otimização com redes aéreas, inclusive no desenvolvimento de algoritmos de posicionamento de UAVs, traz novas oportunidades de pesquisa nessa área.

Esta dissertação procura, deste modo, responder às necessidades dos utilizadores através do desenvolvimento de um algoritmo de posicionamento de UAVs, recorrendo a um modelo ML. Este modelo faz a previsão de qual a melhor topologia de rede aérea, de acordo com o comportamento dinâmico dos utilizadores.

Esta dissertação apresenta duas contribuições. A primeira refere-se a um levantamento sobre os conceitos básicos e fundamentais de algoritmos de *Deep Reinforcement Learning (DRL)*, assim como os principais métodos de aprendizagem usados na sua construção, remetendo para a sua aplicação às redes aéreas. A segunda contribuição é um modelo de ML envolvendo a exploração de técnicas de DRL com métodos *policy gradient* e redes neuronais. Inerente a este modelo, é proposto um algoritmo que visa definir a sua função de recompensa com base nas conexões entre os utilizadores e os agentes (UAVs), evitando a sobrecarga dos respetivos pontos de acesso.

O algoritmo proposto apresenta melhorias significativas relativamente às outras soluções analisadas, colmatando alguns problemas importantes e recorrentes que surgem na utilização de uma arquitetura de redes aéreas, tais como uma fraca Qualidade de Serviço na sobrecarga do sinal sem fios com muitas solicitações ao mesmo ponto de acesso.

Abstract

Nowadays, the increased use of mobile devices, along with the need to satisfy the users' Quality of Service (QoS) demands, particularly in Temporary Crowded Events (TCE), motivates the adoption of flying networks to provide Internet connectivity in these scenarios. Flying networks are built by Unmanned Aerial Vehicles (UAVs) which act as Wi-Fi Access Points (APs) or cellular Base Stations.

The lack of information that maps Machine Learning algorithms and their usage in optimization problems related to Flying networks, including in the design of UAV placement algorithms, brings up new research opportunities in this area.

This dissertation seeks to respond to the users' demands by developing an UAV placement algorithm through a Machine Learning (ML) model. This model allows the determination of the best airborne network topology according to the dynamic behavior of the users.

The contributions of this dissertation are two-fold. First, a survey on Deep Reinforcement Learning concepts and the core learning methods used when developing an DRL algorithm from scratch, in agreement with their application to the networking field. The second contribution is an ML model involving the exploration of DRL techniques with policy gradient methods and neural networks. Attached to this DRL algorithm, it is proposed an algorithm that aims to define the reward function of the DRL model based on the connections between users and agents, avoiding the overload of UAVs.

The proposed algorithm enables significant improvements with respect to other analyzed solutions, overcoming some important issues that come across when dealing with airborne networks, such as the weak QoS when overloading the wireless signal with too many request to the same access point.

Acknowledgments

I would like to thank my supervisors, Rui Lopes Campos and Eduardo Nuno Almeida, for all their support during the development of this dissertation. In addition, I would like to express my gratitude to all the members of the Wireless Networks (WiN) research group from the Center for Telecommunications and Multimedia (CTM) of INESC TEC, which have received me and shared valuable knowledge during this my last academic year.

I would also like to thank the organizing committee of "CTM Open Day 2019" for giving me the opportunity to showcase my research at that time, about Machine Learning and its application in Wireless Networks, to interested and general public.

Finally, I would like to thank my family and friends for their unconditional support during my entire academic path.

Baltasar Aroso

"I'd rather attempt to do something great and fail,
than to attempt nothing and succeed"

Robert H. Schuller

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

A2C	synchronous Advantage Actor-Critic
A3C	Asynchronous Advantage Actor-Critic
ACER	Actor-Critic with Experience Replay
ACKTR	Actor-Critic using Kronecker-factored Trust Region
AP	Access Point
BS	Base Station
CDN	Content Delivery Network
CRF	Conditional Random Field
D4PG	Distributed Distributional Deep Deterministic Policy Gradient
DDPG	Deep Deterministic Policy Gradient
DL	Deep Learning
DP	Dynamic Programming
DPG	Deterministic Policy Gradient
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
ELU	Exponential linear Unit
FBMN	Flying Backhaul Mesh Network
FMAP	Flying Mesh Access Point
GMM	Gaussian Mixture Model
GPS	Global Positioning System
LOS	Line-Of-Sight
LP	Linear Programming
MADDPG	Multi-Agent Deep Deterministic Policy Gradient
MBS	Mobile Base Station
MC	Monte Carlo
MCST	Monte Carlo Tree Search
MDP	Markov Decision Process
ML	Machine Learning
MLMPGA	Multi-Layout Multi-subpopulation Genetic Algorithm
MLN	Machine Learning for Networking
MLP	Multi-Layer Perceptron
MU-MIMO	Multi-User Multiple Input Multiple Output
OFDMA	Orthogonal Frequency-Division Multiple Access
PI	Policy Iteration
POMDP	Partially Observable MDP
PPO	Proximal Policy Optimization
QoE	Quality of Experience
QoS	Quality of Service
ReLU	Rectified Linear Unit

RL	Reinforcement Learning
SAC	Soft Actor-Critic
SNIR	Signal-to-Noise-plus-Interference Ratio
SON	Self-Organizing Network
SL	Supervised Learning
RL	Reinforcement Learning
T3D	Twin Delayed Deep Deterministic
TCE	Temporary Crowded Event
TD	Temporal-Difference
TMFN	Traffic-Aware Multi-Tier Flying Network
TRPO	Trust Region Policy Optimization
UAV	Unmanned Aerial Vehicle
UL	Unsupervised Learning
VI	Value Iteration
WEM	Weighted Expectation Maximization

Chapter 1

Introduction

1.1 Context

The world is increasingly dependent on communications. Although, in the past, the mobile phone was considered a simple device for calls, today it has numerous features that make users increasingly dependent on it. The enhanced potential of these smartphones is accompanied by the development of their hardware resources in order to respond to the users' high demands with regards to software. Nowadays, these electronic devices are constantly receiving emails, downloading applications, exchanging messages on social networks and synching to cloud-based storage. With this constant resort to downloads and uploads, greater bandwidth utilization is inevitable and with this, Quality of Service (QoS) problems arise.

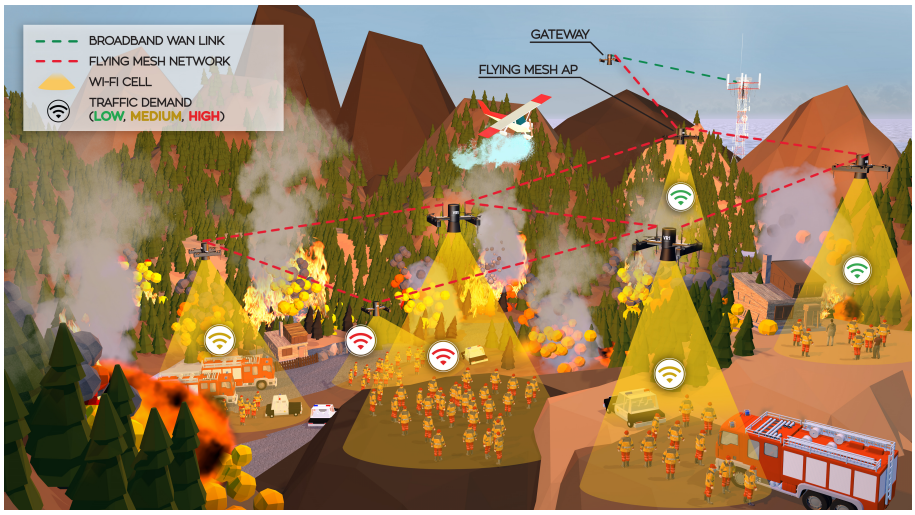
This fact alone shows the incessant growth for bandwidth capacity requirements and the Internet of Things topic has not even been touched. The intense study of this subject already tells us that the future passes through the increase of devices connected to the network for the transmission and reception of data. More wireless network users demand more wireless Access Points (APs), this is, the number of APs should be proportional to the number of users. When this is not accomplished and more users connect to the same AP overloading the signal, or when users move farther away from the APs, throughput decreases.

Nowadays, communication can be a vital need in people's life. Involuntary events, that arise from natural causes or any other type of unforeseen circumstances, such as fires, catastrophes or terrorist attacks, are examples of situations that require a quick action and where the guarantee of a good communication performance have a fundamental impact in people's rescue.

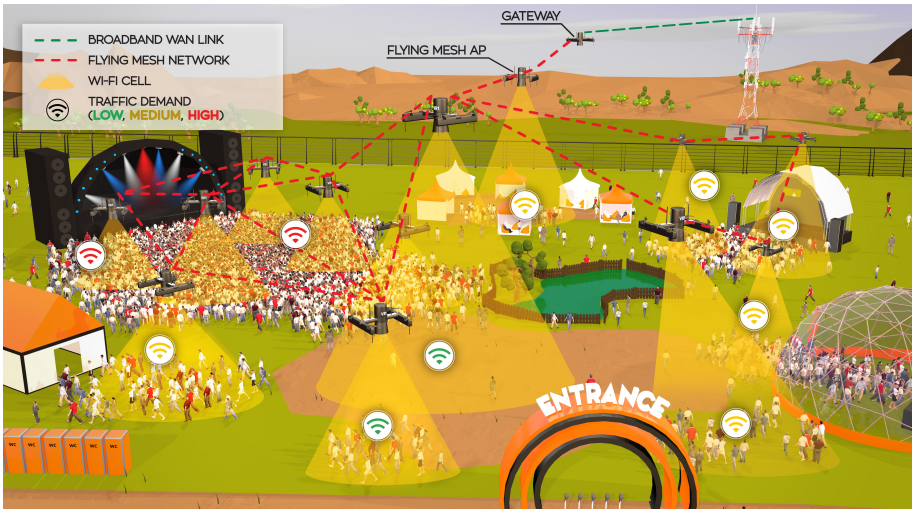
Today's social media are able to easily bring together a large number of people in a specific location for a certain time period with a particular purpose, such as concerts, festivals, manifestations and sports events, generally known as Temporary Crowded Events (TCEs). Unlike other types of events, where users' positions are fixed and known

in advance, TCEs are characterized by the variable movement of the users across the event area, creating high and variable traffic needs which are influenced by the dynamics of the event. Therefore, users' traffic demands vary over time and are defined by the combination of users' positions and offered traffic [12].

With scenarios that require this kind of flexibility, the answer focuses on the use of the aerial space to fulfill the user's requests and provide them the service they are demanding. Flying networks are, thus, the basis for building architectures that enable a fast, simple, easy and efficient deployment to respond to the users' demands, taking into account the limited terrestrial access to the crowd. Flying networks are composed of UAVs acting as Access Points or Base Stations as shown in Figure 1.1. Due to their mobility, UAVs can be positioned according to a three-dimensional reference, which allows them to be located exactly where they are needed.



(a) Emergency Scenario



(b) Music Festival Scenario

Figure 1.1: Two examples of flying networks application scenarios

1.2 Motivation

Machine Learning (ML) is increasingly associated with several areas by its transversality and uniqueness when applied to some specific scenarios. It has also been increasingly applied to the area of communication networks [13]. The traffic prediction and classification capabilities of machine learning have proven that it can be a powerful tool to solve current research problems in communications including optimizing network resources to improve QoS [13, 14].

Although UAV placement algorithms [12] demonstrate interesting results, these can still be improved, in particular, using ML techniques. Besides that, some improvements to this sort of algorithms can be done using ML techniques, enhancing the results and increasing the number of different scenarios where these algorithms can be applied.

Currently, many different approaches are being tested in the networking field, but most of them have some unwanted restrictions, such as static environments and different application scenarios. To the best of our knowledge, there are few studies that focus mainly on improving the QoS in accordance with the management of resources.

Reinforcement Learning has recently been very much used in the networking field as a useful tool to manage network resources [15]. Therefore, and considering some other studies that confirm those recent researches connecting ML techniques and the networking field [13, 14, 16–21], the exploration of Machine Learning to solve the problem of adapting the airborne network topology to the users' needs is analyzed as a very promising solution, pointing to the potential of this dissertation.

1.3 Problem Statement

This dissertation will assume, as its starting and main conceptual definition, a TCE scenario with high densities of moving users located in small and known geographic areas for a limited time period and the usage of UAVs carrying Wi-Fi APs to serve large and variable traffic demands.

The issues caused by the users' variable and unpredictable movements are directly related to the QoS that is provided to them. As mentioned above, to fulfill the requirements of a well defined and preserved broadband Internet connectivity, there is the need to automatically manage the airborne network topology in order to adapt them to dynamic scenarios. This dissertation addresses the problem of finding an accurate technique that determines the best airborne network topology. This topology will be instantly defined according to the users' positions answering to their needs by improving QoS.

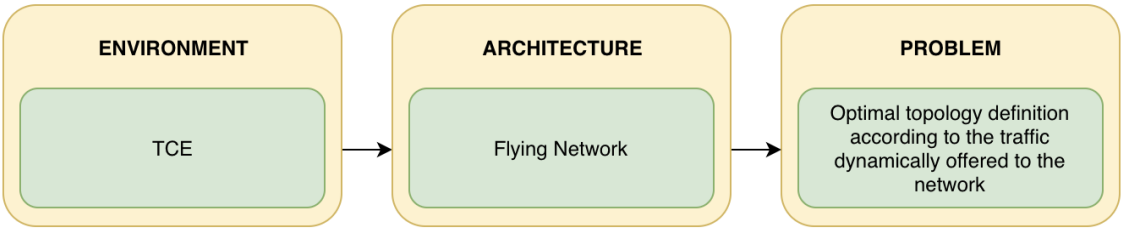


Figure 1.2: Problem Statement Diagram

1.4 Objectives

Considering the problem formulated in Section 1.3, the objective of this dissertation is to explore which is the best ML technique and use it in the development of an ML-based UAV placement algorithm that improves the QoS provided to the users in TCEs[1].

Thus, after this study, an ML model will be developed and trained to be able to determine the best airborne network topology which maximizes the QoS provided to the users.

1.5 Contributions

The main contributions of this dissertation include:

- A survey on the main concepts of Deep Reinforcement Learning (DRL), mapping them in the networking field by its implementation to solve a specific problem. It presents a more detailed description on how and why these kind of algorithms have been developed, giving to the reader the ability to deeply analyze them (Chapters 3, 4 and 5).

  This is distinguished from previous surveys that only express a high-level of ML concepts pertaining to its applications in the networking field ([2,13,14,17]), others which only focus on the main concepts of Deep Reinforcement Learning ([22,23]), and a few explaining how Multi-Agent algorithms work not even mentioning the benefits of Deep Learning in these flying network scenarios ([24,25]).

- Development of a DRL algorithm based on policy gradient methods and neural networks in order to increase the QoS provided to the users in TCEs, by managing the airborne network topology according to the user positions (Chapters 6 and 7).

[1] Throughout this document, it will be considered that, whenever the UAV is mentioned, it is presumed that it bears a Wi-Fi AP.

1.6 Document Structure

This document is structured in the following chapters:

- Chapter 2 [State of the Art] - Exploration of Machine Learning concepts and their advantages over current approaches to solve the problem stated, including a review of existing solutions;
- Chapter 3 [Reinforcement Learning Concepts] - Presentation of the most important concepts of Reinforcement Learning (RL) algorithms followed by the specification of the notation used throughout this document;
- Chapter 4 [Core Learning Solutions] - Enumeration and explanation of the existing core learning methods used to achieve an optimal solution concerning finite Markov Decision Process (MDP) problems;
- Chapter 5 [Policy Optimization] - Exploration of existing policy gradient methods useful to build the presented algorithm;
- Chapter 6 [Proposed Solution] - Presentation of the proposed solution developed to achieve this dissertation's objectives, detailing the basis of the DRL algorithm and its implementation;
- Chapter 7 [Results] - Demonstration of the results obtained with the proposed solution, evaluating and comparing them with other approaches;
- Chapter 8 [Conclusion] - Conclusion of this dissertation, discussing the main outcomes obtained, regarding the work developed, results accomplished and related future work.

Chapter 2

State of the Art

2.1 Concept of Flying networks

The concept of flying networks was born from the need to serve users of a TCE with the necessary means of communication to meet their data transmission rate requirements.

Some studies conclude that the high mobility of mobile terminals greatly affects data transmission rates, thus reducing the QoS by decreasing throughput and increasing jitter [26]. To respond to these needs, flying networks have been emerging. Flying networks are composed of UAVs acting as flying APs or cellular Base Stations, providing greater network flexibility through the mobility of access points. The movement of the mobile terminals and the analysis of user requirements influence the topology formed by these UAVs.

There are several applications for the flying networks, such as search and rescue, event monitoring, disaster response and delivery of goods [26].

There are different algorithms to solve optimization problems, such as finding the best airborne network topology of a flying network, which can be divided into deterministic and non-deterministic algorithms [27]. Inside of these two classes there are multiple optimization algorithms, such as Genetic Algorithms [28], Monte Carlos Search Method [29], Expectation–Maximization Algorithm [19, 30], Greedy Algorithm [31] and Gradient Descent Algorithms [32]. These and more other algorithms can be organized into sub-categories that will be mentioned throughout this document, such as Meta-heuristics, Linear Programming, Dynamic Programming, Monte Carlo Algorithms and Policy Gradient Methods.

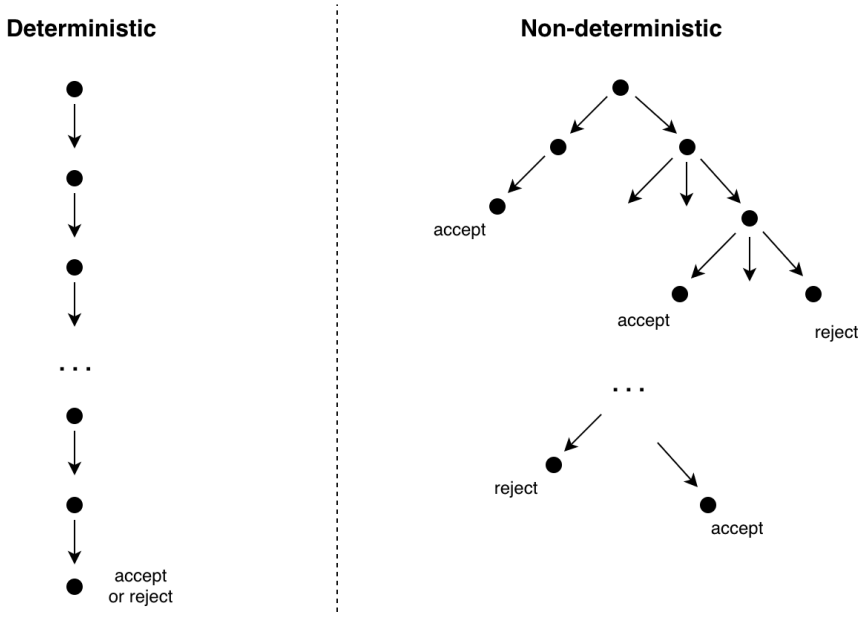


Figure 2.1: Differences between Deterministic and Non-deterministic algorithms

2.1.1 Deterministic Algorithms

A deterministic algorithm corresponds to an algorithm which, from given data entry, will generate the same output. In this sort of algorithms there is a dependent relation between the input and the output, where the data processing always passes through the same sequence of states.

The following algorithm is governed by equations and fixed constants along its iterations, following the concept of a deterministic algorithm, which can restrict its flexibility in adapting to dynamic environments.

Network Planning Algorithm

Considering the Traffic-Aware Multi-Tier Flying Network (TMFN) approach, the Network Planning Algorithm [12] was presented as one algorithm that characterized the users' traffic demands as the average throughput offered by the users and changes the position and range of the Wi-Fi cells of the Flying Mesh Access Points (FMAPs) according with those input data.

The TMFN is organized according to a two-tier architecture, a first tier that corresponds to the access network and is composed of the FMAPs, and a second tier that is associated to the backhaul network where the UAV Gateway is implemented.

Therefore, this architecture creates a mobile network that allows a physically reconfigurable adjustment of its Flying Mesh Access Points according to the user needs and following a metric formed by equations defined *a priori* and potential field generators.

The metric used in this model, a QoS representation, can be degraded by the saturation of the wireless channel and the interference created by the concentration of FMAPs on a small area. However, there are some suggestions to reduce the constraints referred above, like the use of available Wi-Fi channels in the 5GHz band for the FMAPs and the use of the IEEE 802.11ad standard, which enables the appearance of links in the 60 GHz band and the establishment of smaller cells [12].

For future work were pointed out three most relevant changes, mainly focused on the TMFN's access tier: improve the characterization of the users' traffic demands; calibrate the equations and its constants, so they seamlessly adapt to the dynamic scenarios; and evaluate the Network Planning algorithm in more dynamic scenarios.

2.1.2 Non-Deterministic Algorithms

Non-deterministic algorithms are algorithms of probabilistic nature which, for the same input, have different behaviours in different executions, obtaining different results/outputs and thus making a contrast to the deterministic algorithms.

This kind of algorithms are commonly composed by meta-heuristic, a high-level procedure to find/generate or select a heuristic - partial search algorithm - that may provide a sufficiently good solution to an optimization problem, especially with meagre/faulty information or limited computation capacity. This kind of approaches does not guarantee that a globally optimal solution can be found in some group of problems. The non-deterministic factor of this type of procedure is due to the stochastic optimization that is used in many of its implementations, so that the solution found is dependent on a set of randomly generated variables.

MLMPGA Algorithm

Multi-Layout Multi-subpopulation Genetic Algorithm (MLMPGA) is an application to solve multi-objective coverage problems of UAV networks. This multi-objective deployment is based on a weighted fitness function that takes into account coverage, fault-tolerance and redundancy as relevant factors to optimally place the UAVs. The proposed MLMPGA achieves significantly better performance results than the other meta-heuristic algorithms, such as particle swarm optimization, hill climbing algorithm, or classical genetic algorithms, in the vast majority of the simulation scenarios explored in [28].

As future work, this study refers the need to adapt and evaluate the proposed weighted fitness function and MLMPGAs in scenarios that follow more dynamic distributions of nodes, such as random ones, and the idea to model the weighted fitness function as a multi-objective problem based on Pareto Dominance in order to get the Pareto front solution for the three coverage problems announced.

2.2 Machine Learning Concepts

Machine learning is a technique of data analysis that strives to increase performance on a specific task, using statistical methods progressively and with as minimum human intervention as possible. This kind of algorithms seeks to automate and optimize the analysis of input data, by identifying patterns and making autonomous decisions, using a feedback network and learning from it. The output is made out of predictions of a software model that was not programmed explicitly. This software runs an algorithm that adaptively improves its performance by learning from previous samples, as the humans learn from experience.

2.2.1 Machine Learning vs Traditional Programming

Machine Learning was not created as a substitute for traditional programming, but as a complement to it. So, ML is used when traditional programming methods cannot deal efficiently with the problem.

To solve a problem using traditional programming, an engineer initially devises the algorithm and writes the corresponding code; then, it adds the input parameters and the implemented algorithm produces the desired results.

Figure 2.2: Traditional Programming

When applying ML to an identical problem, the method is completely different. Initially, the data scientist collects and prepares the data, then, it uses this dataset to train and test multiple ML models in order to find the one which presents the best results. Input parameters will be inserted into the ML model and the results will, finally, be produced.

One of the biggest differences between these two approaches is that in ML the model is not built by an engineer. Instead, the model is mostly handled by the data scientist with small tweaks to the ML algorithm settings. Besides that, while in traditional programming the input parameters are restricted by the human capacity to build an algorithm which would reasonably use all of those parameters, in ML the only limitations are the Central Processing Unit (CPU) and memory.

The Figure 2.2 and Figure 2.3 illustrate the differences between Traditional Programming and Machine Learning, more specifically a Supervised Learning Approach (which

is explained in section 2.2.2). Basically, while the first gets the output results using only the input parameters and the program devised by the engineer, the second is based on a mathematical model that is trained by the data from a dataset, building a program that can be used in the traditional programming.

Figure 2.3: Supervised Learning approach

2.2.2 Types of ML algorithms

ML algorithms are usually classified in three main categories:

- **Supervised Learning (SL)** - learn from labelled data, in order to conduct classification or regression tasks.

  SL is based on the idea of providing a set of input parameters followed by the expected results, in order to "teach" the ML algorithm with correct answers.

- **Unsupervised Learning (UL)** - learn from unlabelled data, clustering them and/or doing associations to learn more about them.

  UL uses untagged data, so, the algorithm should figure something out without being trained knowing some labels like in SL, trying to find hidden insights in raw data.

  Figure 2.4 shows how data is managed in the two previous learning methods and in the derived method from both, the Semi-Supervised Learning.

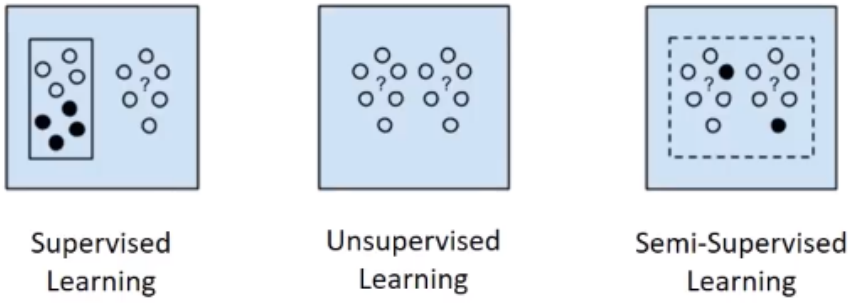


Figure 2.4: ML Algorithms according to their data management [1]

- **Reinforcement Learning (RL)** - learn by interacting with the environment, so as to find the best action series to maximize the accumulated reward function (objective).

  RL algorithms work similarly to the unsupervised ones, where a system must learn the expected output on its own. However, on top of that, a reward function is applied [2]. They are known for their strong capability to deal with decision-making problems (MDPs) and for being highly suitable for problems that have many undetermined parameters that change according to the environment state.

Figure 2.5 summarizes how each ML method is defined, relating their approaches with simple math functions.

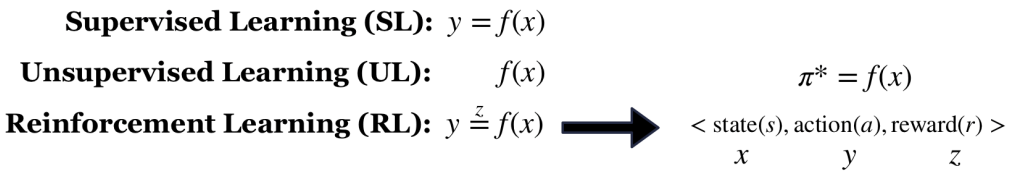


Figure 2.5: Relation between ML types of Algorithms and math functions

Figure 2.6 presents some of the ML techniques described in [2] and used in Self-Organising Networks (SONs), that are defined as adaptive and autonomous networks that are also scalable, stable and agile enough to maintain its desired objectives. Hence, these networks are not only able to independently decide when and how certain actions will be triggered, based on their continuous interaction with the environment, but are also able to learn and improve their performance based on previous actions taken by the system [33].

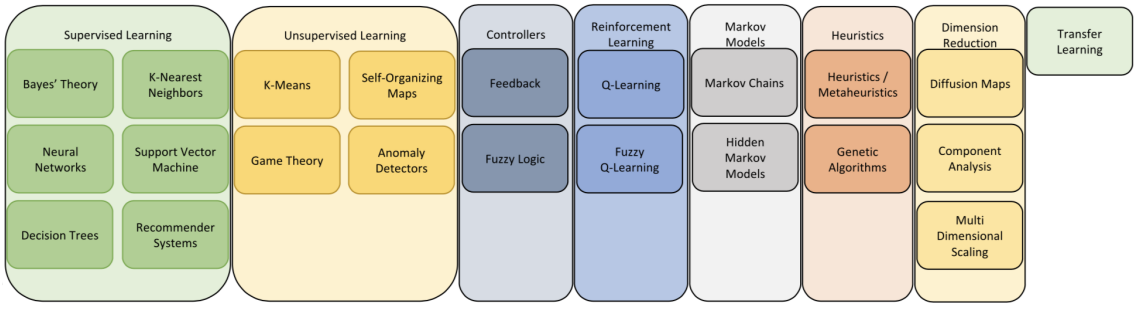


Figure 2.6: Different ML techniques described in [2] and used in SONs

As described in the Figure 2.7 there are numerous applications for the different types of ML algorithms.

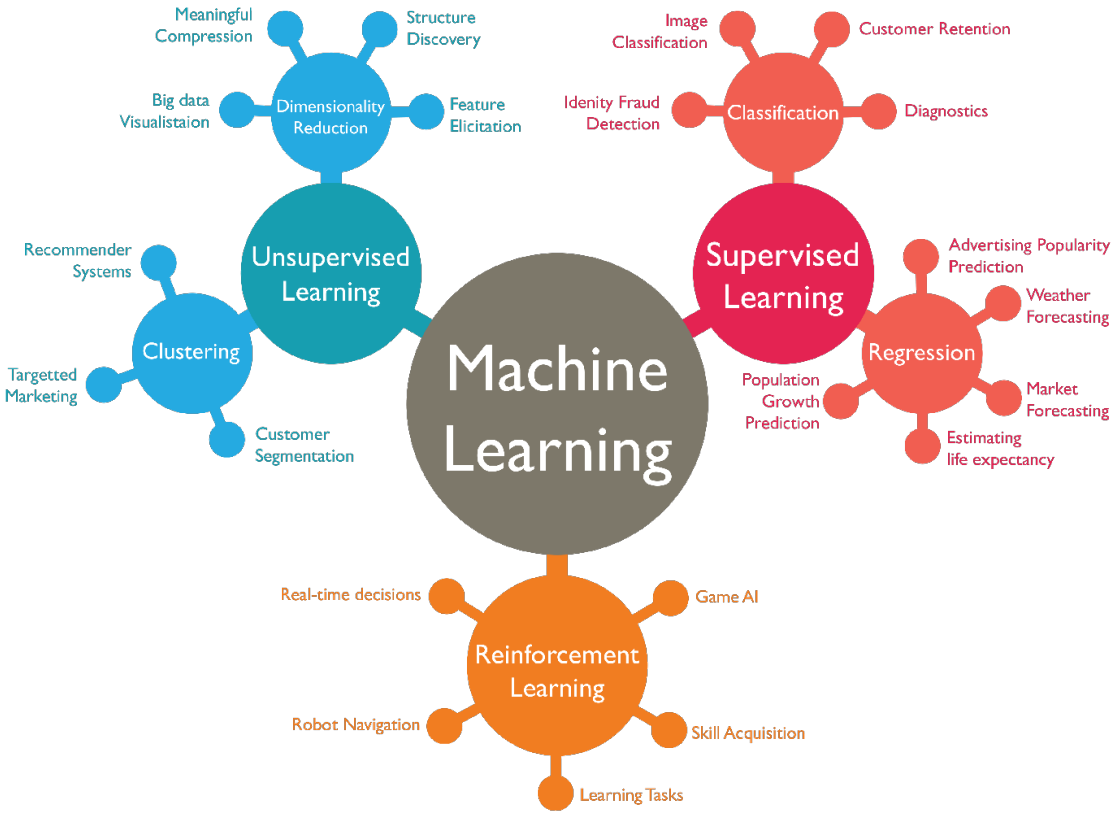


Figure 2.7: Types of ML algorithm applications [3]

2.2.3 Key Elements of Learning Algorithm

To build a Learning Algorithm, there are some key elements in Machine Learning that are indispensable [11], such as:

- **Representation**: Select a model to represent the data. A model must be represented in some formal language that the computer can handle.
- **Evaluation**: Choose an objective function to optimize different given values for the model parameters. An objective function is essential to distinguish good models from bad ones.
- **Optimization**: Estimate the model parameters using a certain optimization algorithm. Finally, the selected optimization method is used to search the highest-scoring model. The choice of an optimization technique is the key to the efficiency of the learner, and also helps to determine the model produced in case the evaluation function has more than one optimum.

Table 2.1 shows some examples of components of Learning Algorithms divided in the three sections mentioned above.

Representation	**Evaluation**	**Optimization**
Instances:	Accuracy / Error Rate	Combinatorial Optimization:
K-nearest Neighbour	Precision and Recall	Greedy search
Support Vector Machines	Squared Error	Beam search
Hyperplanes:	Likelihood	Branch-and-bound
Naive Bayes	Posterior Probability	Continuous Optimization
Logic Regression	Information Gain	Unconstrained
Decision Trees	K-L Divergence	Gradient Descent
Sets of Rules	Cost/Utility	Conjugate Gradient
Propositional Rules	Margin	Quasi-Newton Methods
Logic Programs		Constrained
Neural Networks		Linear Programming
Graphical Models		Quadratic Programming
Bayesian Networks		
Conditional Random Fields		

Table 2.1: Examples of the three components of Learning Algorithm [11]

2.2.4 Phases of an ML algorithm implementation

After defining the three key elements of the learning algorithms, it is time to start the process of building an ML approach to get a solution for a defined problem. This process is divided into four stages:

- **Data Management**: Collect the training data, explore it to better understand its structure and meaning, cleanse it - turn it into a mathematical construct (Representation - Section 2.2.3), that ML models understand -, prepare it to be loaded into the programming environment, and split it into training, validation and testing subsets. If there is no data to collect, it is needed to create a defined dataset to train, validate and test the model. This task is one of the obstacles of ML as will be mentioned later (Section 2.3.5).

- **Train Model**: Choose a learning task (prediction, clustering, etc), add some features and select the appropriate algorithm to train the model. The training process involves initializing some random values, predicting the output with those values, then comparing it with the model's prediction, and then adjusting the values so that they match those predictions/increase the reward function. This training set is performed multiple times in order to achieve the best model.

- **Evaluate Model**: Evaluate the output of the model using different metrics, score them and compare the results. In this phase, some hyperparameters - initially random variables that affect the objective function - are adjusted until a "good" model is obtained. This step is associated with the Evaluation and Optimization components of Learning Algorithm referred in Section 2.2.3.

- **Deploy Model**: Apply the model to brand-new data and monitor the outcomes. The predictions about that never explored data are made and if necessary, the ML algorithm implementation process is restarted, learning from these new observations.

2.3 Machine Learning for Networking

ML is suitable and efficient to solve networking issues for the following reasons [13]:

- performance prediction and intrusion detection are some examples of the importance that classification and prediction can have in network problems;

- many network problems are related to the dynamic behaviour of the systems when they are interacting with the environment, and it is not easy to accurately build models to represent these complex systems behaviors, such as throughput characteristics and changing patterns of UAVs topologies;

- each scenario may have different characteristics (e.g., network states and traffic patterns) and there is often the need to solve their problems independently.

2.3.1 Basic Workflow for MLN

As detailed in [13], Machine Learning for Networking (MLN) has a baseline workflow that is usually followed in the application of ML in the network field, as presented in the Figure 2.8. It starts with the **Problem Formulation**, where there is the need to decide the type and the amount of data to collect as well as the learning model to select with regard to the problem. That target problem can be seen as a classification, regression, clustering or a decision-making problem. Then, it is time for the **Data Collection**, which is divided into:

- **Offline Phase** - collecting historical data in a static environment and with predefined labels; it is important for data analysis and model training;

- **Online Phase** - real-time (dynamic environments) performance details and networks state are often used as inputs or feedback signals for the learning model, being their labels liable to changes;

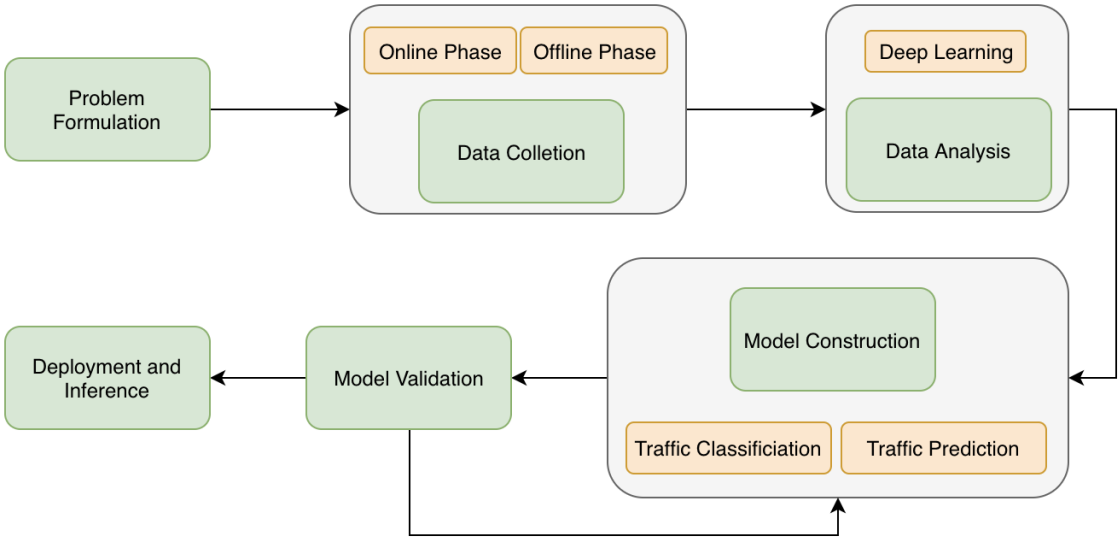


Figure 2.8: Basic Workflow for MLN

Thereafter, comes one of the most important steps, the **Data Analysis**, once it is the one in which the QoS metric should be found. Considering a learning algorithm, it is of extreme importance to find the proper features to fully unleash the potential of data. Normalization and discretisation are some concepts that are usually used in this step, in order to pre-process and clean raw data. **Deep Learning** appears, here, as an excellent approach to help automate the feature extraction (this concept will be more deeply explored in Section 2.3.2).

Once the data is prepared to be processed, the cycle between model construction and validation begins. While **Model Construction** involves model selection, training and tuning, **Model Validation** is an indispensable part of this basic workflow, deciding in which way the algorithm should go, deployment and inference phase or a new model construction, based on the error sources.

Concluding the workflow, there is the **Deployment and Inference** phase, which takes into account the limitations on computation or energy resources, analyzing the trade-off between accuracy and overhead for the performance.

2.3.2 Deep Learning

Concerning the diverse details that a real network system's data has, it is relevant to discuss the usage of Deep Learning.

Deep Learning (DL) has been seen as a promising solution for an efficient resource management and network adaption (*DeepRM* [1] [15] / *DeepRM2* [2] [34]) - the key to improve

[1] DeepRM is an example solution that reflects the problem of packaging tasks with several resource demands on a learning problem.

[2] DeepRM2 is an improvement of the DeepRM solution with faster convergence speed and better scheduling efficiency.

network system performance -, due to its ability to describe the inherent relationship between the inputs and outputs of network systems, having no human interference for that. According to [35], DL has proven to be a very useful approach for intelligent wireless network management owing to its human-brain-like pattern recognition capability. Besides that, this approach makes the network aware of the variations caused by changes in its topology and link conditions, which helps to generate more appropriate control parameters, leading to an optimal solution.

2.3.3 Feasibility Discussion

When talking about a complex ML model, there is always the **feasibility discussion**. Real-time systems with heavy computation are a problem that affects delay-sensitive network applications. The solution lies in a trade-off between information stateless and computation overhead, since the common method is to train the model with global information and incrementally update it with local information on a small timescale.

2.3.4 Dataset Generation Techniques

With regard to the several parameters that need to be collected to analyse the network performance, going to the field and obtaining those values manually is an impracticable option, once it is a hard working and time-consuming task. However, there are some available data sets that have collected those values for the training phase [36]. On the other hand, it is much hard-headed to use a framework that concedes the environment's simulation and the "real-time" obtainment of those network characteristics.

Therefore, the use of those frameworks provides more realistic simulations, introducing real data that replace the deterministic and probabilistic models traditionally studied and used, have demonstrated good results. **ns-3** [37] is an example of a discrete-event network simulator for Internet systems that enables the creation of a dataset using traditional network models (probabilistic and deterministic). However, [38] allows the extension of ns-3 to use real data through trace-based simulations, replicating real scenarios and conditions. It enables the evaluation of the progression of the system in response to realistic scenarios that can be tested with multiple simulations at the same time, exploring different variables under evaluation. The study and improvement of the Trace-based Simulation (TS) approach, *TraceBasedPropagationLossModel* [39], extends its application area to Multiple Access wireless scenarios, increasing the preference of simulator tests which, thus, permits the evaluation of more complex algorithms.

2.3.5 Obstacles of ML

There are some problems related to the application of ML techniques which impact MLNs in the same way.

The **lack of labelled data**, mainly for the SL algorithms, is a big restriction for the ML models. Nowadays, collecting a large amount of high-quality data that contain performance metrics and network profiles is one of the most critical issues of MLN. The main goal of the open database for the networking community is to acquire enough labelled data for different ML models in order to reduce costly and time-consuming data collection. It has been proved that, considering the ML domain, learning with a simulator rather than in a real environment is more effective and with lower trial cost in RL scenarios [13]. In order to substitute the high trial cost of large-scale network systems, which have a limited accessibility, simulators have to be reliable, scalable and fast.

Another problem of ML algorithms, considering RL models, is the **high system dynamics**. The difficulty in, accurately, finding a cost optimization function given the great dynamics of a system is a non-trivial problem.

Many implementations of ML models are related to big and complex problems that require the robustness of the machine learning algorithms. However, this can lead to another issue: the **high cost brought by leaning errors**. The need to train the ML algorithm with a different set of parameters requires as much as high computational resources and time as the complexity of the algorithm grows.

Finally, there is a special obstacle related to ML practical implementations, which is the fact that many learning models, essentially the deep learning ones, are still a **black box**. This forbids the analysis of the learning algorithms' output so as to find useful insights to understand how that output has been chosen, how does the network behave and how to design a high-performance algorithm. The solution for this type of problems is to promote the development of machine learning and the creation of algorithms that understand the decisions of deep learning algorithms, such as convolutional neural networks.

2.4 ML Algorithms to Increase Network Performance

Some studies presented algorithms that already explored the ML techniques in networks, considering the management of the airborne network topology, in order to increase the network performance.

[40] investigates the problem of UAV assignment in areas subject to high traffic demands given by the users. To solve this problem, **a neural-based cost function approach** is formulated. The proposed model chooses user demand patterns that are used to minimize the demand and cost functions by efficiently deploying aerial nodes as intermediate nodes between high demand areas and the Mobile Base Station (MBS). These demand

patterns are driven by a reverse multi-hierarchical neural model which is a combination of input, hidden, and output layer. These neural patterns are then topologically rearranged to form a stable network with a minimized cost function. It is shown that leveraging multiple UAVs not only provides long-range connectivity but also better load balancing and traffic offload.

[20] refers to the search for the best cell where the drone should be associated considering its location. The environment studied consists of a network of aerial drones connected to a standard cellular network on the ground, which is, virtually, divided into the cells where those drones should be mapped. It presents a machine learning method for **predicting a drone's best-serving cells at a given location**, based on the known signal observations in the nearby region, using the **Conditional Random Field (CRF)**[3] position. To conclude, as the analysis of this study is based on Signal-to-Noise-plus-Interference Ratio (SNIR) without explicit modelling the interference, further analyses are required for more complex deployment scenarios, and on integration with existing methods.

[19] presents a new machine learning framework that was proposed for enabling a predictive and efficient deployment of UAVs. The cellular traffic is analyzed and the network congestion is predicted using this ML framework that is based on a **Gaussian Mixture Model (GMM)** and a **Weighted Expectation Maximization (WEM)** algorithm, in order to guarantee a no-delay wireless service.

GMMs are a probabilistic model normally used to represent distributed subpopulations within an overall population. Mixture models, in general, do not require knowing which subpopulation a data point belongs to, allowing the model to learn them automatically, thus, developing a form of an unsupervised learning algorithm. GMMs are parameterized by two types of values: the mixture component weights and the component means and variances/covariances. Assuming that the number of components of the model is known, a technique commonly used to explore these model parameters is the Expectation Maximization, an iterative method used to search for maximum likelihood or maximum a posteriori estimates of parameters in statistical models. As explained in [19], it was used the WEM, that iteratively tries to find the best mixture component weights that provide the best model's output. Concluding, the algorithm presented in this paper has several appearances with RL algorithms, detailed in Section 2.2.2.

2.5 Conclusion

The state of the art is divided into two main concepts, flying networks and machine learning, since the focus of this dissertation is to find a solution that improves the QoS of a flying network by optimizing the distribution of UAVs in a TCE field. Therefore, it is

[3] CRF is a classical inference model widely used in computer vision and natural language processing, and it has been considered for the wireless system study.

necessary to verify the current and relevant algorithms that aim a similar goal without and with the use of machine learning, analyzing its results and limitations. In addition, it is also essential to understand the studied concepts that associate these two clusters, ML and flying networks, to figure out the best way forward to the proposed solution.

Two algorithms without the use of ML techniques are described in section 2.1: one, deterministic, and another, non-deterministic. Network Planning appears as an algorithm based on the use of attractive and repulsive forces to control the UAVs moves. MLMPGA is defined as a meta-heuristic procedure with better results than other meta-heuristic algorithms in the referenced article. These types of procedures are widely used in machine learning algorithms, given their statistical component and the use of stochastic processes to search for an optimal solution. So, and pointing to the future work of the presented papers, it can be concluded that the first one has limitations in adapting to more dynamic scenarios thanks to its deterministic component and the other one reveals an inconsistency in the accuracy of the results given its meta-heuristic nature, not guaranteeing a Pareto front solution, and not having been adapted or tested in more dynamic and random scenarios, which is necessary given the objective of this dissertation.

After describing the main concepts of Machine Learning and its useful components to build an ML model for a network, some algorithms were presented to determine the UAVs positions which improve the performance of the networks. From this study, the usage of an ML framework based on a Reinforcement Learning kind of approach demonstrated an outperforming result concerning the widest sort of scenarios, pointing out its impact on solving this sort of problems.

Chapter 3

Reinforcement Learning Concepts

3.1 Introduction

Reinforcement Learning is the act of learning by interacting with the environment, where the agent learns from consequences of its actions, rather than from being explicitly taught using an *a priori* dataset (supervised learning). The main concept of this type of ML algorithms consists of a *trial-and-error* learning approach in which its actions are selected according to its past experiences (exploitation) and also by new policies as well (exploration) [6].

The RL algorithms are characterized by an evaluative feedback that indicates how favourable the action taken was. There is a need for active exploration of the environment, for an explicit search for good behaviour as the evaluation occurs.

Figure 3.1 shows a diagram that represent the interactions between the Agent and the Environment. The Agent receives the *state* and the following response of the Environment, a *reward* or a *penalty* according to the objective function, and takes the *action* that maximizes the total reward.

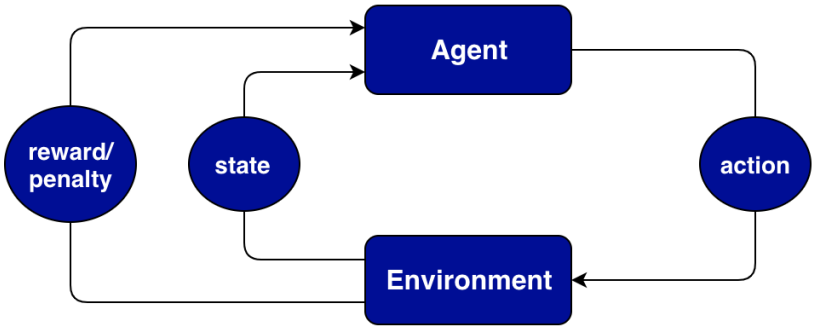


Figure 3.1: Reinforcement Learning algorithm

Throughout this document there will be detailed the most important concepts that are used to build an RL algorithm from scratch, however, some of them are antagonistic topics (namely the ones presented in Section 3.5), so not all will be used for the implementation,

but they are important in explaining how and why the algorithm applied to this particular network problem is composed in this way.

This type of machine learning algorithms respond to one of two possible case scenarios: bandits or MDPs, being the later a broad concept used to simplify the characterization of the problem. This chapter details the main concepts of RL applied to this dissertation's use case. It is important to be aware of the notation used throughout this document and the topics deeply explored in the following chapters.

Chapter 4 presents core learning methods to solve a finite MDP problem finding the optimal policy using the Bellman Equations presented in Section 3.4. Those methods are used to build different algorithms pointed to specific problems that rely on the assumption of concepts defined in this chapter (Section 3.5). Whereas in the model-based algorithms different learning methods are presented, in the model-free the learning methods, Monte Carlo and Temporal-difference, also mention some of the most used algorithms that implement them, MCTS and both SARSA and Q-Learning, respectively.

Chapter 5 refers to policy optimization's methods fully linked with the proposed solution and that combine the core learning methods previously mentioned (namely the model-free and value-based) with a policy-based approach (particularly the policy gradient methods).

Figure 3.2 represents the taxonomy of RL, referring which contents are presented in this three chapters (3, 4 and 5). Note that the concept of the division made in the model-based section is different from the concept considered in the model-free section - in the first one it is used the optimization technique for the division, while in the second one only the parameters of the RL problem are considered - in order to promote a better understanding of those topics with regard to the structure of this document.

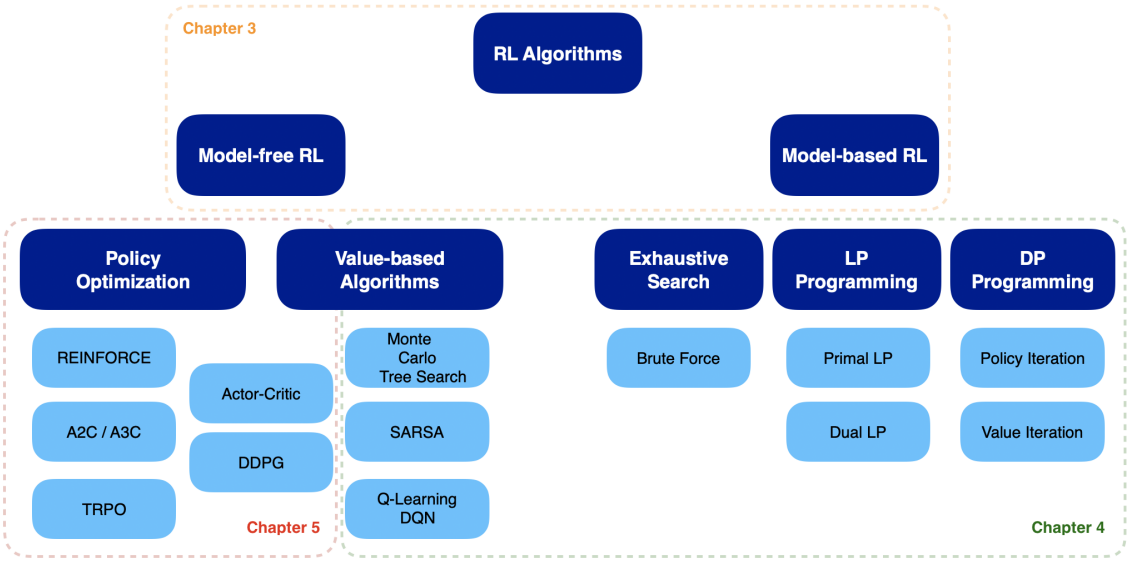


Figure 3.2: Taxonomy of RL

3.2 RL Terminology

As presented in Figure 3.1, there are crucial components in an RL algorithm: agent, environment, state, action and reward/penalty. However, to fully understand which and how methods are used to solve these type of RL problems, there are several concepts that need to be explained in a more practical way, beginning with the definition of the notation and meaning of the variables and functions used throughout this document.

It is assumed that all the variables and functions with a quote on top represent their next value, for example if s is the current state, then s' corresponds to the next state.

- **State**: $s \in \mathcal{S}$ The state represents the relevant characteristics of the environment, such as the positioning of UAVs and users, considering flying networks architectures.

- **Action**: $a \in \mathcal{A}(s)$ The action space - the set of all valid agent's actions in the given environment - can be part of a **discrete domain** - only several and discrete actions are available - or a **continuous domain** - the action is a value from continuous interval, having an infinite number of choices.

  Every action made in the environment takes into consideration its actual state. The actions are taken by the agent and confined by the action space previously mentioned.

- **Model**: $s_0, a_0, r_1, s_1, a_1, r_2, ..., s_{n-1}, a_{n-1}, r_n, s_n$ [1]

  The Model is the agent's representation of the environment. It predicts what the environment will do next [5]. It starts with an initial state that is changed to the next state after the initial action is executed and the first reward calculated. This occurs until the terminal state (s_n) is reached (episode tasks), unless the model follows a continuous task. These concepts will be better explored in Section 3.5.

- **State Transition Function**: $T(s,a,s') \sim P(s' \mid s,a)$

  The environment can follow one of the following models:

  $$\text{deterministic model} \rightarrow s' = f(s,a)$$
  $$\text{stochastic model} \rightarrow s' \sim P(s' \mid s,a)$$

  However, it is usually defined as a stochastic model, which brings it closer to reality and more responsive to dynamic scenarios.

  The State Transition Function is, by the very definition, definitely probabilistic, i.e., the transitions are defined by a probability function $P(s' \mid s,a)$ - described by a stochastic model. Considering the next state s' (which could be the same state) conditioned by the current state s, it will try to follow the action a.

[1] In this document, it is assumed that $s_0 = s$ and $s_1 = s'$

- **Reward/Penalty**: $r = R(s) = R(s,a) = R(s,a,s') \in \mathcal{R}$

  The R function defines the reward for taking the action a, in the state s and transitioning to state s'. The reward can be a positive value (reward) or negative (penalty). For simplicity, this function is called reward function.

  There are three ways of defining a reward function: only using the state, considering an inherent action to reach it ($r = R(s)$); using both the state and the action to be executed ($r = R(s,a)$); or adding a dependency for the state where the agent will transit ($r = R(s,a,s')$). However, these are only notation considerations, since all three possibilities will return the same value. Here, it will be considered a reward function that depends on the state and action to better understand that it will be calculated considering different actions executed in different states.

  The uncertainty of the stochastic environment forces the presence of a discounted future return, since the agent's goal is to maximize the cumulative reward it receives in the long run. Considering that R_t is the cumulative reward until time t, the cumulative discounted return equation can be defined as [6]:

$$\begin{aligned} G_t &= \sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1} \\ &= R_{t+1} + \gamma \cdot (R_{t+2} + \gamma \cdot R_{t+3} + \gamma^2 \cdot R_{t+4} + ...) \end{aligned} \tag{3.1}$$

  where γ is a hyperparameter between $[0,1)$, called discount factor. To simplify the equation it is assumed that $r = R_t \wedge r_1 = R_{t+1}$:

$$\begin{aligned} G &= \sum_{k=0}^{\infty} \gamma^k \cdot r_{k+1} \\ &= r_1 + \gamma \cdot r_2 + \gamma^2 \cdot r_3 + ... \end{aligned} \tag{3.2}$$

  The output of this function at successive time steps is related to each other in a way that is important to mention to better understand the theory behind this sort of RL algorithms:

$$\begin{aligned} G &= r_1 + \gamma \cdot r_2 + \gamma^2 \cdot r_3 + \gamma^3 \cdot r_4 + ... \\ &= r_1 + \gamma \cdot (r_2\ \gamma \cdot r_3 + \gamma^2 \cdot r_4 + ...) \\ &= r_1 + \gamma \cdot G' \end{aligned} \tag{3.3}$$

  The goal of the agent is to maximize the total amount of rewards it receives, i.e., maximize the expected value of the cumulative sum of a received scalar signal (reward) - Equation 3.2 - over a trajectory.

- **Policy**: $\pi(s) \rightarrow a$

The Policy is described as the agent's behaviour function. It defines the rules used by an agent to decide what actions should be taken. Ergo, it is a mapping from states to actions - takes a state and returns an action.

$\pi(a \mid s)$ might be used in a stochastic environment, where executing a specific action is not guaranteed, expressing the probability of executing action a in state s.

$$\text{deterministic policy} \rightarrow a = \pi(s)$$
$$\text{stochastic policy} \rightarrow \pi(a \mid s) = P(s' \mid s, a)$$

- **Optimal Policy**: $\pi^*(s)$

  If the action chosen by the policy is the best one from the entire action space - the one that maximizes the long term expected reward -, that policy is called optimal and defined by Equation 3.4 and 3.5.

$$\pi^*(a \mid s) = \arg\max_{\pi} Q^{\pi}(s, a) \tag{3.4}$$

$$\pi^*(s) = \arg\max_{a} \sum_{s'} P(s' \mid s, a) V^*(s') \tag{3.5}$$

  The state value function, $V(s)$, and the state-action value function, $Q(s,a)$ will be explained next.

- **Value Function**: $V^{\pi}(s)$

  The Value Function estimates how good for the agent is to be in a given state, the benefits of performing a given action in a given state. Hence, it maps a value to a state, telling what action to take while trying to maximize the cumulative discounted return function, as expressed by the following equation [6]:

$$V^{\pi}(s) = E_{\pi}\{G_t \mid s_t = s\} \tag{3.6}$$

  where E_{π} denotes the expected value of a random variable given that the agent follows policy π. G_t is the value of the cumulative discounted return function defined in Equation 3.1 at time step t, and s_t is the state at that same time t. The action a is given by the policy ($a = \pi(s)$) in a deterministic way, thus the value function does not need to take it in consideration to map the value to the action.

- **Q Function**: $Q^{\pi}(s,a)$

  The Q Function is a state-action value function. It defines an expected value when starting in a state s, performing an arbitrary action a while following a policy π, as expressed by Equation 3.7 [6]. This function is commonly used in RL algorithms to

deal with problems where a transition function is not available (model-free).

$$Q^{\pi}(s,a) = E_{\pi}\{G_t \mid s_t = s,\ a_t = a\} \tag{3.7}$$

where E_{π} denotes the expected value of a random variable given that the agent follows policy π. G_t is the value of the cumulative discounted reward function defined in Equation 3.1 at time step t, s_t is the state and a_t is the action, both at that same time t.

The relationship between Q^{π} and V^{π} is:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a \mid s) \cdot Q^{\pi}(a,s) \tag{3.8}$$

where the value of being in state s is the sum of every state-action value ($Q^{\pi}(s,a)$) multiplied by the probability to take that action ($\pi(a \mid s)$).

The following equations help to comprehend Figure 3.3 and Figure 3.4:

$$Q^{\pi}(s,a) = r_1 + \gamma \sum_{s'} P(s' \mid s,a) \sum_{a'} \pi(a' \mid s') Q^{\pi}(s',a') \tag{3.9}$$

$$V^{\pi}(s) = \sum_{a} \pi(a \mid s) \sum_{s'} P(s' \mid s,a) \Big[r_1 + \gamma V^{\pi}(s') \Big] \tag{3.10}$$

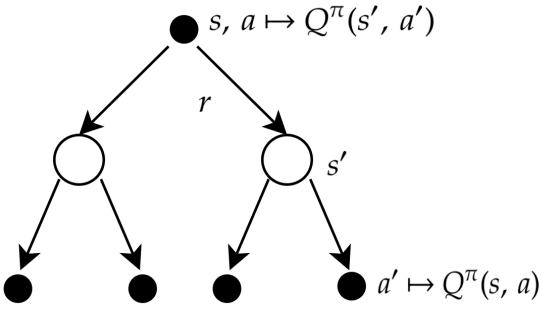


Figure 3.3: Q Function tree following the Equation 3.9

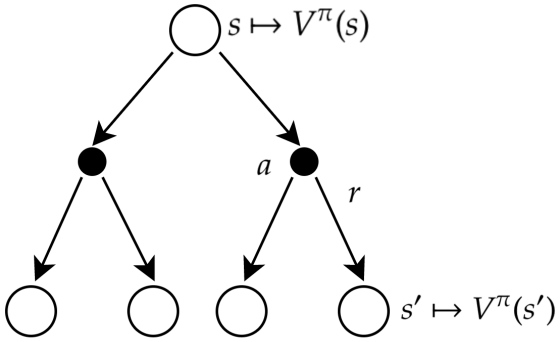


Figure 3.4: Value Function tree following the Equation 3.10

3.3 RL Problems

There are numerous applications of reinforcement learning [41], such as optimizing memory control, human-level video game play, mastering the game of go [42], personalized web services, game theory. These applications were born from well known classified problem. So, in order to use an RL algorithm to solve a specific problem, like the management of the airborne network topology in a TCE to improve QoS, it is essential to explore the types of RL problems and define which one is the best for this purpose.

3.3.1 Bandits

A bandit problem is defined as an evaluative feedback problem of lower complexity, because it does not involve learning to act in more than one situation. It has a *nonassociative* setting in the evaluative aspect of RL [6]. This concept is out of the scope of this dissertation, since the problem that this dissertation tries to solve is more complex and involves an *associative* characteristic that is not present here.

3.3.2 Markov Decision Processes

Finite MDPs are problems that involve evaluative feedback, as in bandits, and also an *associative* aspect, choosing different actions in different situations. They are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations/states through delayed reward [6].

MDPs models are fully determined by 5 quantities that completely characterize the environment ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$, T, θ):

- $\mathcal{S}$ is the set of states according to the world;
- $\mathcal{A}$ is the action space;

- $R: \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R} \subset \mathbb{R}$ is the set of possible reward values;

- $T: \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$ is the transition to state s' if action a is taken in state s. This transition probability only depends on the current state (s), which is called the Markov condition, thus the name of the process. This condition assumes that the current state (s) expresses all the meaningful information that should be considered to take the next action.

- θ corresponds to the specific parameters that differ in each RL algorithm, according to its settings. One example of these parameters is the value of the discount factor (γ).

In MDPs, what is initially tried to learn from the agent is its behaviour, the way of interacting with the environment that obtains high rewards. There are different types of behaviour [8]. The following example is presented to better understand them: imagine someone who asks the best way to go from point A to point B. Applying this example to each type of behaviour, they can be described as follows:

- **Plan**: a fixed sequence of actions. Considering this behaviour structure, the answer would be a fixed path, showing only one option in each crossroad.

- **Conditional plan**: includes conditional statements (like *if*'s). In this case, the answer would be a path constantly updated according to some random events occurred while the person moves from point A to B. For example, alternative paths would be calculated if that person found too much traffic at a specific street or region.

- **Stationary policy/universal plan**: mapping from states to actions. This behaviour structure is very extensive, because it explores all the possibilities with a policy. It has to learn a lot, being a very powerful option, but, at the same time, not easy to implement. For this behaviour structure the answer would be very complex, a set of possible paths according to all different states in which the person could end.

RL algorithms are characterized to use a stationary policy/universal plan, since, from the previous options, it is the one that best fits in a dynamic scenario where the solution is based on optimization methods.

The State Transition function defines the physics of the world, the rules that do not change from one iteration to another, i.e., the same rules are applied in every $s \in \mathcal{S}$. It follows a stationary policy.

3.4 Bellman Equations

Fully knowing the system - which includes a perfect knowledge of the state the agent is in as well as the transition probability and reward functions -, it is possible to estimate the value function for each policy π from a self-consistent equation. These equations are used by the model-based methods (Section 4.2) to solve finite MDP problems.

Starting with Equation 3.7, but using the Equation 3.2 to remove the time variable and make this deduction simpler:

$$\begin{aligned} Q^{\pi}(s,a) &= E_{\pi}\{G \mid s,\ a\} \\ &= E_{\pi}\{r_1 + \gamma G' \mid s,\ a\} \\ &= E_{\pi}\{r_1\} + \gamma E_{\pi}\{G' \mid s,\ a\} \end{aligned} \tag{3.11}$$

adding the transition probability of state s to state s' taking the action a - $T(s,a,s') = P(s' \mid s,a)$ - and the probability of taking an action a' in state s' - $\pi(a \mid s)$.

$$\begin{aligned} Q^{\pi}(s,a) &= r_1 + \gamma \sum_{s'} P(s' \mid s,a) \sum_{a'} \pi(a' \mid s') E_{\pi}\{G' \mid s',\ a'\} \\ &= r_1 + \gamma \sum_{s'} P(s' \mid s,a) \sum_{a'} \pi(a' \mid s') Q^{\pi}(s',a') \end{aligned} \tag{3.12}$$

and finally applying the Equation 3.8:

$$V^{\pi}(s) = r_1 + \gamma \sum_{s'} P(s' \mid s,a) V^{\pi}(s') \tag{3.13}$$

it is deducted the Bellman Equation for V^{π} (Equation 3.13), which expresses a recursive relationship between the value of the current state and the next.

The optimal Bellman equation is obtained using the optimal policy that is calculated using the Equation 3.5:

$$V^{*}(s) = r_1 + \max_{a} \gamma \sum_{s'} P(s' \mid s,a) V^{*}(s') \tag{3.14}$$

In order to better understand the concepts to be explored below, another derivative of Bellman Equation will also be considered (an equivalent to Equation 3.13):

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\{G \mid s\} \\ &= E_{\pi}\{r_1 + \gamma G' \mid s\} \\ &= \sum_{a} \pi(a \mid s) \sum_{s'} P(s' \mid s,a) \Big[r_1 + \gamma E_{\pi}\{G' \mid s'\} \Big] \\ &= \sum_{a} \pi(a \mid s) \sum_{s'} P(s' \mid s,a) \Big[r_1 + \gamma V^{\pi}(s') \Big] \end{aligned} \tag{3.15}$$

3.5 Important Concepts

RL problems are characterized by some of the concepts explained throughout this section, which differentiate them and defines the best approach to solve them.

As in supervised learning, where regression and classification tasks exist, in reinforcement learning it is possible to identify two types of tasks [5]:

- **Prediction**: this corresponds to the estimation of an expected total reward from any given state, assuming that $\pi(a \mid s)$ is provided, i.e., according to policy π the value function V^{π} is calculated with or without the model.

  An example of this task is the previously mentioned Policy Evaluation method.

- **Control**: this type of task tries to reach the policy $\pi(a \mid s)$ that maximizes the expected total reward from any given state, i.e., considering any kind of policy π, it finds the optimal policy π^{*}.

  An example of this task is the previous mentioned Policy Improvement method.

There are two different ways of learning resorting to a policy [4]:

- **On-policy**: learning on the job, which means evaluating or improving the policy that is currently used to make decisions.

- **Off-policy**: the evaluation is made using a target policy - may be deterministic (e.g. greedy) - while following a behaviour policy - may be stochastic - as shown in Figure 3.5. This is compared to the way we learn something while observing others doing the same thing.

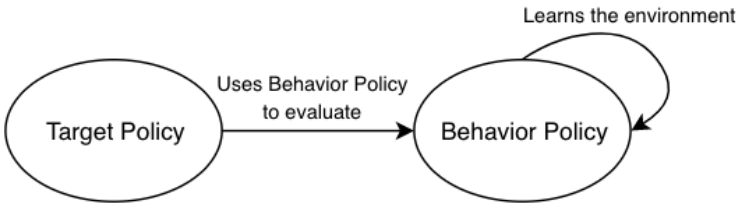


Figure 3.5: Off Policy life cycle

Off policy is seen as key to learn multi-step predictive models of a dynamic world. This method is more powerful and generic, despite its frequent high variance and slower convergence, since the data is due to a different policy. Whereas, on-policy methods are generally simpler and considered first.

When training an algorithm according to some RL problem, it is necessary to define which of these two is the nature of that problems' task [2] [6]:

[2] This task is not related to the prediction and control task, is another kind of task dependent to the problem definition.

- **Episode task**: consists of a task defined into a finite amount of time, an episode. For example, playing a chess game, where there are three final states: win, draw or lose.

  In this task, the reward is only calculated at the end of the episode or distributed evenly across all actions taken in that episode.

- **Continuous task**: is an endless task, where there is no terminal state. For example, the use case of this dissertation - managing an airborne network topology in order to increase the QoS, according to the users' positions in the TCE environment.

  This is a concrete case where the discount factor (γ) is usually applied in the expected reward function. The usage of this hyperparameter enables the control of how the cumulative reward is affected by future rewards, being able to determine some kind of terminal state.

These six concepts are mainly important for the Model-free algorithms that will be addressed in Section 4.3.

It is necessary to define a few more topics related to this problem definition, specifically, to detail the definition of the environment and which information can be extracted from it.

- **State** versus **Observation**:
  While a state has no hidden information, an observation is a partial description of a state, having omitted information.

- **Fully** versus **Partially Observed**:
  As the word itself suggests, a fully observed environment provides to the agent a complete observation of its state, while a partially observed environment only gives it a partial observation.

Making use of the above topics, there are two essential concepts that are indispensable to the problem definition and which will be further discussed in Chapter 5.

3.5.1 Partially Observable MDP

Partially Observable MDP (POMDP) is a discrete time stochastic control process that defines a way of talking about "non-Markov" environments. Being a partially observed environment, the agent only receives, at each time step, an observation of its environment which does not allow it to identify the state with certainty.

It is a 7-tuple $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \Omega, \mathcal{O}, \gamma)$ where [4]:

- Ω is a finite set of observations, ω_t,

- $\mathcal{O} : \mathcal{S} \times \Omega \rightarrow [0,1]$ is a function that returns the conditional probability of seeing an observation $\omega_t \in \Omega$ given a state $s_t \in \mathcal{S}$.

besides the already defined $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \gamma)$, which correspond to the 5-tuple of an MDP.

The agent chooses an action $a_t \in \mathcal{A}$ after receiving an observation $\omega_t \in \Omega$ that depends on the state $s_t \in \mathcal{S}$ with probability $\mathcal{O}(s_t, \omega_t)$. Then, the environment follows the same procedure as a simple MDP, it transits to the state $s_{t+1} \in \mathcal{S}$ with probability $T(s_t, a_t, s_{s+1})$ and delivers to the agent a reward $r_t = R(s_t, a_t, s_{t+1}) \in \mathcal{R}$.

3.5.2 Multi-agent Systems

A multi-agent system corresponds to a set of multiple interacting agents within an environment. Assuming a multi-agent POMDP with N agents, its tuple $(\mathcal{S}, \mathcal{A}_1, ..., \mathcal{A}_N, T, R_1, ..., R_N, \Omega, \mathcal{O}_1, ..., \mathcal{O}_N, \gamma)$ differs from the one presented in POMDPs since [4]:

- $\mathcal{S}$ is a finite set of states describing all the possible configurations of the N agents,
- $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2 \times ... \times \mathcal{A}_n$ is a finite set of actions for each agent $n \in [0, N]$,
- $R_n : \mathcal{S} \times \mathcal{A}_n \times \mathcal{S} \to \mathcal{R} \subset \mathbb{R}$, $\forall n$ is the reward function for each agent $n \in [0, N]$,
- $\mathcal{O}_n : \mathcal{S} \times \Omega \to [0, 1]$ is the set of conditional observation for each agent $n \in [0, N]$,

There are two main topics that cover the most important settings to be defined when applying this sort of systems [4]:

- **Collaborative** versus **Non-collaborative**:
  If all the agents work together in order to achieve the same goal, they are in a collaborative mode, where they have a shared reward measurement ($R_i = R_j$, $\forall i, j \in [1, ..., N]$). Whereas in a non-collaborative configuration, each agent receives their own reward. As in both cases, each agent i aims to maximize a discounted sum of rewards $\sum_{t=0}^{T} \gamma^t r_t^{(n)}$. A non-collaborative setting may lead to a competition scenario, where all agents seek to maximize their reward while sharing the same environment.

- **Centralized** versus **Decentralized**:
  In a decentralized configuration, each agent decides its own action conditioned only by its local information, while in a centralized configuration, the algorithm has access to all the observations (ω_n) and all the rewards (r_n).

  A collaborative environment, when combined with decentralization, might lead to the emergence of communication between agents in order to share information, whereas, while combined with a centralized setting, it can be reduced to a single-agent RL problem.

This type of systems are challenging since the agents update their policies individually as the learning progresses. Therefore, the environment looks non-stationary to any particular agent, since they do not know how the other agents will act and consequently,

how their actions will affect their rewards. This can be explained by the use of a partially observed environment and agents that follow intrinsic stochastic policies. For these reasons, it is observed a high variance of the expected global return, which makes the learning process harder, namely when used in conjunction with bootstrapping [4].

Chapter 4

Core Learning Solutions

To solve an MDP it is required to find the right action to be taken in a specific state, to find the optimal policy π^*. The learning process of RL problems is described by the Figure 4.1. There are five main approaches divided in two types of models as presented in Figure 3.2: Model-based algorithms - know the model that is use for planning value/policy functions - and Model-free Algorithms - do not know the model, need to interact. The first one is divided in Exhaustive Search, Linear Programming and Dynamic Programming, while the second one includes the Monte Carlo and Temporal-Difference Methods.

Model-based algorithms learn a model from experience and use it to plan the value function and/or the policy, while model-free algorithms have no model to use, so they learn the value function and/or the policy through experiences collected from agent's interactions with the environment.

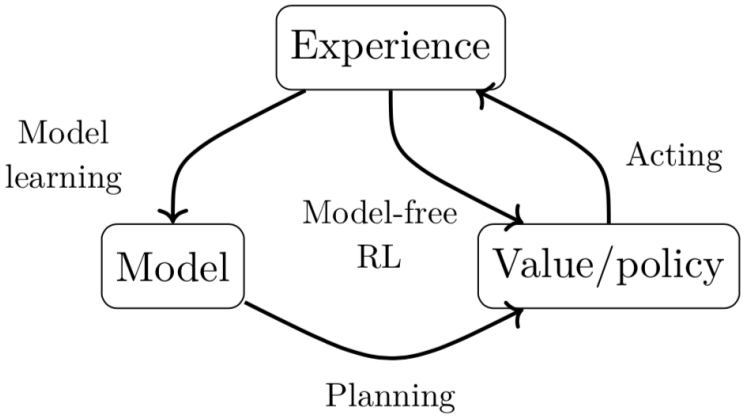


Figure 4.1: General schema representing the different approaches for RL [4]

Before diving into those approaches, it is important to take into consideration the transversal issues that might appear in solving an MDP problem and how to bypass them.

4.1 Learning Issues

When trying to solve an MDP problem, first it is necessary to learn that MDP problem and only then solve it. There are two possible approaches when learning a sequential decision-making task [4]:

- **Offline learning**: when only limited data of a given environment is available for the learning process.
  This learning mode generally needs a **batch setting** to define the amount of data used by the agent to learn without the possibility of interacting further with the environment.

- **Online learning**: when the learning process occurs while the agent is gradually gathering new experience from its interactions with the environment.
  In this configuration, the learning problem is more complex and, even without requiring a large amount of data (sample efficiency), learning is not influenced only by the algorithm's ability to generalize well from limited experience.

In order to solve this issue, the agent can gather experience using an exploration/exploitation strategy. Besides that, he can request a replay buffer to cache its experiences, so they could be processed at a later time.

4.1.1 Exploration/Exploitation Strategy

Exploration means trying random actions in different states, which helps discovering the underlying MDP, and **exploitation** means following the so-far optimal policy, which helps maximizing rewards.

After some exploration the agent might have found a set of apparently rewarding actions and it is time to exploit them, following a new so-far optimal policy. However, RL algorithms suffer from the **curse of dimensionality**, since it tries to cover all possibly relevant situations in a real-world scenario. This results in a large action and state spaces, which raises the problem of convergence, i.e., it takes an unfeasible amount of time to get all the possible action and state combinations to first learn the MDP problem.

Therefore, RL algorithms try to implement both of these concepts at the same time: learn the MDP (exploration), while solving it to find the optimal policy (exploitation). About the convergence problem, it is solved with the **generalisation** of the value function, usually applying a function approximation. Generalisation is a central concept in the field of reinforcement learning. This concept is related to the ability to achieve good performance in an environment where the collected data is limited, so it is directly related to the notion of sample efficiency, for example, when the state-action space is too large to be fully visited [4].

In addition, RL algorithms require smart exploration techniques. Randomly choosing actions without reference to an estimated probability distribution lead to poor performance. ϵ**-greedy** is an example of a solution for that problem, making the agent choose the action that follows the so-far optimal policy with a probability of $1-\epsilon$ and a random action with probability ϵ.

4.1.2 Experience Replay

A replay buffer ensures that the mini-batch updates are made using a reasonably stable data distribution stored in the replay memory, helping for convergence / stability. This approach is particularly well-suited when using an off-policy learning algorithm since experiences from past policies (different policies) do not introduce any bias, but contrariwise, are good for exploration [4].

A replay buffer allows processing the transitions in a different order than they are experienced. Furthermore, there is the possibility to use **prioritized replay**, that allows the control of the frequency with which the transitions are considered. This control can be made giving them different meaning, like ordering them according to some characteristic or value, choosing which experience to store and which one to replay. In [43], that meaning was related with the magnitude of the transitions' TD error, i.e., experiences with a bigger TD error value would have a bigger probability of being selected. These methods seek to the replay of the "unexpected" transitions more often in order to make the agent more flexible to dynamic scenarios.

4.2 Model-based Algorithms

Model-based Algorithms are, as the word itself indicates, algorithms that use their knowledge of the model to solve an MDP problem. Knowing the state transition function, $T(s,a,s')$, and the reward function, $R(s,a)$, it is possible to achieve the optimal value function and/or the optimal policy using the Bellman Optimality Equation. Value iteration and policy iteration are two methods that use this kind of algorithms and which have their origins in the field of Dynamic Programming (DP).

4.2.1 Exhaustive Search

This method consists in exploring all the possibilities - all the deterministic Markov policies -, evaluate each policy and return the best one. This is considered a naive approach since the number of policies is exponential, $|\mathcal{A}|^{|\mathcal{S}|}$ [1].

[1] $|\mathcal{S}|$ is the size of the set of spaces and $|\mathcal{A}|$ the size of the action space

4.2.2 Linear Programming

Linear Programming (LP) is an optimization framework, in which it is possible to have linear constrains in a linear objective function and get a solution in polynomial time, as long as the number of variables and constraints are polynomial. Thus, to use these methods it is mandatory to encode the MDP in a linear program. There are two ways of solving an MDP problem with a linear program [44, 45]:

- **Primal Linear Program (P-LP)**: This method tries to:

$$\min_{s} \sum_{s} V(s) \tag{4.1}$$

according to the following equation:

$$V(s) \geq R(s,a) + \sum_{s'} T(s,a,s')V(s'), \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{4.2}$$

being the non-linear function *max* substituted in the Bellman Equation - presented in Equation 3.13 where $r_1 = R(s,a,s')$ - by a *greater than* symbol, $\geq$.

Therefore, V^* is the solution of the above linear program. This program has $|\mathcal{S}||\mathcal{A}|$ constrains and $|\mathcal{S}|$ variables.

- **Dual Linear Program (D-PL)**: This method tries to:

$$\max_{\lambda} \sum_{s} \sum_{a} \lambda(s,a)R(s,a) \tag{4.3}$$

according to the following equation:

$$\sum_{a'} \lambda(s',a') = \mu(s) + \gamma \sum_{s} \sum_{a} \lambda(s,a)T(s,a,s'), \forall s,s' \in \mathcal{S}, \forall a,a' \in \mathcal{A}, \lambda(s,a) \geq 0 \tag{4.4}$$

where:

$$\lambda(s,a) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s,\ a_t = a) \tag{4.5}$$

and the optimal policy is defined as:

$$\pi^*(s) = \arg\max_{a} \lambda(s,a) \tag{4.6}$$

This program has $|\mathcal{S}|$ constrains and $|\mathcal{S}||\mathcal{A}|$ variables.

4.2.3 Dynamic Programming

Dynamic Programming is a mathematical and computer programming method of optimization developed by Richard Bellman. Its main purpose is to simplify large and complex problems by dividing them into simpler sub-problems in a recursive manner, following a *divide-and-conquer* approach. It is a solution to problems that contain the following properties: Optimal Structure - follow the *Principle of Optimality* and their optimal solutions can be decomposed into sub-problems -, and Overlapping Sub-problems - their sub-problems recur many times and the sub-problems' solutions can be stored and reused again.

MDPs satisfy both properties: the Bellman Equation gives recursive decomposition and the Value Function stores and reuses solutions. Consequently, and due to the complexity of the problem that this dissertation is trying to solve, this could be a useful approach.

4.2.3.1 Policy Iteration

Policy Iteration (PI) is the way of finding an optimal policy. As the Bellman equation shows (Equation 3.14), the purpose of recursion is to seek a consecutive policy improvement evaluating V^{π} over each iteration. Thus, a monotonic sequence is obtained by improving policies and value functions [6]:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi^* \xrightarrow{E} V^* \tag{4.7}$$

where $\xrightarrow{E}$ denotes a *policy evaluation* and $\xrightarrow{I}$ denotes a *policy improvement*.

As the monotonic term implies, each new policy used in the Bellman equation has to be a strict improvement over the previous one (unless it is already optimal).

Policy Evaluation is an iterative computation, which starts with the value function for the previous policy - if the model has a terminal state, the initial approximation, V^{π_0}, is defined with value 0 - and, consecutively, calculates a new return of the value function for an arbitrary policy, i.e., each successive approximation is obtained through the usage of Bellman Equation for V^{π} (Equation 3.14) as an update rule. Policy Evaluation is also referred as a prediction method that estimates V^{π}.

Policy Improvement is the process of creating a new policy from the improvement of the original one, by making it greedy with respect to the value function of the original policy. The new policy is chosen according to a *Domination* concept, which defines the *Policy Improvement Theorem* [6]:

Policy Improvement Theorem. *Let π and π' be any pair of deterministic policies such that, for all $s \in S$*

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \tag{4.8}$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$V^{\pi'}(s) \geq V^{\pi}(s) \tag{4.9}$$

Therefore, *Strict Domination* appears as [8]:

$$\begin{aligned} \pi_1 \geq \pi_2 \quad iff \quad & \forall s \in \mathcal{S} \quad V^{\pi_1}(s) \geq V^{\pi_2}(s) \\ \wedge \; & \exists s \in \mathcal{S} \quad V^{\pi_1}(s) > V^{\pi_2}(s) \end{aligned} \tag{4.10}$$

Policy improvement must only accept the exchange of the old policy for a strictly better policy, except when the original policy is already optimal.

These concepts are being applied considering deterministic policies. However, it is a natural extension to consider changes at all states and to all possible actions, selecting, at each state, the action that seems to be the best according to the state-action value function, $Q^{\pi}(s,a)$. In other words, it is considered the new greedy policy, π', calculated as:

$$\begin{aligned} \pi'(s) &= \arg\max_a \; Q^{\pi}(s,a) \\ &= \arg\max_a \; E_{\pi}\{r_1 + \gamma G' \mid s,\, a\} \\ &= \arg\max_a \; \sum_{s'} P(s' \mid s,a)\Big[r_1 + \gamma V^{\pi}(s')\Big] \end{aligned} \tag{4.11}$$

In particular, the Policy Improvement Theorem carries through as stated for the stochastic case. Besides that, if there are ties in the policy improvement steps, such as in Equation 4.11, where it is possible to have several actions at which the maximum is achieved, then, in the stochastic case, there is no need to select a single action, each of those actions will have a proportional probability of being selected in the new greedy policy.

Concluding, the greedy policy takes the action that looks best in the short term - after one step of lookahead - according to V^{π}. This is in agreement with the Policy Improvement Theorem, so this greedy policy is as good as, or better than, the original policy, following a monotonic sequence.

It is said that a policy is nearly optimal if the amount that it could improve according to the new calculated state-value, per iteration, is very small, i.e., less than a predefined hyperparameter ϵ. This assumption is given by [8]:

$$\pi \;\text{ is }\; \epsilon\text{−optimal} \quad iff \quad |V^{\pi}(s) - V^{\pi^*}(s)| \leq \epsilon, \quad \forall s \in \mathcal{S} \tag{4.12}$$

The following algorithm (1) explains how this method can be implemented.

Algorithm 1 - Policy Iteration

procedure INITIALIZATION:
  $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$, except that $V(terminal) = 0$
end procedure

procedure POLICY EVALUATION:
  repeat
    $\Delta \leftarrow 0$
    for all $s \in \mathcal{S}$ **do**
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_s P(s', r \mid s, \pi(s))\Big[r_1 + \gamma V(s')\Big]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
  until $\Delta < \epsilon$ ▷ the small positive number determining the accuracy of estimation
end procedure

procedure POLICY IMPROVEMENT:
  policy-stable $\leftarrow$ *true*
  for all $s \in \mathcal{S}$ **do**
    $a \leftarrow \pi(s)$
    for all $a \in \mathcal{A}$ **do**
      $Q^\pi(s,a) = \sum_{s'} P(s', r \mid s, a)\Big[r_1 + \gamma V^\pi(s')\Big]$
    end for
    $\pi(s) \leftarrow \arg\max_a Q^\pi(s,a)$
    if $a \neq \pi(s)$ **then**
      policy-stable $\leftarrow$ *false*
    end if
    if *policy-stable* **then**
      stop and return $V \approx v^*$ and $\pi \approx \pi^*$
    else
      go to 2
    end if
  end for
end procedure

4.2.3.2 Value Iteration

Value Iteration (VI) consists of an iterative application of the Bellman Optimality Equation backup in order to find the optimal policy π^*. VI uses, as in PI, a Bellman Equation as an update rule to calculate the $V(s)$, but in this case the optimal one, Equation 3.14. It focuses on finding the optimal state value function, V^*, and once that is found, it extracts the optimal policy from it.

Comparing both iteration methods, the inner loop of PI (Policy Evaluation) is a lot like VI. It is going as fast as VI, but it is doing a lot more work than VI, since VI does not have an implicit policy. Each iteration of PI is doing pretty much all the work of VI

plus multiple Policy Improvements, which is only done once in the VI, at the end, after obtaining V^*.

In addition, while the greedy policy converges in finite steps, the value function itself does not necessarily converge, but it is going to get closer to the solution, thus, it is used the ϵ hyperparameter to build a condition ($\Delta < \epsilon$) that defines the last iteration, like it is made in PI.

The following algorithm (2) explains how this method can be implemented.

Algorithm 2 - Value Iteration

procedure INITIALIZATION:
  $V(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$, except that $V(terminal) = 0$
end procedure

procedure VALUE EVALUATION:
  repeat
    $\Delta \leftarrow 0$
    for all $s \in \mathcal{S}$ **do**
      $v \leftarrow V(s)$
      for all $a \in \mathcal{A}$ **do**
        $Q(s,a) = \sum_{s'} P(s', r \mid s, a)\Big[r_1 + \gamma V(s')\Big]$
      end for
      $V(s) \leftarrow \max_a Q(s,a)$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
  until $\Delta < \epsilon$ ▷ the small positive number determining the accuracy of estimation
end procedure

procedure CONCLUSION:
  Output a deterministic policy, $\pi \approx \pi^*$: $\pi(s) = \arg\max_a \sum'_s P(s', r \mid s, a)\Big[r_1 + \gamma V(s')\Big]$
end procedure

4.2.3.3 Complexity of DP

DP methods are polynomial time algorithms for finite and discounted MDPs.

Assuming that $|\mathcal{S}|$ is the size of the set of spaces, $|\mathcal{A}|$ the size of the action space, γ is the discount factor and ϵ the variable that defines the accuracy of the estimation [46]:

- **Cost of PI**:
  - Policy Evaluation:
    - Solving a system of linear equations[2]: $O\left(|\mathcal{S}|^3\right)$

[2] In theory, value determination can probably be done somewhat faster, since it primarily requires inverting a $N \times N$ matrix, which can be done in $O\left(|\mathcal{S}|^{2,376}\right)$ [47]

* Iterative, using hyperparameters (γ and ϵ):

$$O\left(|\mathcal{S}|^2 \frac{\log\left(\frac{1}{\epsilon}\right)}{\log\left(\frac{1}{\gamma}\right)}\right) \tag{4.13}$$

– Policy Improvement:

* Solving a system of linear equations: $O\left(|\mathcal{A}||\mathcal{S}|^2\right)$

* Greedy, using hyperparameters (γ):

$$O\left(\frac{|\mathcal{A}|}{1-\gamma}\log\left(\frac{|\mathcal{S}|}{1-\gamma}\right)\right) \tag{4.14}$$

- **Cost of VI**:

– Only using state value function, $V^{\pi}(s)$ / $V^{*}(S)$:

$$O\left(|\mathcal{S}|^2|\mathcal{A}|\right) \tag{4.15}$$

– Using state action value function, $Q^{\pi}(s,a)/Q^{*}(s,a)$:

$$O\left(|\mathcal{S}|^2|\mathcal{A}|^2\right) \tag{4.16}$$

As explained in 4.2.3.2, it is easy to deduct that each iteration of PI is computationally more expensive than each iteration of VI. However, PI typically requires fewer iterations to converge than VI. So, it is not possible to ensure that PI is better that VI or vice versa, because PI has a faster convergence at the cost of greater computational expense.

The DP methods that was described so far used synchronous backups, i.e., all states are backed up in parallel. There are Asynchronous DP methods that back up states individually, in any order, and can significantly reduce computation [5]. However, throughout this document, asynchronous methods will not be considered since their improvements only rely on a computation point of view and the focus is the results' accuracy.

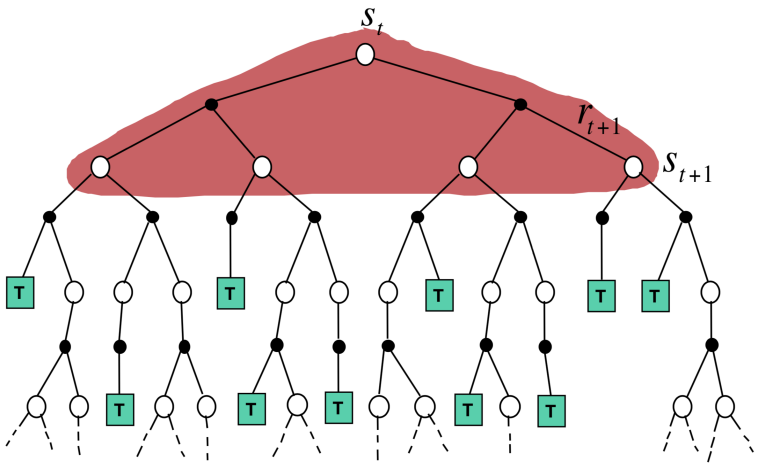


Figure 4.2: Dynamic Programming Methods tree [5]

4.3 Model-free Algorithms

This sort of algorithms relies on a *trial-and-error* learning approach to update its knowledge of the model. Therefore and referring Section 3.1, although the previous methods are useful to better understand all the topic that reinforcement learning covers, RL algorithms, truly belong to this section. These algorithms do not have the transition probabilities and rewards for the RL problem. It has no information about the model, it needs to learn it by itself. Therefore, the following methods will use the state-action value function, $Q(s,a)$, to update their policies, $\pi(a \mid s)$, taking into consideration the combination of different states and actions.

Before diving into the model-free methods, it is important to pay attention to the concepts defined in Section 3.5 to fully understand those methods.

4.3.1 Monte Carlo Methods

Monte Carlo (MC) methods only require experience, sample sequences of states, actions and rewards from current or simulated interaction with the environment. They learn value functions directly from episodes of experience. They are used in episode tasks, which means that it only calculates the reward at the end of an episode. Their learning process of the new value functions is made from sample returns - like an average of a bunch of numbers - with the MDP, whereas in DP methods the value functions were computed from knowledge of the MDP. However, as in the previous methods, the values functions and the corresponding policies still interact to obtain optimality. Besides that, the order of the problems remained the same, first the prediction problems - computation of V^π and Q^π for a fixed arbitrary policy π - then the control problem and its solution by

policy iteration.

Starting with the MC **prediction** and considering that each occurrence of state s in an episode is called a *visit* to s, there are two different approaches to estimate $V^{\pi}(s)$ [6]:

- **First-visit MC**: the average returns are determined only for first time the state s is visited in an episode.

- **Every-visit MC**: the average returns are calculated every time s is visited in an episode. It is described by:

$$V(s_t) \leftarrow V(s_t) + \alpha\,[G_t - V(s_t)] \tag{4.17}$$

  being G_t the current return following time t and α a constant step-size parameter.

As the number of returns are observed, the average should converge to the expected value - this underlies the MC methods. So, both first and every-visit MC converge to $V^{\pi}(s)$ as the number of visits (or first visits) to state s go to infinity.

At the end of this prediction there is the **control** phase, which follows the same pattern mentioned in DP. However, for MC policy iteration, it is natural to alternate between the evaluation and improvement processes on an episode-by-episode basis. In addition, while in the policy iteration method mentioned in Section 4.2.3.1 it was used the state-value function V^{π} for the evaluation. Here it is used the action-state value function Q^{π}, since there is no knowledge of the model and, consequently, of the state transition functions. This process is shown in Figure 4.3 and is described by Equation 4.18.

$$\pi_0 \xrightarrow{E} Q^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} Q^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi^* \xrightarrow{E} Q^* \tag{4.18}$$

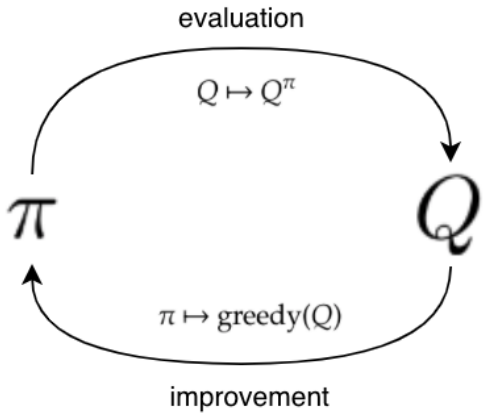


Figure 4.3: Monte Carlo Policy Iteration cycle

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [29] is a recent and strikingly successful example of decision-time planning. It has proven to be useful in an extensive variety of competitive settings; however, it is not limited to those scenarios. It can be effectively used for single-agent sequential decision problems if there is an environment model simple enough for fast multi-step simulation [6].

Figure 4.4 shows the MCTS. This tree is composed by big white circles - nodes -, which represent each state, and by small black circles, that correspond to the possible actions. The nodes that follow a blue arrow represent the states that were **selected**, being the root node the current state. The algorithm follows the **tree policy** - an informed policy that balances exploration and exploitation, e.g. selecting actions using an ϵ-greedy -, pursuing the blue arrows, and when it sees itself stuck in a state where it does not know the next best node to go further, the tree will **expand** from that node and the action-state value functions will be calculated through the **simulations** made in each scenario according to each possible action and following a **rollout policy**[3]. Then, recursively, it will **back up** what it has learned from those simulations all the way to the top. That updates the estimates ($\hat{Q}(s,a)$) of the previously selected nodes and expands the tree policy one step (one more action / blue arrow) until that last reached leaf, like a policy improvement. Then, a new selection phase will take place, using the new tree policy, until getting stuck again, hence a new expansion phase starts, repeating the whole process.

It is important to emphasize that the expansion follows some kind of greedy algorithm to prevent the exploration problems (Section 4.1.1) and it stops according to an *a priori* parameter proportional with the trade-off between speed, accuracy and resources.

This method benefits from online, incremental, sample-based value and policy improvement. Besides that, it saves action-value estimates attached to the tree edges and updates them using RL's sample updates [6].

MCTS methods have some peculiar properties [8]:

- useful for large state spaces;
- need lots of samples to get a good estimate;
- planning time independent of the number of states;
- running time exponential in the affected horizon - $O\left((|A|.X)^H\right)$, being X the number of steps executed.

[3]Rollout Algorithms are decision-time planning algorithms based on MC control applied to simulated trajectories that all begin at the current environment state. However, this concept is out of the scope of this dissertation, so will not be explored [6].

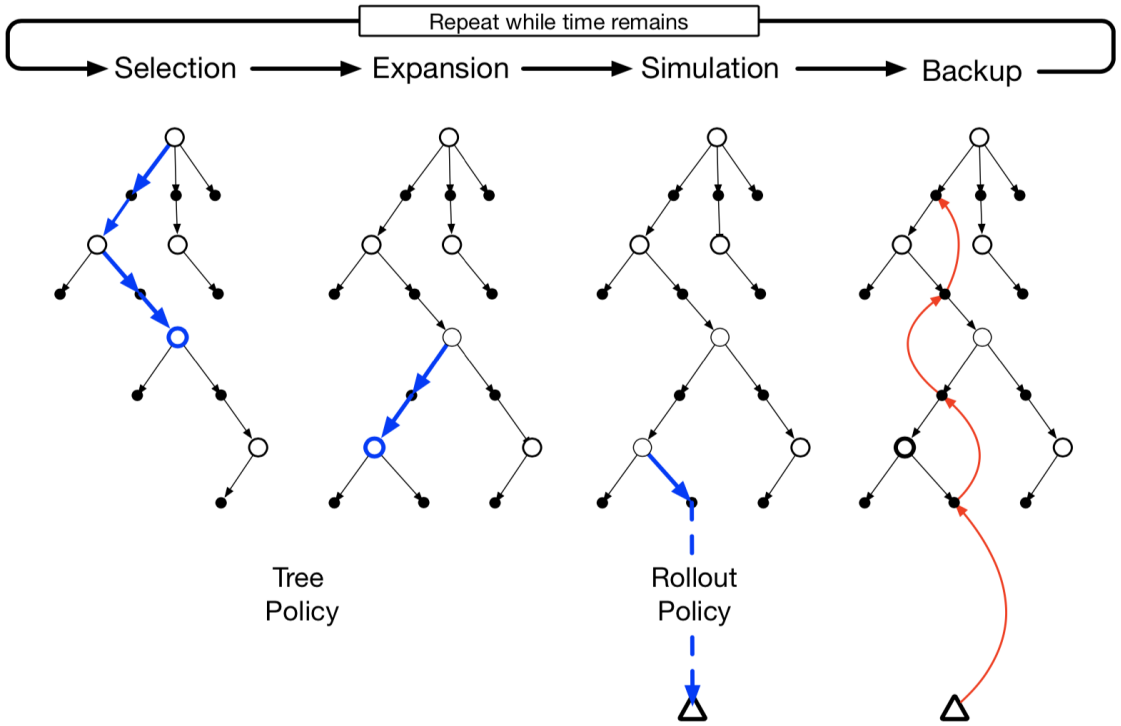


Figure 4.4: Monte Carlo Iteration cycle [6]

4.3.2 Temporal-Difference Methods

Temporal-Difference (TD)[4] learning is the combination of MC and DP ideas. As in MC methods, it can learn directly from raw experience without the knowledge of the model that describes the dynamics of the environment, and, just like DP it bootstraps, i.e., it updates their value estimates according to other value estimates [6].

What mainly distinguishes this method from the other two (MC and DP) are the approaches to the prediction problem, since the control problem follows, like MC, the idea of the generalized policy iteration, abstracted from DP - Section 4.2.3.1.

Whereas MC methods must wait until the end of the episode to determine the increment to $V(s_t)$ (when G_t is known) - Equation 4.17, in the **TD prediction** that wait is defined only by the time step. At time $t+1$, TD methods immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(s_{t+1})$, as shown in the following equation:

$$V(s_t) \leftarrow V(s_t) + [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{4.19}$$

Sample updates, used in TD and MC methods, differ from the expected updates used

[4] In order to simplify the definition of this concept according to the scope of this dissertation, it will not be mentioned the λ-return concept, so it will be assumed $\lambda = 0$, i.e., $TD(\lambda) = TD(0)$

in DP methods since the first ones are based on a single sample successor rather than on a complete distribution of all possible successors [6].

Taking back and comparing the three ways of updating the state value function, $V(s_t)$:

$$\begin{aligned} \text{DP}: &\quad V(s_t) \leftarrow E^{\pi}\left[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right] \\ \text{MC}: &\quad V(s_t) \leftarrow V(s_t) + \left[G_t - V(s_t)\right] \\ \text{TD}: &\quad V(s_t) \leftarrow V(s_t) + \left[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right] \end{aligned} \tag{4.20}$$

and considering that the current expected value is:

$$\begin{aligned} V^{\pi}(s_t) &= E^{\pi}\left[G_t \mid s_t\right] \\ &= E^{\pi}\left[R_{t+1} + \gamma G_{t+1} \mid s_t\right] \\ &= E^{\pi}\left[R_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t\right] \end{aligned} \tag{4.21}$$

the DP is the only one that knows the model of the environment; however, its target is an estimate because despite knowing $V^{\pi}(s_t)$, $V^{\pi}(s_{t+1})$ is not known and the current estimate, $V(s_{t+1})$, is used instead. In addiction, MC target is an estimate because the current expected value, $V^{\pi}(s_t)$, is not known as in DP (due to its model-based characteristic), so, a sample return, G_t is used in place of the real expected return. Concluding, the TD target is an estimate for both reasons: it samples the expected values, $R_{t+1} + \gamma V(s_{t+1})$, and it uses the current estimate, $V(s_{t+1})$, instead of the true and unknown $V^{\pi}(s_{t+1})$. Thus, TD methods combine the sampling of MC methods with the bootstrapping of DP [6].

The TD error is defined as $R_{t+1} + \gamma V(s_{t+1}) - V(s_t) = \text{TD target} - V(s_t)$

The **TD control** problem can be classified according to whether they use an on-policy or off-policy approach:

- **SARSA** is an on-policy method. SARSA is an acronym that means *State-Action-Reward-State-Action*, following the tree shown in Figure 4.5 and the Equation 4.22

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[R + \gamma Q(s',a') - Q(s,a)\right] \tag{4.22}$$

- **Q-Learning** is an off-policy method, that estimates a state-action value function for a target policy (greedy) that deterministically selects the action of highest value (apparently the best next action a') while following the behaviour policy (ϵ-greedy). Equation 4.23 represents this method.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[R + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \tag{4.23}$$

There is a third way in which TD methods can be extended to control, called actor-critic methods. These methods will be deeply explored in the next chapter, since they are

not strictly value-based methods as the ones described in this subsection.

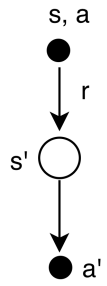


Figure 4.5: SARSA tree

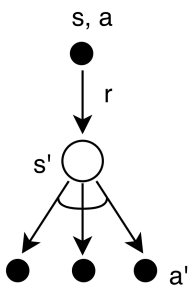


Figure 4.6: Q-Learning tree

4.3.3 n-step Method

When comparing MC and TD methods observing their equations (4.20) and the figures 4.7 and 4.8, it is easy to understand that both arise from an n-step method [6], described by the following equations:

$$\begin{aligned} &n = 1 \quad \text{(TD)} \quad && G_t^{(1)} = R_{t+1} + \gamma V(s_{t+1}) \\ &n = 2 && G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(s_{t+2}) \\ &\vdots \\ &n = \infty \quad \text{(MC)} && G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T \end{aligned} \tag{4.24}$$

being $t \in [0,\ T]$ and

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(s_{t+n}) \tag{4.25}$$

thus, n-step temporal-difference learning is defined as:

$$V(s_t) \leftarrow V(s_t) + \alpha(G_t^{(n)} - V(s_t)) \tag{4.26}$$

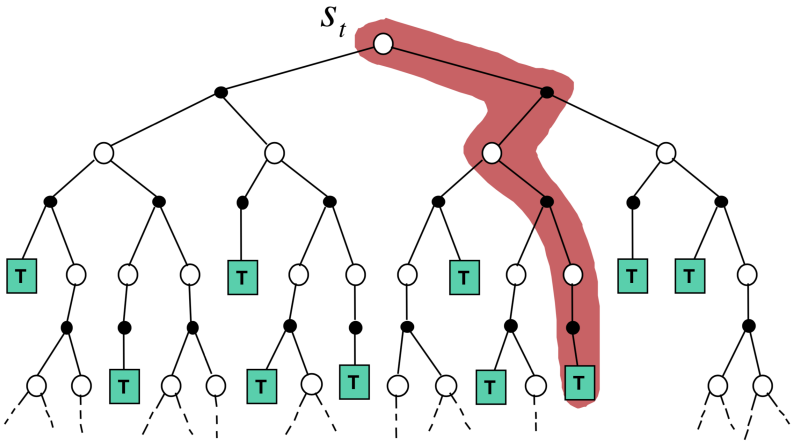


Figure 4.7: Monte Carlo Method tree [5]

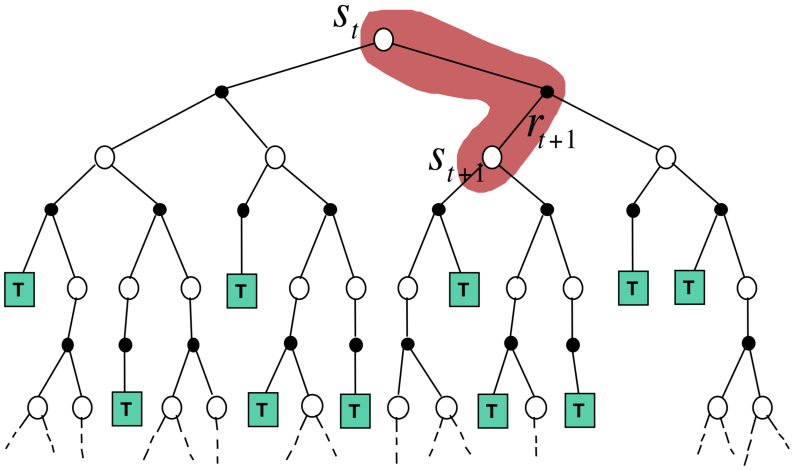


Figure 4.8: Temporal-Difference Methods tree [5]

4.4 Conclusion

It is important to emphasize that the worst case convergence guarantees of LP methods are better than those of DP. In addition, when compared to DP methods, LP methods become impractical at a much smaller number of states.

MC methods learn value functions and optimal policies from experience in the form of sample episodes, which give it three advantages over DP methods [6]:

1. MC methods can be used to learn optimal behavior directly from interaction with the environment, without needing any model that describes the dynamics of the environment.

2. For many applications, it is easier to simulate sample episodes, even with the difficulties of building the explicit model of transition probabilities required in DP.

3. It is easy and efficient to restrict the subset of states to a region of special interest as it is done in MC methods, removing the expensive evaluation of the rest of the state set.

In addition to these three advantages, not using the bootstrap concept implemented in DP methods may allow MC methods to reduce the violation of Markov's property (Section 3.3.2).

Whereas MC methods only work for episode task problems - it has to wait until the end of the episode to get the reward - and learn from complete sequences, TD methods work for both episodic and continuous task problems - it will only wait until the next time step (not next episode) to update the value estimates -, and it can learn from incomplete sequences too.

TD methods are announced in [6] as the most widely used reinforcement learning methods, due to their great simplicity, i.e., they can be applied online, while getting experience from its interaction with an environment, requiring a minimal amount of computation, since they can be expressed nearly completely using single equations that can be implemented with small computer programs.

Convergence is a tricky question when comparing TD and MC methods, since "no one has been able to prove mathematically that one method converges faster than the other" [6]. Nevertheless, and considering the example 6.2 presented in [6], TD methods have usually been found to converge faster that constant-α MC methods on stochastic tasks.

Summing up, Figure 4.9 presents an unified view of reinforcement learning methods.

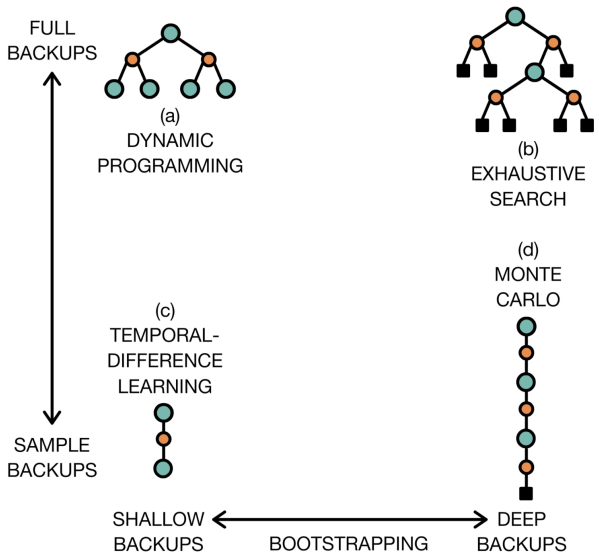


Figure 4.9: Unified View of Reinforcement Learning methods [7]

Chapter 5

Policy Optimization

5.1 Introduction

Policy optimization, as its name implies, combines the use of optimization algorithms as a resource for policy updating when considering policy-based methods to find the optimal solution.

Most of the policy optimization algorithms presented in the literature are exposed in chapters 3 and 4 of [48], referencing their usefulness and several applications. Some examples of these algorithms are: REINFORCE [49], Actor-Critic [50], Deterministic Policy Gradient (DPG) [51], Deep Deterministic Policy Gradient (DDPG) [52], Trust Region Policy Optimization (TRPO) [53], Synchronous Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C) [54], Actor-Critic with Experience Replay (ACER) [55], Actor-Critic using Kronecker-factored Trust Region (ACKTR) [56], Proximal Policy Optimization (PPO) [57], Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [58], Soft Actor-Critic (SAC) [59], Distributed Distributional Deep Deterministic Policy Gradient (D4PG) [60], Twin Delayed Deep Deterministic (TD3) [61].

Throughout this chapter some of these algorithms will be analyzed, pointing to the proposed solution. Gradient Descent, also called back-propagation when applied to neural networks, is one of the most popular optimization strategies, also used in some of the previously mentioned algorithms. This strategy will be detailed according to the Policy Gradient Methods that will be discussed next.

5.2 Policy Gradient Methods

Policy gradient methods belong to a broader class of policy-based methods that includes, among others, evolution strategies [48].

Policy-based methods directly optimize the expected cumulative reward, known as the objective function, by finding a good policy using parameterized policy. These policies

can select actions without consulting a value function; however, it may be used to learn the policy parameter. The policy's parameter vector is defined as $\theta \in \mathbb{R}^l$, $l \ll |\mathcal{S}|$.

Therefore, the probability that action a is chosen at time t given the state s, at the same time, with parameter θ is:

$$\pi(a \mid s, \theta) = \pi_\theta(a \mid s) = Pr\{a_t = a,\ s_t = s,\ \theta_t = \theta\} \tag{5.1}$$

It is assumed that π is differentiable with respect to its parameter, thus $\frac{\delta\pi(s,a)}{\delta\theta}$ exists [62].

If a learned value function is used as well, then $w \in \mathbb{R}^d$ corresponds to its weight vector.

These methods based their learning of the policy parameter on the gradient of some scalar performance measure $J(\theta)$ with respect to the policy parameter. They aim to maximize performance, so their updates approximate gradient ascent in J:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{5.2}$$

where α is a positive-definite step size as mentioned previously in this document and $\widehat{\nabla J(\theta_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure according to θ_t [6]. Therefore, the policy parameters are updated approximately proportional to the gradient:

$$\Delta\theta \approx \alpha\frac{\delta J(\theta)}{\delta\theta} = \alpha\nabla_\theta J(\theta) \tag{5.3}$$

where $\nabla_\theta J(\theta)$ is the policy gradient given by:

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\delta J(\theta)}{\delta\theta_1} \\ \vdots \\ \frac{\delta J(\theta)}{\delta\theta_n} \end{pmatrix} \tag{5.4}$$

If this can be achieved, it is usually granted that θ will converge to a locally optimal policy in the performance measure $J(\theta)$ [62].

Computing the gradient $\nabla_\theta J(\theta)$ is tricky, since it depends on the selected action (directly determined by π_θ) and the stationary distribution of the states while following the target selection behaviour (indirectly determined by π_θ). Policy gradient is part of model-free methods, thus, not knowing the environment makes it difficult to estimate the effects that a policy update has on the state distribution. Here comes the *policy gradient theorem*, which simplifies the gradient computation of $\nabla_\theta J(\theta)$ to:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a \mid s) \\ &\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta\pi_\theta(a \mid s) \end{aligned} \tag{5.5}$$

This is the scheme that characterizes the policy gradient methods.

How to measure the quality of a policy π_θ?
Considering a continuous task, there are two ways:

- using average value per time-step:

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s) \tag{5.6}$$

- or average reward per time-step:

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) R_s^a \tag{5.7}$$

where $d^{\pi_\theta}(s) = lim_{t\to\infty} Pr\{s_t = s \mid s_0, \pi_\theta\}$ is a stationary distribution of states under π_θ, which is assumed as real and is independent of s_0 for all policies [62].

Summarizing, the process is: give state; get action probability; take an action in the environment given by the policy π; observe the next state and rewards; update the parameters in the network based on the rewards collected - backpropagation -; repeat it until the end of the training process.

These methods have some **advantages**, such as better convergence properties, effective in continuous spaces and can learn stochastic policies; but they also have some **disadvantages**, as customarily converge to a local rather than a global optimum and their evaluation of the policy is typically inefficient and high variance [5].

5.2.1 Parameterized Policy

Parameterized policies use a set of parameters, adjusted to change the behaviour through an optimization algorithm, in order to return computable functions.

- **Stochastic Policy Gradient** are usually denoted by π, according to the following equation:

$$a_t \sim \pi_\theta(\cdot \mid s_t) \tag{5.8}$$

  where θ corresponds to the policy parameter and the π_θ the stochastic policy.

- **Deterministic Policy Gradient** are usually denoted by μ, according to the following equation:

$$a_t = \mu_\theta(s_t) \tag{5.9}$$

  where θ corresponds to the policy parameter and μ_θ is the deterministic policy.

5.2.2 Actor-Critic Methods

While value-based methods learn the value function with an implicit policy (e.g. ϵ-greedy) and policy-based methods have no value function, learning the policy by itself, the actor-critic appears using some of these two methods, it both learns the value function and policy (Figure 5.1).

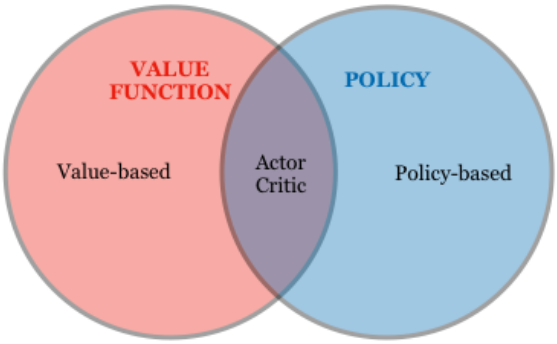


Figure 5.1: Venn Diagram representing the relation of Actor-Critic, Value and Policy-based methods

The previously mentioned types of parameterized policies can be represented by a neural network updated by gradient ascent. The policy gradient typically requires an estimate of a value function for the current policy and the actor-critic architecture appears as one common approach [4].

An actor-critic algorithm [50] learns the policy and the value function by two different entities: the agent and the critic. The value function is used for bootstrapping, i.e., updating a state from subsequent estimates, in order to accelerate the learning process and reduce variance [48].

The actor is related to the policy, it updates the policy parameter θ for $\pi_\theta(a \mid s)$ in the direction suggested by the critic, which estimates the approximate state-action value function for the current policy π as $Q(s,a;\theta) \approx Q^\pi(s,a)$. So, while the actor decides which action to take, the critic tells the actor how good its action was and how good it should adjust. This adjustment is made according to the *TD error* calculated with the value function, as shown in Figure 5.2.

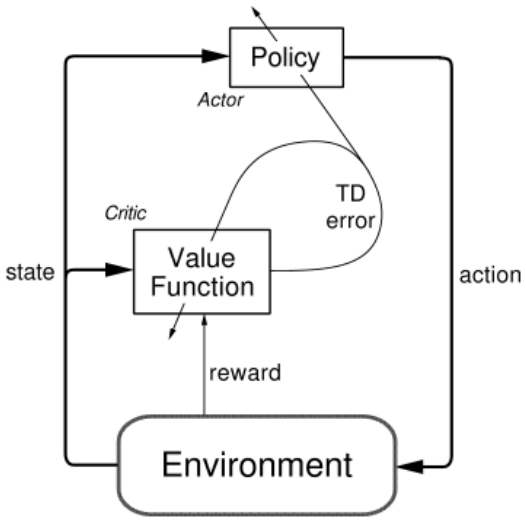


Figure 5.2: Actor-Critic architecture [6]

Although the critic estimations can be defined from a set of tuples $< s, a, r, s' >$, possibly taken from a replay buffer as a simple off-policy approach that uses a pure bootstrapping algorithm TD(0) where, at each iteration, the current state-action value function is updated towards a target value:

$$Y_k^Q = r + \gamma Q(s',\ a = \pi(s');\ \theta) \tag{5.10}$$

this is not computationally efficient as it uses a pure bootstrapping technique that has a slow reward propagation backwards and is prone to instabilities. Thereby, the ideal architecture should be **sample efficient** - being able to use both off and on-policy trajectories, through a replay buffer - and **computationally efficient** - it should be able to profit from the stability and rapid spread of rewards of on-policy methods to samples obtained from near on-policy behaviour policies [4].

5.2.3 Deep Deterministic Policy Gradient

This method was presented in [52] as an actor-critic and model-free DDPG algorithms that can be applied in continuous action spaces, through the extension of Deep Q-Network (DQN) [63] and DPG [51].

With reference to DPG, it uses value gradient method to estimate the gradient with backpropagation, and it avoids the optimization of action value function at each time step in order to obtain a greedy policy as in Q-Learning. This will make it unfeasible in complex action spaces with large and unconstrained function approximators, such as deep neural networks [48].

With regard to its usage of DQN, it deploys experience replay and an idea similar to target network, like a soft target which, rather than copying the weights directly as in DQN, it updates the wights θ' of that soft target network slowly to track the learned

networks weights θ, as followed:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{5.11}$$

where $\tau \ll 1$

As an off-policy, this method learns the actor policy using experiences from an exploration policy that adds noises sampled from a noise process in the actor policy. It is shown in [48] that "DDPG can solve problems with 20 times fewer steps of experience than DQN". However, as almost all model-free RL methods, it still needs a lot of training episodes to find solutions.

Chapter 6

Proposed Solution

This chapter starts with the problem analysis in agreement with the RL concepts presented in Chapter 3. In the next section, it is determined the type of machine learning algorithm used to solve the problem stated in Section 1.3, based on the studies exposed in Chapter 2. After that, it is described which are the most compliant methods to use according to the analysis of the problem and the approaches detailed in Chapter 5. Finally, the proposed ML algorithm is explained in detail.

6.1 Assumptions and Problem Formulation

There are multiple studies and algorithms tested to solve several restrictions that appear when trying to apply core learning algorithms to specific use cases. Each problem has its own properties. Therefore, when building an algorithm to solve a problem it is necessary to take into account its properties, mapping them to the diverse concepts mentioned in Section 3.5.

There are several assumptions to take into account in order to decide which is the best path to follow:

- The agents are defined as UAVs carrying Wi-Fi APs and the environment with characteristics of a TCE, such as a music festival.
- The environment is defined as a **bi-dimensional scenario**, so the agents will not be able to move in the Z axis (not increasing or decreasing their altitude).

  Due to the nature of the problem and in order to simplify the algorithm without compromising the goal, it is considered a discrete domain. Therefore, the action space is limited to the X (horizontal) and Y (vertical) axis, with the possible actions:

  - UP: increase the Y axis value;
  - DOWN: decrease the Y axis value;

  - RIGHT: increase the X axis value;
  - LEFT: decrease the X axis value;
  - STAND: the coordinates do not suffer any operation.

  Although the action space seems to be defined in a discrete domain that does not mean that the algorithm needs to follow a discrete action space. As the problem was defined to fulfill the users' demands, it is necessary to manage the airborne network topology and, for that purpose, they need to take actions in the real world, with a **continuous action space**. However, and in order to test which is the best direction to take that action, the previously mentioned action space was defined as discrete.

- Agents' rewards are calculated with a **cumulative discounted return function**, G_t.
- Agents act without knowledge of the model due to the unpredictable users' movement. This falls into the **model-free** algorithms scope.
- As the goal of the proposed solution is to increase QoS in a TCE environment managing the airborne network topology, the training process needs to be evaluated according to this scenario, following a **continuous task**, where there is no terminal state that defines the end of specific episodes. Thus, Monte Carlo methods are not an option.
- As explained in Section 3.5, the **off-policy** is the key to learn predictions in a dynamic world. It is able to solve complex problems, so this will be the type of policy applied in the proposed solution. This statement removes SARSA from the Temporal-Difference methods' options.

Considering the previous assumptions, Q-Learning, a value-based method, and Policy Optimization methods, which can be a combination of value and policy-based methods, are the two main possibilities available for the proposed solution. The first one has been very much used in DRL with the application of Deep Q-Networks [63–65]. However, its restrictions about the use of state-action value functions, single-agent algorithms [66] and scalability concerns [24] limit its use according to the following constraints:

- The algorithm will be running in real-time, in a continuous task, thus it needs to converge fast to the optimal policy in order to ensure a good QoS to the users as soon as possible. Thus, **policy-based methods** are considered a better approach, since they usually have better convergence properties than value-based methods and are effective in high-dimensional or continuous action spaces [48].
- The problem is defined as a **POMDP** (Section 3.5.1) on account of its decision-making issue and need to learn how to act in dynamic scenarios, exploring different situations.

- Considering that the number of UAVs is greater than or equal to one, this problem needs to be treated as a **multi-agent problem**. The system is assumed to be a **collaborative multi-agent** in order to make all UAVs have the same goal, increasing QoS.

6.2 Proposed Solutions

Considering the main characteristics of the 4 types of machine learning algorithms, supervised, unsupervised, semi-supervised and reinforcement learning, RL algorithms are the ones that are most suitable to the problem addressed in this work.

RL algorithms take actions according to changes in the environment, following an objective function. In the same way, the resolution of the stated problem involves the adaptation of the airborne network topology according to the dynamic movements of the users, seeking to ensure the best possible QoS.

Concerning the problem to be addresses in this work, the RL algorithm to be defined considers the *environment* as the users' disposal in the field and the *agent* as the airborne network topology. Thus, the *actions* the *agent* will have in the *environment* will be reflected in changes in the QoS provided to the users. Likewise, the *state*, that will be passed from the *environment* to the *agent* corresponds to the users' disposal and the airborne network topology facing the previous action and the *reward/penalty* reflects its impact in the objective function, guiding the training process to the goal. In this way, the *agent* can "learn from mistakes", learn from the *action* that was done before, moving forward to the optimal solution and realizing how to deal with different scenarios.

However, DL appears as an extra tool that can be added to an RL algorithm, giving it the ability to extract deeper features from the networks that can help it learn faster and somehow smarter [7]. In this case, deeper features from the *environment* that can help it to learn some habits about the users and which, statistically, can improve the choices made by the algorithm, hence, increasing the average QoS. Therefore, DRL seems to be the best technique to seek an optimal solution for the formulated problem.

While the methods presented in Chapter 4 correspond, mostly, to approaches that return an optimal policy and, consequently, exact solutions, the methods introduced in Chapter 5 correspond to approximate solution methods that can be applied effectively to more complex and larger problems (bigger set of state and action spaces). The first ones are fundamental core learning algorithms that are used to build these approximate solution methods. The environment defined in this use case is characterized as a dynamic and real-time scenario, which brings up the need to use a vast set of state and action spaces.

There are three main approaches to solve an RL algorithm as shown in Figure 6.1. This figure explains that given a loop (+) where a state s, action a and reward r are passed as

input of, either a model learner, a value update or a policy search method, the algorithm will finally get a policy π. Value-based methods, known as model-free algorithms (Section 4.3), only derive the optimal policies after optimizing value functions, while policy-based methods directly optimize the objective function. It is natural to expect that policy-based methods are more useful in continuous space, since there is an infinite number of actions and/or states to estimate the state-action value in a continuous space rather than in a discrete space. Therefore, value-based approaches are computationally much more expensive and have a slower convergence.

As shown in Figure 6.1 policy-based methods describe a higher level of direct learning, which can be easily represented as faster convergence, and a lower supervised component, that refers to its independence of the model. Therefore, these methods are inferred to be more appropriate for real-time learning problems and dynamic scenarios with a low level of model knowledge.

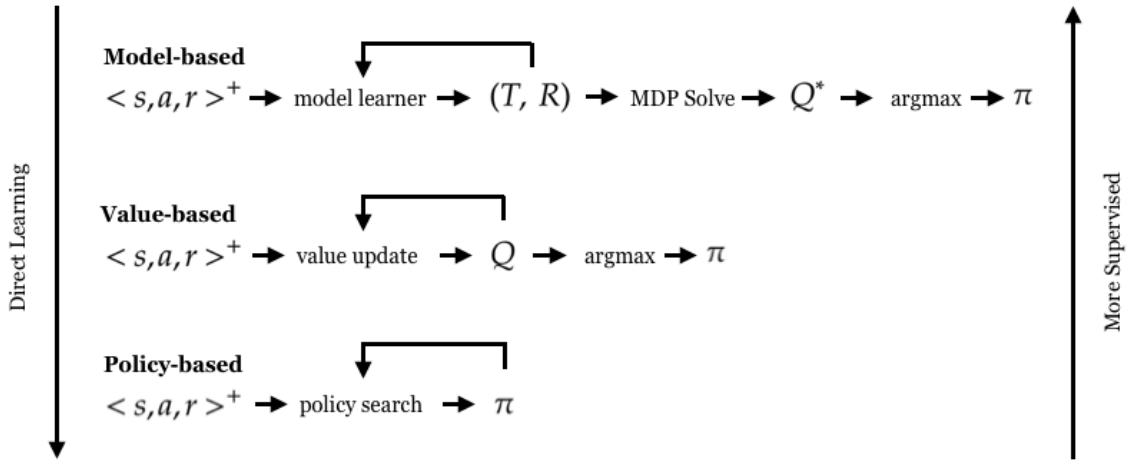


Figure 6.1: Reinforcement Learning main methods [8]

Chapter 5 explained with more detail how policy gradient methods - a class of policy-based method - work, and which are the most important and useful algorithms that use these methods.

6.3 Algorithm Planning

Following the constraints of Section 6.1, the main approach should be policy optimization, namely, policy gradient methods as explored in Chapter 5.

REINFORCE is a Monte Carlo Policy Gradient that uses likelihood ratio method by sampling returns from interactions with the model-free environment. It can use a baseline value from the return G_t to reduce the variance of gradient estimation while keeping the bias unchanged: $A(s,a) = Q(s,a) - V(s)$ [4].

Although REINFORCE methods learn both a policy and a state-value function, it is not considered an actor-critic method because its state-value function is not used for bootstrapping, so it is not a critic. The bias introduced by bootstrapping is often beneficial, since it reduces variance and accelerates learning. Even using the previously mentioned and unbiased baseline, it converge asymptotically and slowly to a local minimum, like all Monte Carlo methods, due to the calculation of estimates with high variance. Thus, this method is unhandy to use in continuing and online problems, like this one [6]. Therefore, the starting point in this algorithm planning is the actor-critic method.

As referred in [4], actor-critic architecture is a common approach in the context of collaborative multi-agent systems. It is implemented with a centralized critic during learning and a decentralized actor. Besides that and as already affirmed in Section 6.2, "deep RL have a large potential for many real-world tasks that require multiple agents to cooperate in domains such as robotics, self-driving cars, etc." [4]. [4] refers that, in DRL, both actor and critic can be represented by non-linear neural network function approximators.

According to the previous affirmations and the content presented in Section 5.2.3, DDPG seems to be the next right step in the algorithm planning, since it shows very good results using an actor-critic architecture that matches most of the conclusions announced in Section 6.1.

[58] presents the **MADDPG algorithm**, which satisfies the last missing constraint, a cooperative multi-agent system. Therefore, this will be the base algorithm to use for solving the stated problem.

6.4 MADDPG Algorithm

This algorithm proposes a general-purpose multi-agent learning algorithm which (1) leads to learned, at execution time, policies that only use local information, like from their own observations (POMDP), (2) does not assume a differentiable model of the environment dynamics or any particular structure on the communication method between agents (model-free), and (3) is applicable to both cooperative and competitive interactions, or mixed interactions involving physical and communicative behaviour.

In the viewpoint of an agent, the environment is non-stationary regarding the quick updates of the other agents' policies, which makes them unknown for it. MADDPG appears as an actor-critic model redesigned particularly for handling such a dynamic environment and interactions between agents. It assumes a centralized training with decentralized execution, which permits to ease training given the usage of extra information by the policies, as long as that information is not used at test time. To do so it was proposed a simple extension of actor-critic policy gradient methods where the actor only has access to the local information, whereas the critic is augmented with extra information

about the policies of the other agents. After the training is concluded, the execution phase only uses the local actors acting in a decentralized manner.

Since the centralized critic function uses the decision-making policies of the other agents, this paper shows that the agents can learn approximate models of other agents online, using them in their own policy learning process, effectively.

It was also introduced a method that uses an ensemble of policies at the agents' training procedure in order to improve the stability of multi-agent policies. However, this requires a robust interaction between the multiple collaborator and competitor policies.

6.5 Proposed Cooperative Mesh MADDPG Algorithm

The scenario used in this algorithm is build from a *gym* environment [67] composed by users and UAVs carrying a Wi-Fi AP. The actions imposed by the policy are processed in a discrete domain as mentioned in Section 6.1. However, their output describes the flawless movement of the UAVs according to a continuous trajectory followed by the proportional forces calculated with the action taken and the correspondent acceleration. In addition, and in contrast to the MADDPG algorithm, the users execute random movements in a continuous task. However, it is used an episode task approach after some steps in order to increase the world dynamics with a completely new positioning of the users and the agents. Therefore, after each episode it is calculated and stored the cumulative average reward and it is made a reset to the environment, until the predefined number of steps are complete. In short, there are three main predefined values that manages the steps of this algorithm:

- *episode_length*: correspond to the estimate number of moves that the agent needs to do to stabilize near a so-far optimal position;
- *total_episodes*: corresponds to the total number of episodes defined for the training process;
- *batch_size*: corresponds to the number of collected experiences for episode randomly taken from the replay buffer, $\mathcal{B}$, and which defines the size of the mini-batch samples S.

The agent follows a specific trainer with the following characteristics:

- two distinct Multi-Layer Perceptron (MLP) model: one for the actor and other for the critic updates (deeply explained at Section 6.5.2),
- an Adam optimizer used in both models to minimize the loss function w.r.t. variables in Q and policy functions [68],

- and a replay buffer, B, to save the observations, actions, rewards and new observations. The experience replay buffer is used when the following conditional statement is fulfilled:

$$|\mathcal{B}| \geq batch\_size \cdot episode\_length \tag{6.1}$$

Considering the MADDPG, the proposed algorithm follows a multi-agent POMDP system, where:

- N deterministic and continuous policies are used as $\mu_{\theta_i} = \mu_i$[1] w.r.t. parameters θ_i, which will suffer softly updated according to the following equation:

$$\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i' \tag{6.2}$$

- $Q_i^{\mu}(X, A)$ is a centralized action-value function that takes as input the N actions from the N agents, $A = (a_i, \ldots, a_N)$, in addition to some state information X, which consist of the observations of all agents, $X = (w_1, \ldots, w_N)$;

- the experience replay buffer B contains the tuples (X, A, R, X'), caching experiences of all agents, where given a current observation X, agents take actions A, and get rewards $R = (r_1, \ldots, r_N)$, leading to a new observation X'.

- the gradient is written as:

$$\nabla_{\theta_i} J \approx \frac{1}{S} \sum_j \nabla_{\theta_i} \mu_i(w_i^j) \nabla_{a_i} Q_{\mu_i}(X^j,\ A^j \mid a_i = \mu_i(w_i^j)) \tag{6.3}$$

  where the tuples $(X^j,\ A^j,\ R^j,\ X'^j, \theta^j,\ w^j)$ belongs to the sample, S, randomly taken from the replay buffer, B.

- the centralized action-value function is updated with regard to the following loss function:

$$\begin{aligned} \mathcal{L}(\theta_i) &= \frac{1}{S} \sum_j (y^j - Q_{\mu_i}(X^j,\ A^j))^2\ , \\ y^j &= R_i^j + \gamma Q_{\mu_i'}(X'^j,\ A'^j \mid a_k' = \mu_k'(w_k^j))\ ,\ k \in [1, N] \end{aligned} \tag{6.4}$$

  where $\mu'_{\theta_j} = \{\mu_{\theta_1'}, \ldots, \mu_{\theta_N'}\}$ is the set of target policies with delayed parameters θ_i'. The loss function uses Mean Square Error as the evaluation metric for the regression problem that DQN tries to solve when making new predictions from the last sample of collected experiences taken from the replay buffer. This metric uses the square of the error to calculate the loss function, which enables the effect of larger errors to become more pronounced then smaller error, so the model can now focus more on larger errors.

[1] To simplify the notation, from now on, it will only be considered μ_i in substitution of μ_{θ_i}

Algorithm 3 - Cooperative Mesh MADDPG Algorithm

env ← *tceEnvironment*()
for episode = 1 to *total_episodes* **do**
  $X \leftarrow env.reset()$
  $\epsilon \leftarrow$ random $\epsilon - greedy$ process for action exploration
  for step = 1 to *episode_length* **do**
    $a_i = \mu_\theta^i(w_i) + \epsilon$ ▷ softly update
    Execute $A = (a_1, \ldots, a_N)$
    Update Reward Function according to algorithm 4 or 5
    $X', R \leftarrow env.step(A)$
    Collect experience storing (X, A, R, X') in replay buffer $\mathcal{B}$
    $X \leftarrow X'$
    for agent $i = 1$ to N **do**
      Take random mini-batch of S samples from $\mathcal{B}$ - (X^j, A^j, R^j, X'^j, θ^j, w^j)
      Calculate TD target: $y^j = R_i^j + \gamma Q_{\mu_i'}(X'^j, A'^j \mid a_k' = \mu_k'(w_k^j))$, $k \in [1, N]$
      Train Q value model
      Update Critic minimizing loss function: $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j (y^j - Q_{\mu_i}(X^j, A^j))^2$
      Train policy model
      Update Actor using the sampling policy gradient:
$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(w_i^j)\nabla_{a_i} Q_{\mu_i}(X^j, A^j \mid a_i = \mu_i(w_i^j))$$
    end for
    Update target network parameters: $\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$
  end for
end for

While the scenario defines the number of users, M, present in the environment, the algorithm manages the number of agents, N, according to their (C)apacity, following the equation:

$$N \geq \frac{M}{C \cdot (1 - \zeta)} \tag{6.5}$$

where $\zeta \in \mathbb{R}^+$ correspond to an hyperparameter that defined the percentage of redundant channel occupancy avoiding the Wi-Fi AP saturation carried by the UAV.

The definition of the reward function appears as a vital part of the DRL algorithm, since it drives the algorithm to the optimal solution according to the goal, which is increase QoS. In addition, the learning process that is guided by this reward function should be followed with two neural networks models, one to estimate the state-action value function and other to achieve the new so-far optimal policy. These topics will be discussed in the next sections.

6.5.1 Reward Function

Reward function is a crucial topic with respect to the way that the RL algorithm should follow in order to approach the optimal solution. This algorithm calculates the reward function according to two different options, considering the users' point of view associated with the cooperative component of the agents or the agent's connection base related to its link state. It is important to emphasize that the reward function is dependent of the agent, being the return value only associated to it.

In addition, the reward values of these approaches will be associated with the distance between the agents and the users, due to its relation with QoS as will be explained in Section 7.1. Therefore, the objective will be minimize the distance between each user and the agents in order to maximize the QoS provided to the users.

The first approach relies on the calculation of the distance between each agent and the user. Thus, the reward value decreases by the distance between the nearest agent and the current user while all users are traversed, as stated in the following algorithm:

Algorithm 4 - Cooperative Reward Function

1: $rew \leftarrow 0$ ▷ reward value
2: **for all** $u \in$ Users **do**
3: **for all** $a \in$ Agents **do**
4: **if** $isCollision(a, curr\_a)$ **then** ▷ $curr\_a$ is the current agent receiving this reward
5: $rew \leftarrow rew - collision\_factor$
6: **end if**
7: $distances[N] \leftarrow \sqrt{u.pos^2 - a.pos^2}$
8: **end for**
9: $rew \leftarrow rew - \min(distances[N])$
10: **end for**

The reward is the same for all agents (shared reward). It is calculated from the user's point of view and considering the shortest distance to the nearest agent as penalty, this is, the accumulative sum of these distances is what the algorithm seeks to improve / reduce, since this is considered a negative reward, i.e., the greater the distance's absolute value the worse.

It is further considered that for each collision between the agents the reward is decremented by $collision\_factor$ units - $isCollision()$. This is one of the parameters that are changed along the This is due to two main reasons: (1) avoiding collisions between the UAVs assuming an equal and constant altitude (Z axis invariant) and (2) even considering a variation in the Z axis, this prevents a waste of coverage by the overlap of each UAV.

The second algorithm was created to prevent specific cases were a small percentage of the users, named group B, would be disposed far away from the platoon, named group A, which was well covered by a certain number of UAVs, numbered as agents 1. Therefore, in this case, the other number of UAVs, numbered as agents 2, would be able to join the group B, but that does not happen since the previous algorithm used a shared reward,

not mapping the users with the UAVs. According to the Algorithm 4, agents 2 would be disposed between the two groups, proportionally to the crowd's size and their center. Figure 6.2 shows one example of this specific case with two agents trying to cover four blocks of users.

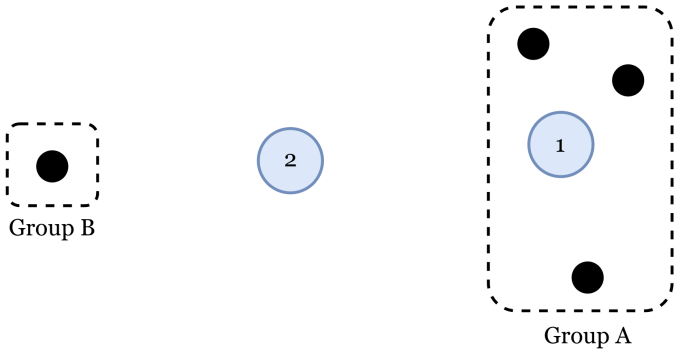


Figure 6.2: Scenario obtained with Algorithm 4 where the *agent 2* should cover the *group B* considering that *agent 1* is capable of covering *group A* by himself

Whereas the Figures 6.3 and 6.4 present the results of the usage of the two different algorithms to this special case, the Figures 6.5 and 6.6 detail the essential distance calculations used in both algorithms, highlighting the ones that directly influence the reward function (green/dashed arrows).

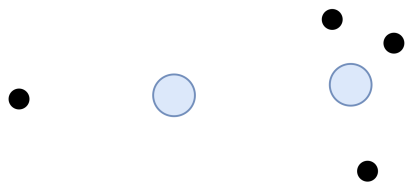


Figure 6.3: Result of Algorithm 4

Figure 6.4: Result of Algorithm 5

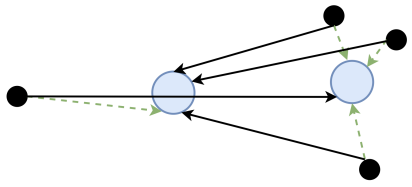


Figure 6.5: Metric used in Algorithm 4

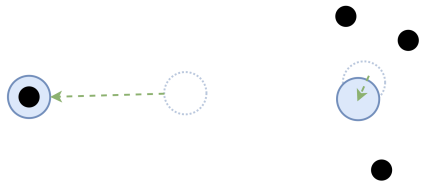


Figure 6.6: Metric used in Algorithm 5

The second approach is a link state method that confines the agents' association with the nearest users according to a centralized prioritization of the agents and a cumulative distribution function, which defines the number of users connected to the agent pursuant to their distances. This algorithm combines the cooperative multi-agent characteristic to increase the same goal in a competitive dispute for the users' connection according to a centralized order that enables the agent's communication.

Algorithm 5 - Link State Reward Function

$rew \leftarrow 0$ ▷ reward value
for all $a \in$ Agents **do**
  if $isCollision(a, curr\_a)$ **then** ▷ $curr\_a$ is the current agent receiving this reward
    $rew \leftarrow rew - collision\_factor$
  end if
  for all $u \in$ Users **do**
    $distances[M] \leftarrow \sqrt{u.pos^2 - a.pos^2}$
  end for
  $agent\_distances[N] \leftarrow distances[M]$
end for
Order $agent\_distances[N]$ according to the $priority\_metric$ (Equation 6.6)
$conn \leftarrow 0$
for all $agent\_d \in agent\_distances[N]$ **do**
  $conn \leftarrow$ number of users already connected
  if x is the last agent **then** ▷ the last agent gets the remaining users ($M - conn$)
    $conn\_users[N] \leftarrow getConnectedUsers(agent\_d, M - conn)$
  else
    $conn\_users[N] \leftarrow getConnectedUsers(agent\_d)$
  end if
  if $agent\_d$ is the current agent **then**
    set i as the current index of $users[N]$
    break
  end if
end for
for all $u \in$ Users **do**
  if $u \in conn\_users[i]$ **then**
    $rew \leftarrow rew - \sqrt{u.pos^2 - curr\_a.pos^2}$
  end if
end for

The *link state reward function* algorithm (5) assumes that all the agents have the same capacity and calculates the distances in line with the agent's point of view, i.e., while in the *cooperative reward function algorithm* (4) the distances were determined from the current user to each agent (passing through all the users), here it is the opposite, the distances are calculated from the current agent to each user (passing through all the agents), requiring the communication between the agents. After these calculus, the list of distances, $agent\_distances[M]$, is sorted according to the value given by the Equation 6.6

creating a prioritizing list of agents.

$$priority\_metric = K \cdot \langle\ agent\_distances[M]\ \rangle + (1 - K) \cdot \sigma \tag{6.6}$$

where $\langle\ agent\_distances \rangle$ is the average of the elements of the list *agent_distances* and K is an hyperparameter with regard to the weight given to that average.

Equation 6.6 arises from the need to determine a way to define the priority of the agents' list, taking into account the distance of each agent to the users. However, it is relevant to note the use of the standard deviation to avoid cases such as those shown in figures 6.7 and 6.8, where a lower mean does not imply a greater number of users near the agent.

Analyzing those figures it is easily recognized that even with two users contributing for a low average, the blue/dashed agent should not be the first to choose with which users it will connect, since the other agent have a more consistent distance with the majority of the users, which makes it position almost optimal at an initial state. Therefore, it is required the definition of a trade-off between these two math options, average and standard deviation. This is in charge of the hyperparameter K.

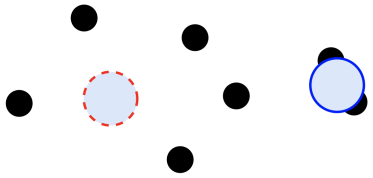


Figure 6.7: Example about agents' prioritization

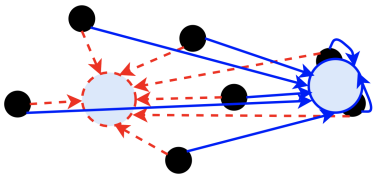


Figure 6.8: Distances calculation in this special case

This priority relies on the possibility to choose first which users they must be associated with, being those users out of the next agent's choices until no more users left. The number of users associated with each agent is defined through a cumulative distribution function build from the agents' array of distances to each user, *getConnectedUsers*(), following the Equation 6.7:

$$F(x) = \int_a^b f(x)dx \tag{6.7}$$

where $F(x)$ is the probability of $a \leq x \leq b$ and $f(x)$ represents the probability density function:

$$f(x) = \exp(-x),\ \forall x \geq 0 \tag{6.8}$$

where x corresponds to the distance between the current agent and the user. Figure 6.9 shows both these functions considering an example where the input data (x) seems to have a higher frequency in lower values.

This distribution adds a stochastic property to the choice of the users connected to each agent, removing an error propagation. The level of confidence on which users should be connected to the agent decreases as the number of minimums returned from the list of distances increases, hence, this distribution is build according to this list, being the connection probability related to the calculated distance.

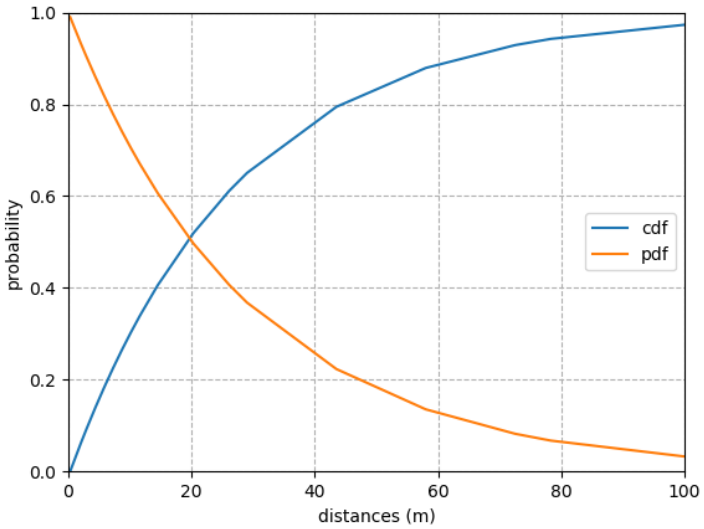


Figure 6.9: Cumulative Distribution Function (cdf) and Probability Density Function (pdf) used to define the number users to be connected to a specific agent, according to their distances

The random exponential number, calculated regarding this probability, defines the agent's maximum reach. Thus, all the users that confine a distance to the agent lower than this value will connect to it. In order to avoid the case in which the return of the cumulative distribution function would be a number lower than the minimum distance present in the array, leading to the waste of an agent's coverage, it is introduced an offset, precisely that minimum distance, ensuring that at least the nearest user to the agent connects to it.

Furthermore, it is fundamental to verify if the number of users to be connected exceed the already mentioned percentage of agent's capacity, given by $C \cdot (1 - \zeta)$. A positive answer makes the algorithm increase the *scale* parameter of the next agent's cumulative distribution function, which is a special sort of numerical parameter of a parametric family of probability distributions. Hence, the larger the scale parameter, the more spread out the distribution, as shown in Figure 6.10.

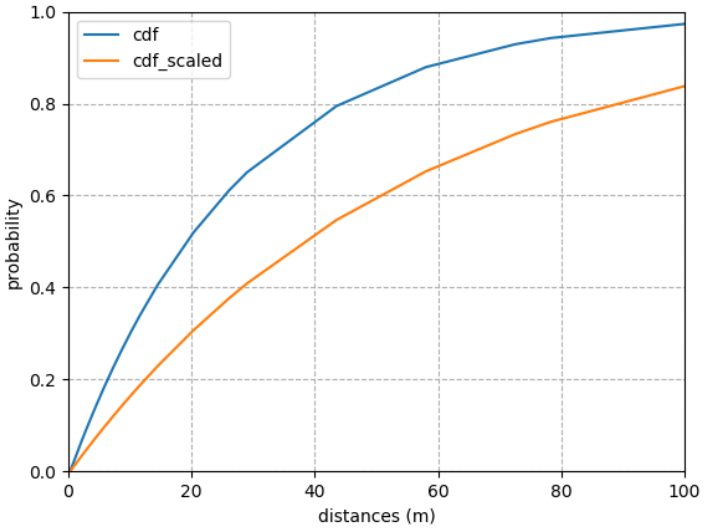


Figure 6.10: Impact of the scale in the slope of the cdf graphic

The increment in this *scale* attribute of the function consists in reducing the slope of the line, thus increasing the probability of choosing larger distances, which in turn increases the number of connections. This can be shown in the Figure 6.10. The value of this increment is defined by a hyperparameter, ξ, and the number of agents, N, as represented in Equation 6.9.

$$\Delta\beta = \frac{\xi}{N} \tag{6.9}$$

Having the knowledge of which users are connected to the current agent, the next step is to calculate the new optimal position where the agent should be, using a method similar to the *cooperative reward function* algorithm (4). The reward value will decrease with the distance between the current agent and the connected users. Therefore, the algorithm will try to find the optimal position of the agent where that distance is minimized, decreasing the absolute reward value.

Note that as the agents move this calculation gets new and different results, which makes the new so-far optimal position of the agent change throughout each iteration, providing "confusing" and incoherent indications to each agent. Therefore, it is introduced a connection update which will happen according to a percentage of the *episode_length* in order to update of the connected users and, consequently, the so-far optimal position. This percentage will be proportional to the acceleration of users' movements, i.e., if they move faster more changes to the environment occur at each time step, so the updates will need to be more frequent.

6.5.2 Neural Networks

Neural networks are introduced to this algorithm given the need to calculate the state-action value function used to find the so-far optimal policy, in order to determine the best action to be taken by a specific agent.

MLP is one of the basic building blocks of artificial neural networks. It is a feed-forward structure commonly used in back propagation algorithms to increase the test accuracy [69] [70] as needed when the goal is to recursively determine the state-value function and achieve the optimal policy. Before diving into a deep explanation of this kind of models, it is necessary to clarify the perceptron concept. In general terms, it is a linear function, which given an input x, will produce an output based on some internal parameters. Thus, mathematically, a perceptron is an implementation of the linear function given by:

$$f(x) = w \cdot x + b \tag{6.10}$$

where w corresponds to the weight and b to the bias. However, unlike regular functions, a perceptron can learn the optimal values for w and b, which is made by minimizing the average output error for a set of right example pairs $(x,\ f(x))$.

When talking about an MLP, as the name suggests, this concept is extended to a multi-layer network, that is build from single perceptrons (neuron) organized in layers. Accordingly, assuming N as the size of the input layer and M the size of the first hidden layers, $W = \{\{w_{11},\ldots,w_{1n}\},\ldots,\{w_{m1},\ldots,w_{mn}\}\}$ is a matrix $m \times n$ and $b = \{b_1,\ldots,b_m\}$ is a vector of size m:

$$f(x) = W \cdot x + b = \begin{bmatrix} w_{11} \cdot x_{11} + w_{12} \cdot x_{12} + \ldots + w_{1n} \cdot x_{1n} + b_1 \\ \ldots \\ w_{m1} \cdot x_{m1} + w_{m2} \cdot x_{m2} + \ldots + w_{mn} \cdot x_{mn} + b_m \end{bmatrix} \tag{6.11}$$

The weights and bias parameters are updated over each iteration during back propagation in agreement with the type of optimizer used to produce slightly better and faster results. An optimization algorithm aims to maximize/minimize a specific objective function, in this case, that function is referred as the loss function expressed in Equation 6.4, so the goal is to minimize it. As mentioned above, this algorithm uses the **Adam optimizer** due to its attractive benefits on non-convex optimization problems described in [68]:

- straightforward to implement;
- computationally efficient;
- little memory requirements;
- invariant to diagonal rescaling of the gradients;
- well suited for problems that are large in terms of data and/or parameters;
- appropriate for non-stationary objectives;

- appropriate for problems with very noisy/or sparse gradients;
- hyper-parameters have intuitive interpretation and typically require little tuning.

One of the configuration parameters defined in this optimization algorithm is the learning rate.

Having several layers for a perceptron enables the approximation to non-linear functions. However, simply adding new layers leads to the creation of new linear functions. Therefore, the activation functions are used, which introduce the desired non-linearity to the system. Examples of some of the most used activation functions are *tanh*, *sigmoid*, Rectified Linear Unit (ReLU) and Leaky ReLU.

Both *tanh* and *sigmoid* are sigmoidal functions (have a 's' shape), but the first one has a more extensive output, from -1 to 1, while the second maps the input values to output values between 0 and 1. The advantage of *tanh* is that the negative inputs will be mapped to even more negative output values.

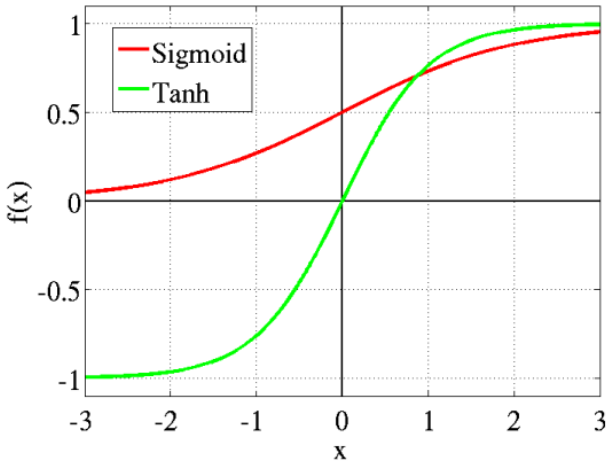


Figure 6.11: Comparison between *tanh* and *sigmoid* functions [9]

While *sigmoid* is a very common used differentiable and monotonic activation function, ReLU appears as a faster learner and a common approach nowadays, since it simply implements a *max* operation between 0 and x.

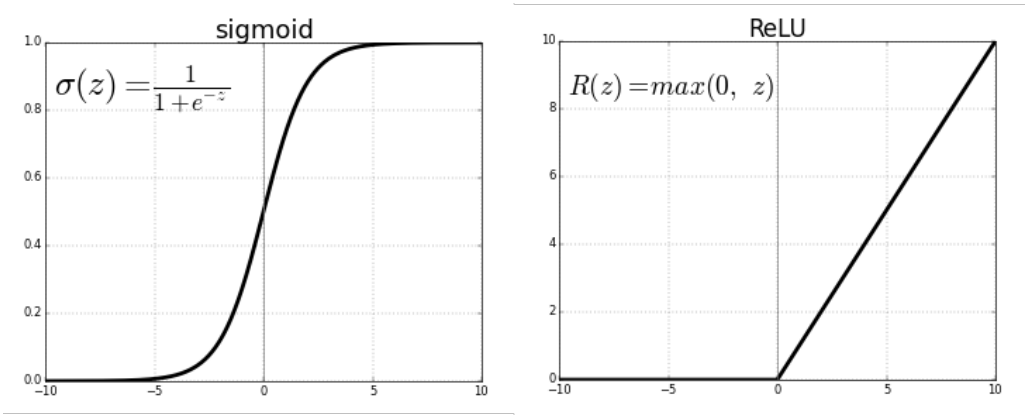


Figure 6.12: Comparison between *sigmoid* and ReLU functions [9]

Besides that, ReLU emerges as a solution for the vanishing gradient problem, where the problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. Considering the worst case, this may completely stop the neural network from further training. As presented in Figure 6.13, ReLU shows a linear derivative that prevents these cases from happening, while the other methods have an increasingly smaller slope as their value is mapped to extreme values. Traditional activation functions such as the *sigmoid* functions have gradients between 0 and 1, and backpropagation computes gradients by the chain rule. This results in multiplying n of these small numbers to compute gradients of the first layers in an n-layer network, meaning that the gradient (error signal) decays exponentially with n while the front layers train very slowly.

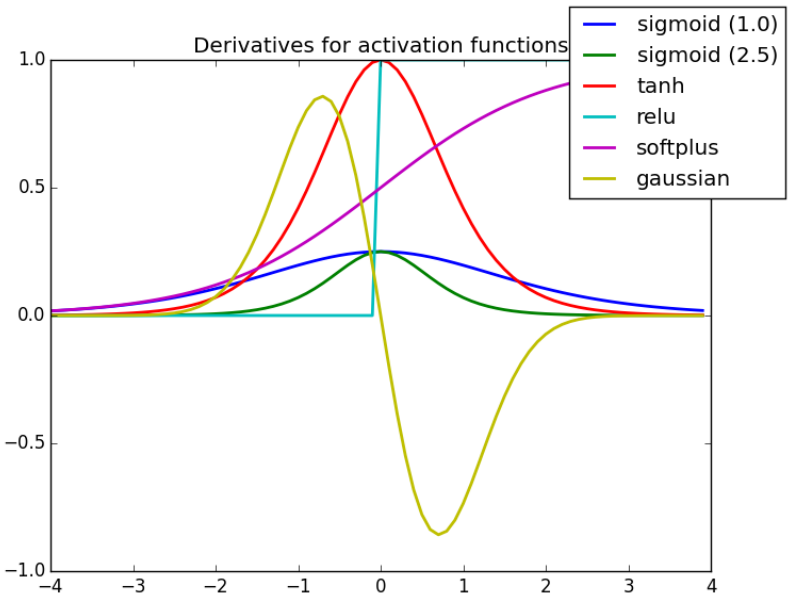


Figure 6.13: Derivative of Some Activation Functions [9]

The Leaky and Parametric ReLU approach is similar to the ReLU, but instead of mapping the negative values to zero, it have a small negative slope (defined as $\alpha = 0.01$ by default) as presented in Figure 6.14 and defined in the following equation:

$$f(x) = \begin{cases} \alpha x, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases} \tag{6.12}$$

This prevents the fully connected layer from the "dying ReLU" problem: neurons can sometimes be driven into states in which they become inactive for basically all inputs [71]. This problem affects the resulting graph by not mapping the negative values appropriately. Therefore, Leaky ReLU reveals to be a good approach, fast and with a range of $[-\infty, \infty]$. When the α is a random number, it is called Randomized ReLU, but that is out of the scope of this dissertation.

Exponential linear Units (ELUs) arrive as an enhancement of ReLU, being similar to Leaky ReLUs but with an important difference: rather than using a linear component for negative values, they use the exponential term, $e^x - 1$. The major effects are presented in the range of the activation, that changed from $[-\infty, \infty]$ to $[-1, \infty]$. This spawns a self-regularizing tendency of the networks to prefer values that are either positive or have a small magnitude in what concerns the usage of this activation function [72].

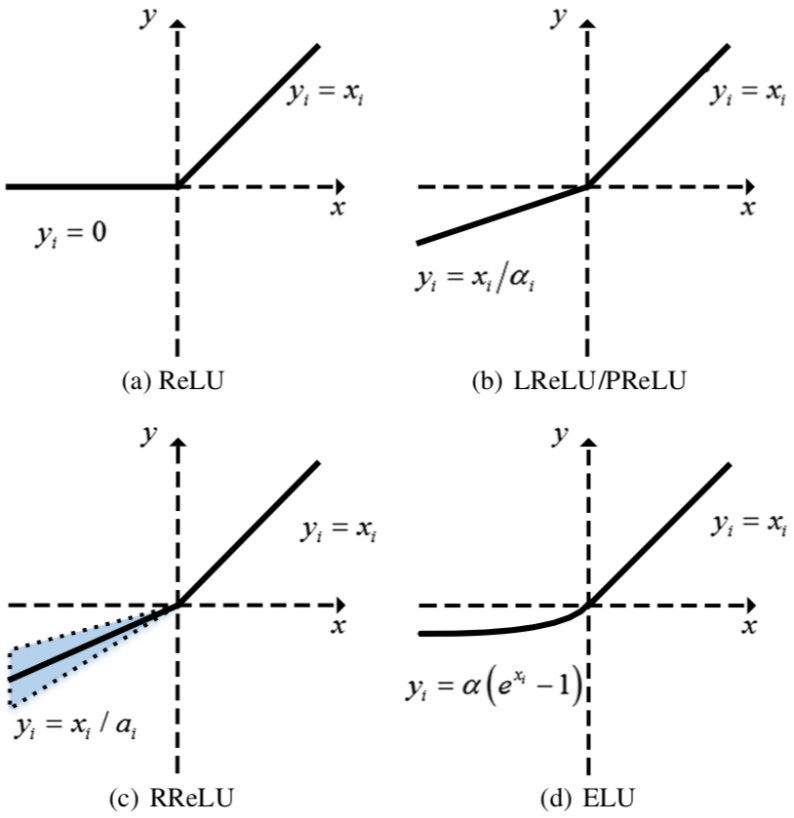


Figure 6.14: Comparison between the different types of ReLU functions [10]

The usage of an MLP neural network aims to extract important features of the network in order to get a more accurate output probability, customized for the network and appropriate for the ultimate goal, resulting in an approximation of the optimal point as the training phase proceeds.

Two different neural networks were used in this algorithm implementation: one to get the state-action value function given the actions and observation information of the environment (which will be named as **Q value model**), and another created to determine the best action to be taken, considering the observations of a specific agent and relating them to a probabilistic vector mapped to each one of the action space's elements (which will be named as **policy model**).

The size of the first neural networks considers the following parameters:

$$\begin{aligned} input_{neurons} &= N \cdot (4N + 2M) + N \cdot |\mathcal{A}| \\ output_{neurons} &= 1 \end{aligned} \tag{6.13}$$

where the number 4 corresponds to the 4 coordinates that describe the agents movement - (x, y) for the position and (x, y) for the velocity - and 2 corresponds to the users' positioning, (x, y).

Note that although the users are not static, their movements are completely random, not conferring any additional information to the neural network. The single output neuron corresponds to the state-action value function $Q_{\mu_\theta}(X, A)$, considering the deterministic policy μ_θ made regarding the actual parameters θ and the already mentioned set of observations X and actions A. Remember that N is the number of agents, M is the number of users and $|\mathcal{A}|$ is the size of the action space.

The size of the second neural networks follow the following parameters:

$$\begin{aligned} input_{neurons} &= 4N + 2M \\ output_{neurons} &= |\mathcal{A}| \end{aligned} \tag{6.14}$$

This MLP model returns a set of agent probabilities to take one of the actions defined by the action space, according to the agent's observations of the environment. The output result of this model takes the probability of executing a specific action in the discrete domain and makes a vector addition of the results to obtain the final action to be taken considering a continuous space.

After some tests detailed in the next chapter, it was concluded that the MLP models implemented in this algorithm should use a **ELU** as the activation function of the first fully connected layer and a **ReLU** for the second one.

Since the output values of the policy model should appear as a probability, after using the ReLU it is necessary to apply a ***Softmax* function** in the output values. This corresponds to a normalized exponential function which "squashes" a K-dimensional vector z with arbitrary real values to a K-dimensional vector $\sigma(z)$ composed by real values

between 0 and 1 that add up to 1, following the equation [71]:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}, i = 1, 2, ..., K \tag{6.15}$$

Both MLP models are build with two hidden layers, however each model has specific settings. While the Q value model needs to have a bigger number of neurons in each hidden layer (approximately twice the size according to equations 6.13 and 6.14 and since it is only defined by powers of 2), the policy model demands the application of a *softmax* function to their output values. These and other main characteristics of both models are presented in the figures 6.15 and 6.16.

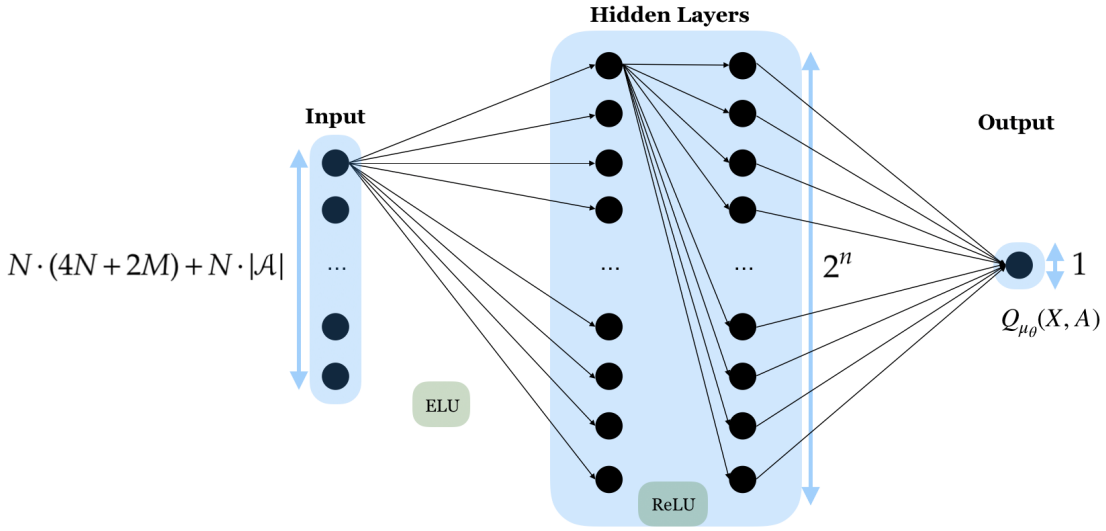


Figure 6.15: Multi-layer Perceptron Model for Q Value updates

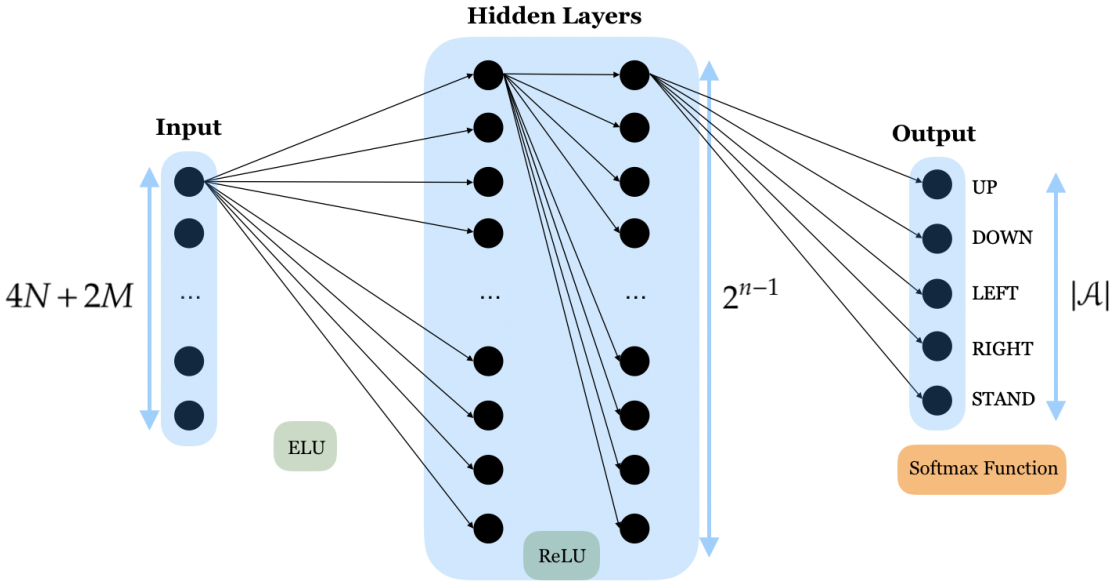


Figure 6.16: Multi-layer Perceptron Model for Policy updates

6.6 Tools Used

Due to the numerous and recent investigations related to this area, there is a great range of tools that can be used to create a RL algorithm. However, Tensorflow and OpenAI Gym were the main used tool to train the algorithm.

Tensorflow is a well-known open-source framework created by Google for many programming languages. It has more features than Keras, being not so easy to use, but presenting more flexibility and control of the network.

Open AI Gym is one of the most popular environments for developing and comparing reinforcement learning models. Besides that, it is compatible with high computational frameworks like TensorFlow.

Besides these tools, **Anaconda** was also used. It is an open-source distribution of Python that includes **Spyder**, a powerful scientific environment for Python which was very useful for data visualization, exploration, debugging and interactive execution. Moreover, Spyder proffers built-in integration with several popular scientific packages, including SciPy, NumPy, Pandas, Matplotlib, IPython, QtConsole, SymPy, and more.

Chapter 7

Results

This chapter starts with the explanation of some network concepts and their relations in order to map the results obtained and presented in the successive sections with the QoS parameters used to evaluate the flying network. It is demonstrated that those QoS parameters, namely the throughput, decrease with increasing distance, allowing the use of the distance as criterion for the manipulation of the reward function.

7.1 Relation between Reward Function and QoS

The deployment of the Cooperative Mesh MADDPG algorithm in a TMFN architecture should take into consideration the relation of its reward function and the wireless technologies used nowadays.

It will be considered the recent IEEE 802.11ax, also known as Wi-Fi 6, which has brought numerous improvements over the previously and very commonly used nowadays IEEE 802.11ac. Besides the use of Orthogonal Frequency-Division Multiple Access (OFDMA) uplink and downlink and the modulation reaching the 1024-QAM, another technical improvement was the extension of the guard intervals duration that allow for better protection against signal delay spread, which is common in outdoor environments, as a TCE. Besides that, with this specification of IEEE 802.11 it is now available the Multi-User Multiple Input Multiple Output (MU-MIMO) in both downlink and uplink direction. This technology confers directional beams to multiple simultaneous users as shown in Figure 7.2 instead of what is present in Figure 7.1.

The **Friis transmission equation** describes how the received Power, Pr, changes with distance, d:

$$P_r(d) = \frac{P_t G_t}{4\pi d^2} \frac{G_r \lambda^2}{4\pi} = G_t G_r \left(\frac{\lambda}{4\pi d}\right)^2 P_t \text{ [W]} \tag{7.1}$$

where G_t and G_r are the gain of the transmitter and receiver antenna respectively, and λ is the wavelength associated to the aperture (area) of receiving antenna.

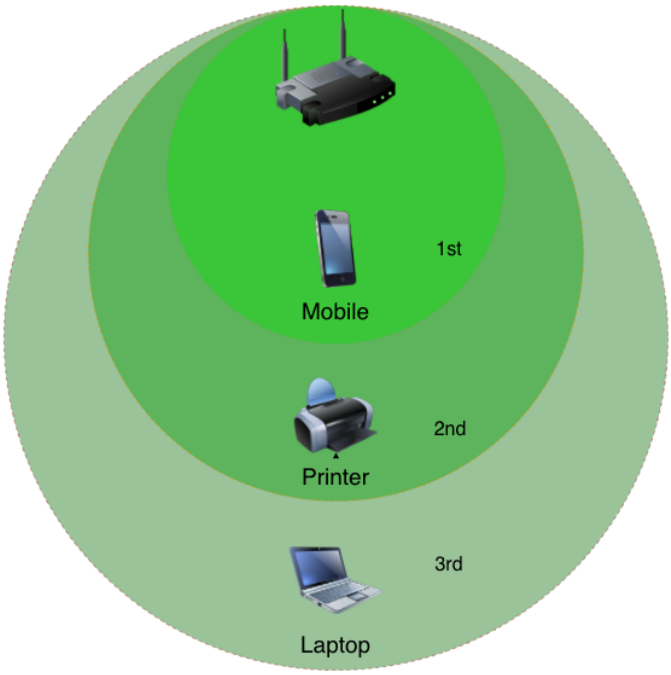


Figure 7.1: Single-User MIMO where Wi-Fi connects to one device at a time

Considering that the UAVs will be at a considerable altitude in a free space and that Line-Of-Sight (LOS) can be assumed when considering a flying network architecture, the propagation model can be represented as a **Free Space Path Loss model**.

Assuming the presence of an Additive White Gaussian Noise (AWGN) the capacity of a wireless channel is given by the **Shannon theorem**:

$$C = B \cdot log_2(1+\gamma) \text{ [bit/s]} \tag{7.2}$$

where B is the bandwidth and γ is the SNIR, which can be expressed as:

$$\gamma = SNIR = \frac{P_r}{N_0 B + \sum_{i=1}^{I} P_{r_i}} \tag{7.3}$$

being N_0 the noise power spectral density and $\sum_{i=1}^{I} P_{r_i}$ the power received from interfering nodes.

Assuming that each Wi-Fi AP carried by the UAV is configurable with four spatial streams in a 16-QAM modulation with a coding rate of 3/4, the bit rate given by each directional beam will be dependent on the channel's frequency and the guard intervals duration, which will be defined as $20MHz$ and $1600ns$, respectively. Therefore, and considering the standard definitions of IEEE 802.11ax, the data rate of each AP will be of approximately $50Mb/s$.

Since the goal of this problem solution is to maximize the QoS, it is necessary to assure that the consideration of the distance between the users and the agents is in agreement

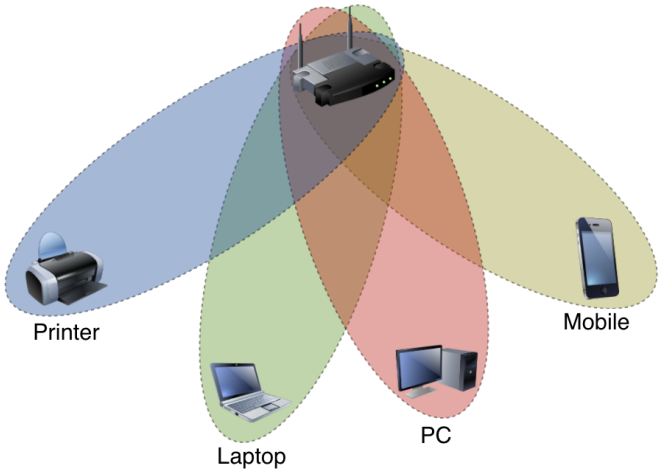


Figure 7.2: Multi-User MIMO to improve capacity where Wi-Fi connects to multiple devices once, at the same speed

with the Key Quality Indicators or QoS parameters. The three most important parameters that design the QoS evaluation are the packet delay, packet loss ratio and throughput.

All these three topics are interconnected and related to the capacity, C, since it is the tight upper bound on the rate at which information can be reliably transmitted over a communication channel. So, mixing up all the previous equations (7.1, 7.2 and 7.3) and discarding the interferences:

$$\begin{aligned} C = B \cdot log_2\left(1 + \frac{G_t G_r \left(\frac{\lambda}{4\pi d}\right)^2 P_t}{N_0 B}\right) \quad &\Leftrightarrow \\ \Leftrightarrow C = B \cdot log_2\left(1 + \frac{G_t G_r \frac{\lambda^2}{16\pi^2 d^2} P_t}{N_0 B}\right) \quad &\Leftrightarrow \\ \Leftrightarrow C = B \cdot log_2\left(1 + \frac{G_t G_r \lambda^2 P_t}{16\pi^2 d^2 N_0 B}\right) & \end{aligned} \tag{7.4}$$

According to Equation 7.4 and assuming that B, G_t, G_r, P_t and N_0 are non-negative constants and that log_2 is a monotonically increasing function, if the value returned by the fraction inside log_2 increases, C increases as well. This fraction is inversely proportional to the square of the distance, hence, the capacity will rapidly decrease with increasing distance. □

Figure 7.3 proves the previous considerations according to the following parameters:

- Specification of IEEE 802.11: ax
- Modulation and Coding Scheme (MCS) index: 4
- Modulation: 16-QAM
- Coding Rate: 3/4

- Bandwidth: 20 MHz
- Guard Intervals Duration: 1600 ns
- Data Rate: 50 $Mbit/s$
- RX antenna gain: $G_r = 3\ dBi$
- TX antenna gain: $G_t = 3\ dBi$
- Noise Power Spectral Density: $N0_{dBm} = -174\ dBm/Hz$
- Noise Figures: $NF = 0$
- Power Received from Interfering Nodes: $\sum_{i=1}^{I} P_{r_i} = 0$
- Packet Size: $MTU = 1000\ bits$
- Power Transmitted: $P_t = 20\ dbm$

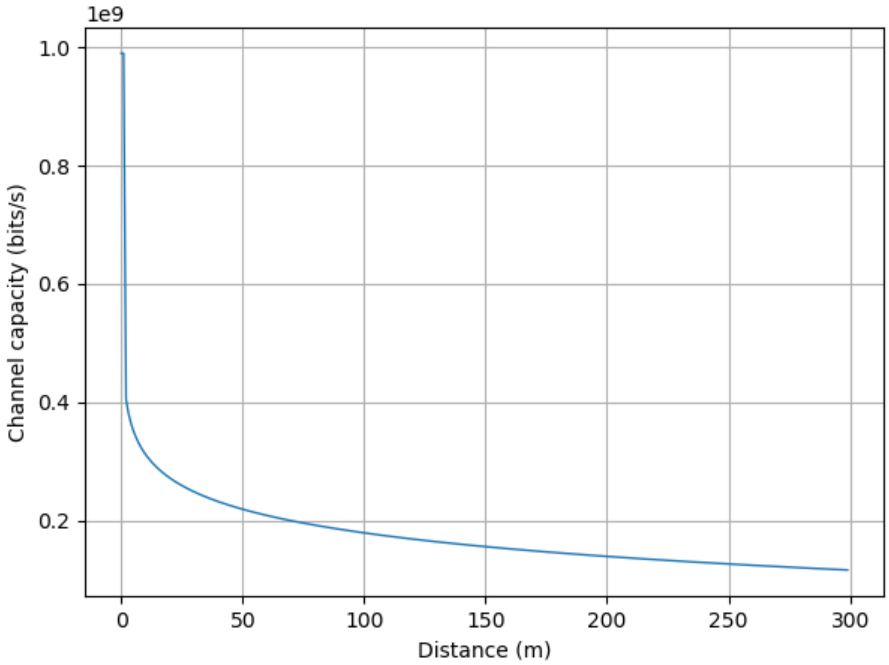


Figure 7.3: Channel Capacity

7.2 Test Cases

Considering the problem stated in Section 1.3 and the absence of an ML algorithm to manage the UAV topology according to the users disposal in the TCE, the main solutions pass through an *a priori* determination of a fixed point to place the UAVs, which is assumed

as the strategic point given by the geometrical center of the environment, or the control of their positioning manually, an approach that will not be considered since the goal is to automatically find the optimal solution, i.e., nowadays it is desired to reduce the real-time human intervention to solve optimization problems.

Therefore, in order to evaluate the Cooperative Mesh MADDPG algorithm, it is defined a *baseline* approach to solve this problem, which is described by the fixed and geometrical positioning of the UAVs as shown in the Figure 7.4.

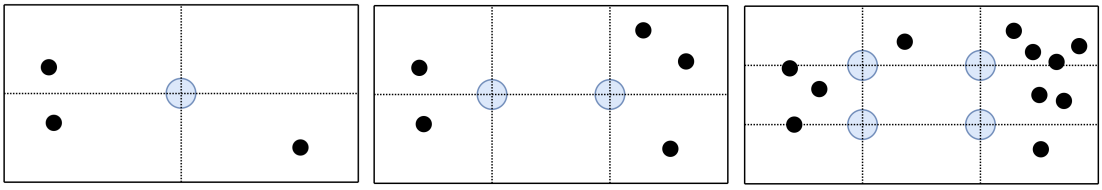


Figure 7.4: Example of the *baseline* positioning for one, two and four agents (UAVs)

Besides this *baseline* approach and, as discussed in Section 6.5.1, there are two more approaches that aim to find the best UAV topology using an RL algorithm:

1) the *cooperative reward function algorithm*, which considers the minimum distance between the user and each agent (4)

2) the *link state reward function algorithm*, which takes into account the overloaded channel and the UAVs connections with the users (5)

In addition to these three approaches were defined two scenarios, all with an agent capacity, C, of 4 users, following the Equation 6.5:

A) **single-agent**: 1 agent and 3 users

B) **multi-agent**: 2 agents and 5 users

Note that the *baseline* approach is not a DRL algorithm, hence it cannot be directly compared to the other two approaches. The other two approaches differ in the reward function calculation which is made with different algorithms (*cooperative* vs *link state*), but both used to train the DRL algorithm. Each of these methods defines a way to calculate their performance according to the users' disposal versus the UAV positioning:

- *baseline*: sum of the distances between each user and the nearest agent - defined as *standard metric*
- *cooperative*: *standard metric* plus the negative reward of -1 given when agents collide during the training phase.
- *link state*: difference between the agent's current position and the estimated optimal position obtained by the average positioning of the connected users. This metric also counts the agents collision penalization as the *cooperative* approach does it.

This is the only way to know if the user can associate himself to the agent or if the agent is "full", which already points out its advantage facing the other approaches, not leading the learning process to the overload of the APs.

Another metric is defined to correctly evaluate the QoS performance assuming the relation between the reward function and the QoS detailed in Section 7.1. The *ideal metric* consists of the sum of the distances between each agent and the connected users plus the agents collision penalization. This metric is called ideal because it avoids considering the sum of distances between users and agents who, despite being closer, may already be full of capacity. Contrariwise, the *standard metric* does not take into account that cases where the Wi-Fi AP carried by the UAV is overloaded and the user cannot be associated to the nearest agent, i.e., it considers an unlimited agent capacity.

To evaluate these three approaches it is necessary to find the best way to compare them. The performance of the *baseline* approach is estimated separately of the other two, which run simultaneously as the training process proceeds. While the two algorithms take into consideration the movable position of the UAVs as the DRL algorithm receives its rewards, the *baseline* approach uses the fixed positions of the UAVs. All three methods use the random users' disposal on the field.

Therefore, since the *baseline* approach cannot know which users the other agents connect with, unlike the other two algorithms, this method is evaluated using the *standard metric* as mentioned above. The other two algorithms can be evaluated by any of the two metrics, but the preference is to use the *ideal metric*.

Concluding, in the next section the three approaches will be evaluated using the *standard metric* and, more accurately, the two algorithms will be evaluated using the *ideal metric*. The impossibility of evaluating the quality of the *baseline* approach with the *ideal metric* already demonstrates a restriction that gives advantage to the two other approaches, this is, to the use of a DRL algorithm.

7.3 Algorithm Parameters

Before diving into the results exploration, it is necessary to indicate which are the values defined for the multiple constants and hyperparameters. During the evaluation process, where the following parameters were changed multiple times, the best-found combination in agreement with both scenarios was:

- percentage of redundant channel occupancy avoiding AP saturation: $\zeta = 0.1$
- learning rate for Adam optimizer: $\eta = 0.01$
- discount factor: $\gamma = 0.95$
- weight given to the average distance to users when prioritizing agents: $K = 0.5$

- maximum total incremental value to the scale parameter distributed by all agents to increase users connection probability: $\xi = 0.1$
- rate for updating the target network (θ): $\tau = 0.01$
- agents collision penalty: $collision\_factor = 10$
- percentage of the collision area relative to the dimensions of each agent: $collision\_area = 1.2$
- number of episodes to optimize at the same time: $batch\_size = 512$

Besides these parameters, the size of the training process was defined as:

- total number of episodes: $total\_episodes = 100000$
- save rate of episodes: $save\_rate = 1000$
- maximum size of each episode: $episode\_length = 50$
- connection update according to the percentage of the $episode\_length$: $conn\_rate = 0.4$

Assuming that the number of units of the hidden layers should be at least bigger than the size of the input layer in order to improve the feature extraction property of the neural network, in Q value model it is calculated in line with:

$$num\_units\_q > N \cdot (4N + 2M) + N \cdot |\mathcal{A}| \tag{7.5}$$

which, as explained in Section 6.5.2, implies that the number of units of the hidden layer for the policy model be:

$$num\_units\_p = 2^{\log_2(num\_units\_q)-1} \tag{7.6}$$

For **scenario A**, where $N = 1$, $M = 3$ and $|\mathcal{A}| = 5$:

$$\begin{aligned} & num\_units\_q > N \cdot (4N + 2M) + N \cdot |\mathcal{A}| \quad \Leftrightarrow \\ \Leftrightarrow\ & num\_units\_q > 1 \cdot (4 \cdot 1 + 2 \cdot 3) + 1 \cdot 5 \quad \Leftrightarrow \\ \Leftrightarrow\ & num\_units\_q > 15 \end{aligned} \tag{7.7}$$

it was concluded that:

$$num\_units\_q = 16 \implies num\_units\_p = 8 \tag{7.8}$$

For **scenario B**, where $N = 2$, $M = 5$ and $|\mathcal{A}| = 5$:

$$\begin{aligned} & num\_units\_q > N \cdot (4N + 2M) + N \cdot |\mathcal{A}| && \Leftrightarrow \\ \Leftrightarrow\ & num\_units\_q > 2 \cdot (4 \cdot 2 + 2 \cdot 5) + 2 \cdot 5 && \Leftrightarrow \\ \Leftrightarrow\ & num\_units\_q > 46 \implies num\_units\_q = 64 \end{aligned} \tag{7.9}$$

it was concluded that:

$$num\_units\_q = 64 \implies num\_units\_p = 32 \tag{7.10}$$

Note that the results presented in the next section will compare the three approaches (*baseline*, *cooperative* and *link state*) in the two scenarios (A and B) using both the *standard* and *ideal metrics*.

7.4 Results Analysis

As mentioned in Section 6.5.2, the choice of the ELU and ReLU as the activation functions for the two hidden layers of the MLP model respectively, is in agreement with the results presented in Figure 7.5.

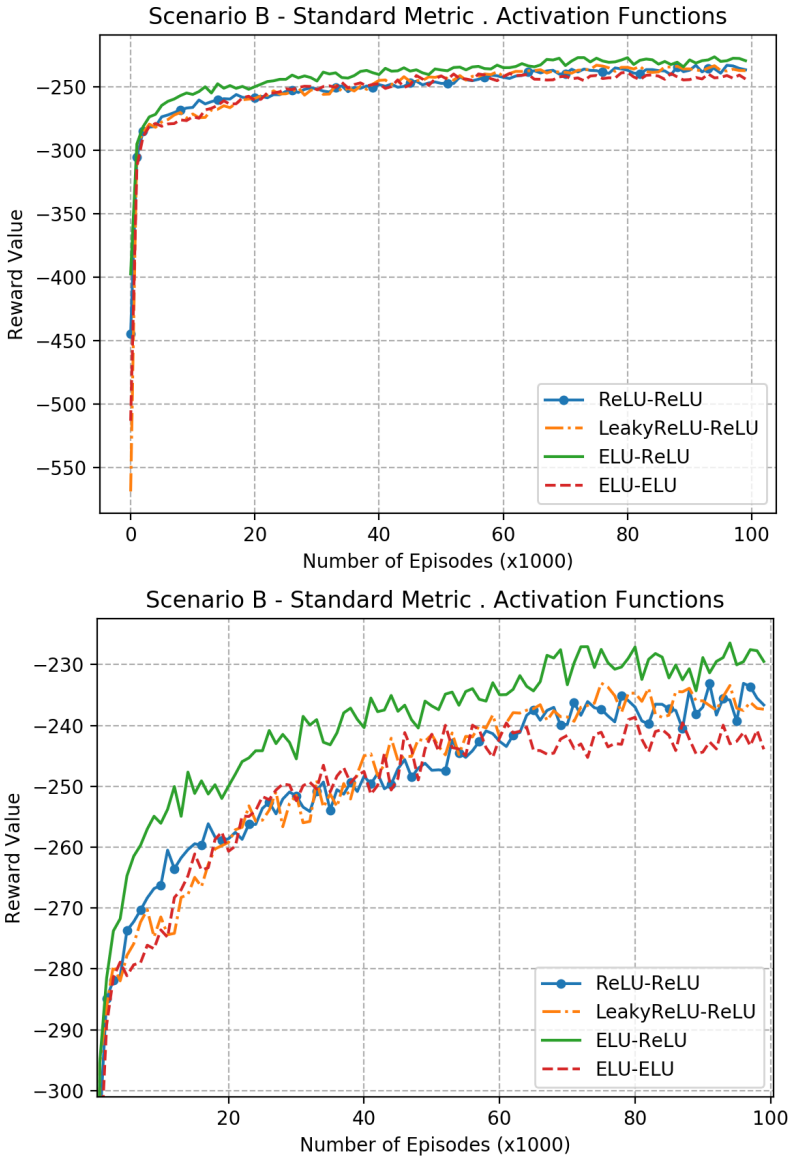


Figure 7.5: Rewards given by the *standard metric*, considering scenario B, the training process with the *link state reward function* algorithm and different activation functions, from an overview and zoom perspective, respectively

Combining these five conditions (three approaches and two scenarios) and assuming the characteristics of the algorithms described in Section 6.5, the following figures presents the final results:

- **Scenario A**:

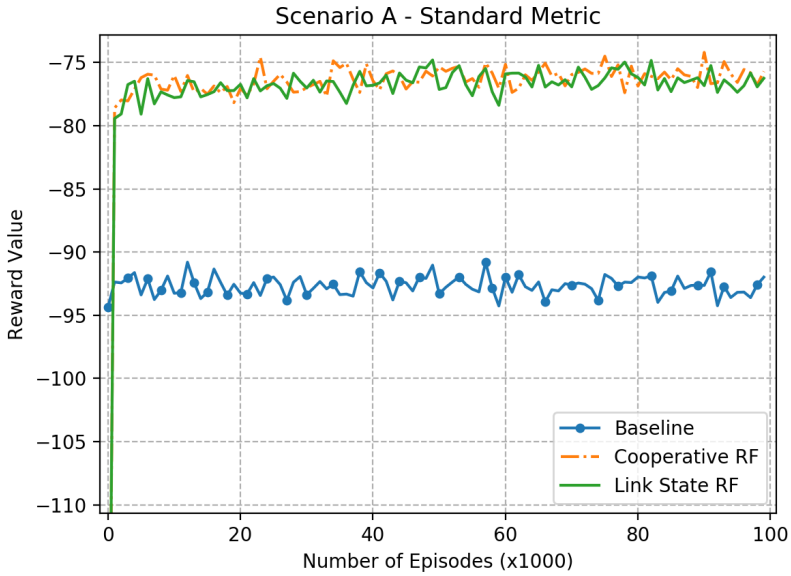


Figure 7.6: Rewards given by the *standard metric*, considering scenario A

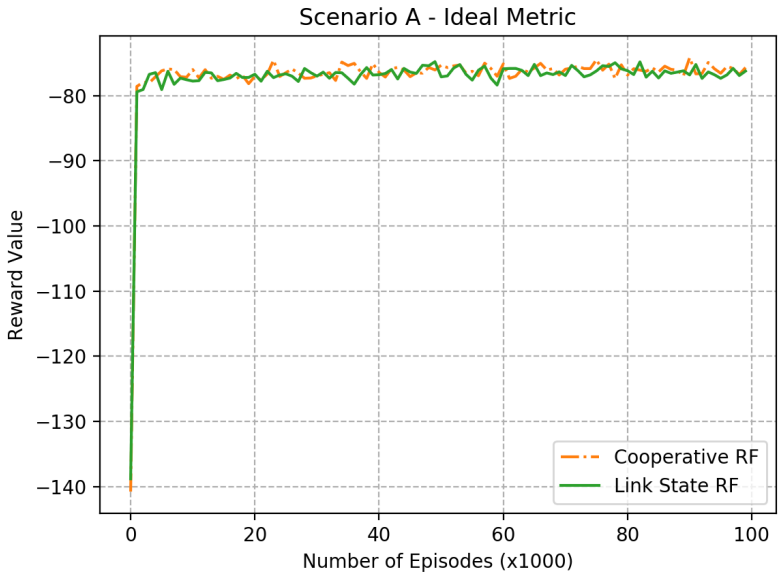


Figure 7.7: Rewards given by the *ideal metric*, considering scenario A

By the analysis of Figure 7.6 both algorithms show similar results. It is important to point out that the values presented in this figure are from two different training processes, one trained with the *cooperative reward function* and another with the *link state reward function*. Therefore, the curves shown in the graphic are not exactly the same, depending on a random set of variables - initial positioning of agents and user, plus users arbitrary movements. However, its similarity is sufficient for the

results analysis and to conclude that both DRL approaches present better results than the *baseline*.

In addition, this scenario is a special case since the great distinction between the two algorithms is not present here. *Link state reward function* algorithm (5) appears to solve the connection problem between the users and the agents, informing the algorithm in which way the users are distributed by the agents throughout each iteration in order to avoid situations of overload of the bandwidth of the Wi-Fi AP carried by the UAV. Therefore, since this scenario assumes that the single-agent can cover all users, it is expected that they have equal results according to both *standard* - distance between each user and its nearest agent - and *ideal metric* - distance between each user and its corresponding agent - as shown in Figure 7.7.

- **Scenario B**:

  Figure 7.8 remarks for the improvements achieved with each of the algorithms vis-à-vis the *baseline* approach. While the *baseline* is almost constant due to its strategic positioning, the other two curves tend to stabilize up to the minimum reward achieved. Note that the reward value can never be greater than 0, being this minimum only achieved in a scenario with an equal number of users and agents.

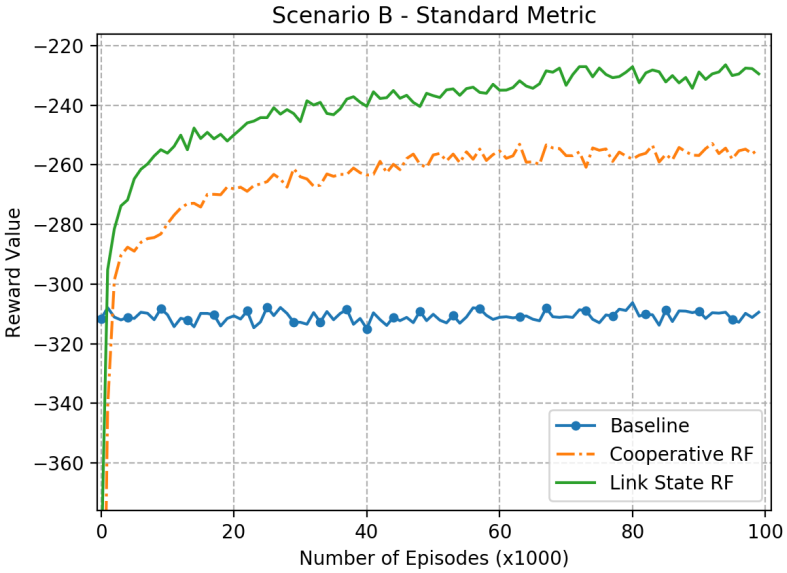


Figure 7.8: Rewards given by the *standard metric*, considering scenario B

Figure 7.9 reveals that, taking into account the *ideal metric*, the results obtained when training the DRL algorithm using *link state reward function* are better than the ones presented using the *cooperative reward function*. This is easily explained since the *link state* approach trains according to this metric, being more accurate when considering important and real aspects of flying network architecture implementations, such as the already mentioned access point overload.

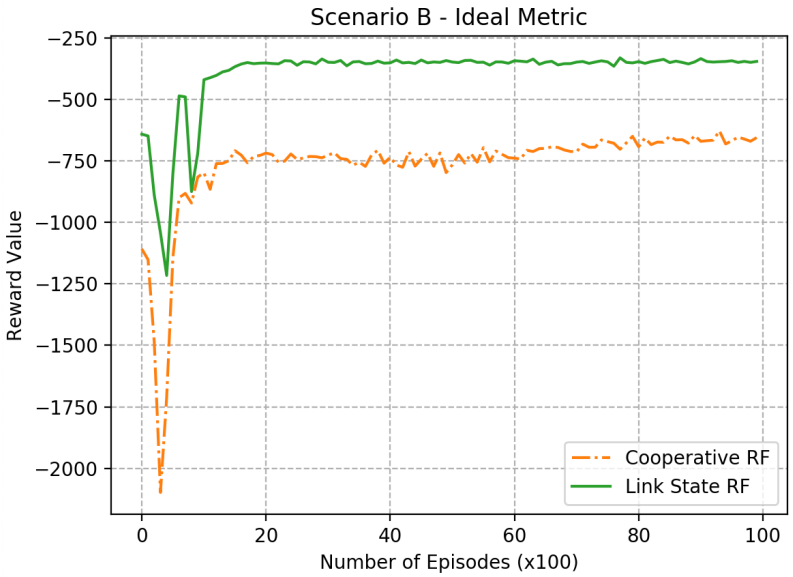


Figure 7.9: Rewards given by the *ideal metric*, considering scenario B

The results related to the *cooperative* approach can be analyzed through the Figure 7.10, while those concerning the *link state* approach are exhibited in Figure 7.11.

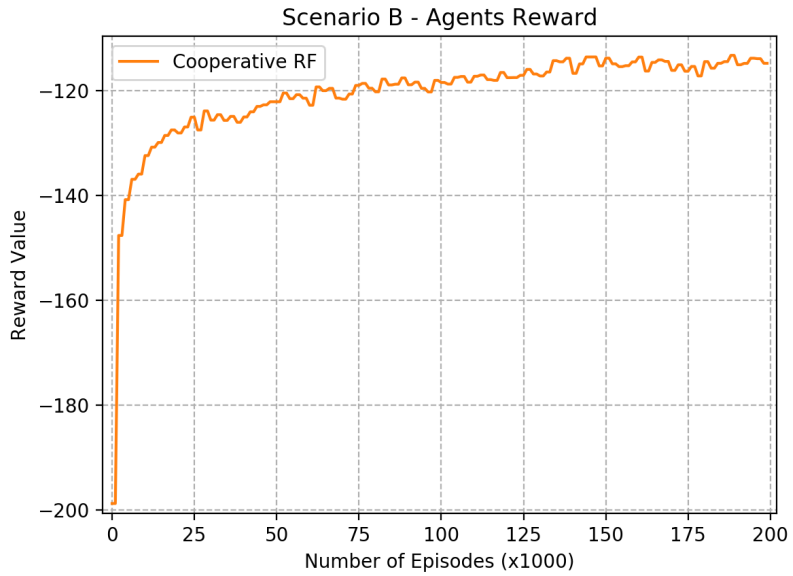


Figure 7.10: Rewards given by the metric used to train through the *cooperative reward function* of algorithm 4 (*standard metric*), considering scenario B

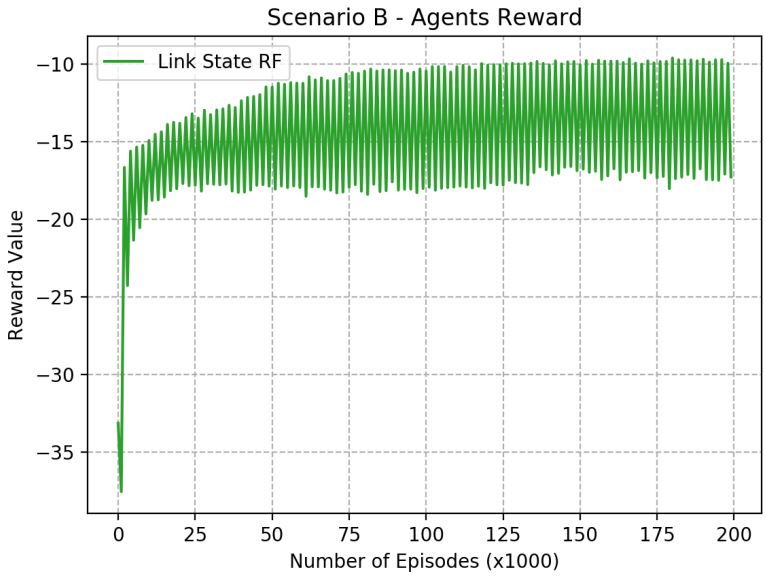


Figure 7.11: Rewards given by the metric used to train through the *link state reward function* of algorithm 5, considering scenario B

The cooperative and competitive component of these algorithms (*cooperative* approach and *link state* approach, respectively) can be analyzed through Figures 7.10 and 7.11. The first figure shows a stepped graph regarding to the shared rewards of the two agents, while in the second figure the reward are not shared, both agents receive different rewards. Thus, the return value of the *ideal metric* estimated in the *link state reward function* algorithm for each agent presents a frequent number of peaks in the stabilization zone, keeping its average monotonic, which indicates the cooperative work of the agents where both compete for the connection to users with view to a lower system reward, changing its reward value in a quasi-complementary way.

In addition, it is important to emphasize the impact of the experience replay in this algorithm, showing that the results are indeed related to the *batch_size* as observable by Figure 7.12. In this figure it is clear that the results improved after the optimization of the previous batch size number of episodes.

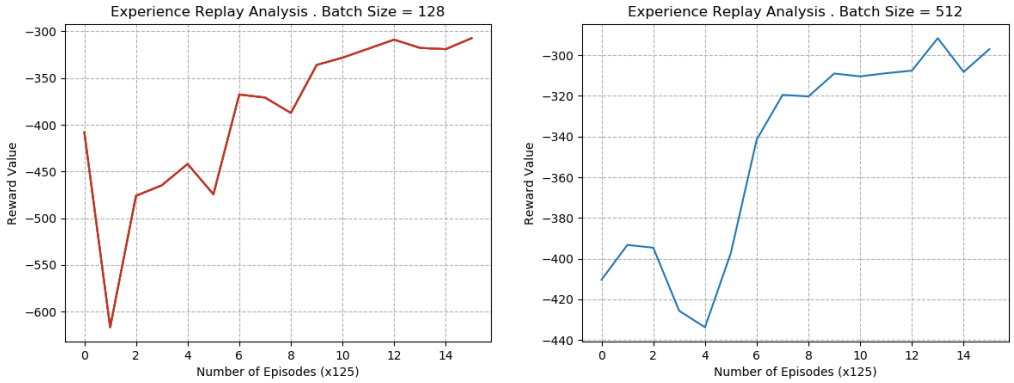


Figure 7.12: Results of a small training phase in scenario B, where the inflection points match the optimization of the *batch_size* number ($1 \cdot 125 \approx 128$ (left) and $4 \cdot 125 \approx 512$ (right)) of the previous episodes

Furthermore, the definition of the number of units used in the hidden layers is related to the starting point of the algorithm reward values as shown in Figure 7.13, which can be inferred from the fact that, theoretically, more units indicate a deeper analysis of the input data. However, the convergence is not very much affected by the number of units. The number of units has been chosen taken into account the trade off between the amount of time that an hidden layer with more units spends running the algorithm versus its not very significant benefits, as mentioned in Section 7.3.

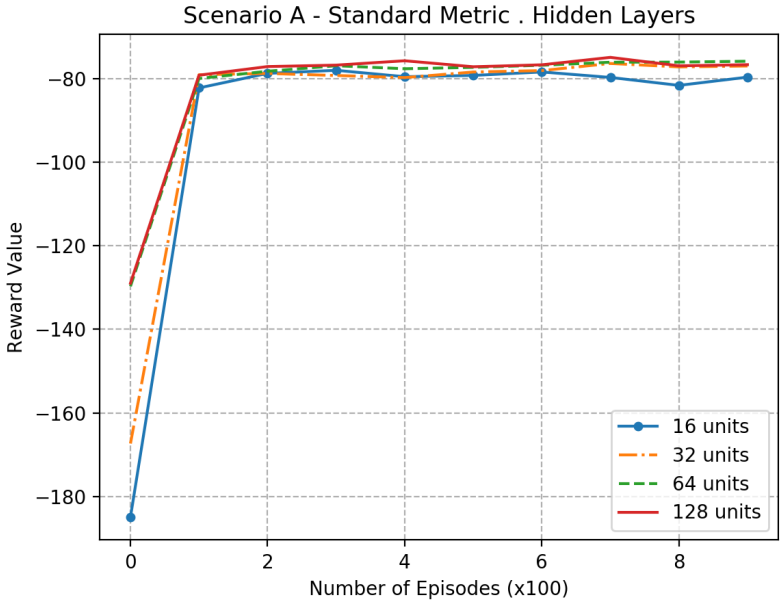


Figure 7.13: Rewards given by the *standard metric*, considering scenario A, a training process of 10 000 episodes with the *link state reward function* algorithm and different number of units, from an overview and zoom perspective, respectively

Chapter 8

Conclusion

The scope of this dissertation reveals to be extensive due to the lack of background in Machine Learning and Deep Learning that this master degree presents. Therefore, there was a need to explore several reinforcement learning techniques, both in their theoretical and practical components. During this research, a survey was carried out, aiming to bridge together the three main concepts of this work: reinforcement learning, solution optimization, and flying networks.

The main focus of this dissertation was the exploration of Deep Reinforcement Learning techniques, namely DQN, until its limitations to single-agent systems. Due to that premise, the policy gradient methods were introduced to answer the optimization problem of the airborne network topology management concerning a multi-agent POMDP system. DPG appear as one useful method which combined with DQN put together the two core pieces of the proposed Cooperative Mesh MADDPG algorithm.

The use case of this dissertation is defined according to a flying network architecture, assuming a TCE as its environment and UAVs as the agents, with a viewpoint of an RL algorithm. When managing multiple agents seeking the same goal, but acting isolated, the cooperation aspect can be challenging. This was one of the major tasks while developing the proposed UAV placement algorithm. Since the reward function has a crucial part in the learning process - "teaching" the algorithm which way it should follow throughout each iteration - its definition requires a complex algorithm able to solve that cooperation aspect. From the three presented approaches, the *link state reward function* algorithm is presented as the best solution to avoid an UAV overhead, pointing out to the optimal and instantaneous airborne network topology.

Neural networks were also explored and used in this dissertation, by the inherent use of an MADDPG algorithm as the baseline of the proposed algorithm. It was implemented an MLP neural network and studied the best parameters to define it, such as the usage of an Adam Optimizer and ELU and ReLU as the two successive activation functions of its hidden layers.

The results show that the objectives of this dissertation were successfully fulfilled, referring to the scientific contribution of a DRL algorithm for flying networks.

Firstly, comparing the obtained results with the *baseline* approach, it was concluded that the DRL proposed algorithm, which have two different implementation regarding the definition of the reward function, showed remarkable improvements. These improvements are, theoretically, even better when testing these approaches to more crowded scenarios and with a wider distribution of the users in larger environments, being the geometric points of the mesh design by the *baseline* approach more spaced and presenting worse results. In addition, the *link state reward function* algorithm demonstrates how to bypass problems arisen from the implementation of multi-agent systems in network environments, such as the overhead of UAVs, presenting a solution that faces algorithms as Monte Carlo K-means.

Secondly, comparing the achieved results with the state of the art, this dissertation introduced a new answer to the stated network problem composed of DRL and policy gradient methods. The usage of GMM and WEM in a similar problem, as reported in the state of the art, also presents very promising results, but it is not considered as a deep reinforcement learning technique. While the solution proposed in this dissertation considers completely random users' distributions and movements in a continuous task scenario, the other referred algorithm uses a cellular traffic distribution dataset to train the model and evaluate the performance, making assumptions that restrict its implementation to completely dynamic scenarios learned in real-time, not being able to precisely compare the results of the two algorithms.

The mapping of the metric used in the reward function to the commonly used QoS parameters (packet delay, packet delivery ratio and throughput), allows to theoretically prove the applicability of this model in the networking field and the potential improvements against the *baseline*.

The dimension of the scientific content involved in the creation of this algorithm induces that the results can be further analyzed. Some of these potential improvements are:

- further analysis of the parameters referred in Section 7.3 and exploration of the potentiality of the algorithm increasing the number of test cases;
- simulate the proposed algorithm in ns-3 taking advantage of its integration with Open AI Gym. This enables the usage of more realistic data and the QoS parameters as directly inputs to shape the reward function.

References

[1] Lex Fridman. Mit course, 6.s191: Introduction to deep learning 2018. http://introtodeeplearning.com, 2018. Accessed 2018.

[2] P. V. Klaine, M. A. Imran, O. Onireti, and R. D. Souza. A Survey of Machine Learning Techniques Applied to Self-Organizing Cellular Networks. *IEEE Communications Surveys Tutorials*, 19(4):2392–2431, 2017. doi:10.1109/COMST.2017.2727878.

[3] Armando Arroyo GeekStyle. Business intelligence and its relationship with the big data, data analytics and data science. https://www.linkedin.com/pulse/business-intelligence-its-relationship-big-data-geekstyle/, 2017. Accessed 2018.

[4] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018. arXiv: 1811.12560. URL: http://arxiv.org/abs/1811.12560, doi:10.1561/2200000071.

[5] David Silver. Ucl course on reinforcement learning. http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html, 2015. Accessed 2018.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[7] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A Brief Survey of Deep Reinforcement Learning. *IEEE SIGNAL PROCESSING MAGAZINE*, page 16, 2017.

[8] Udacity. Udacity course of reinforcement learning. https://eu.udacity.com/course/reinforcement-learning--ud600, 2018. Accessed 2018.

[9] SAGAR SHARMA. Activation Functions in Neural Networks, September 2017. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[10] Huizhen Zhao, Fuxian Liu, Longyue Li, and Chang Luo. A novel softplus linear unit for deep convolutional neural networks. *Applied Intelligence*, 48(7):1707–1720, July 2018. URL: http://link.springer.com/10.1007/s10489-017-1028-7, doi:10.1007/s10489-017-1028-7.

[11] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78, October 2012. URL: http://dl.acm.org/citation.cfm?doid=2347736.2347755, doi:10.1145/2347736.2347755.

[12] E. N. Almeida, R. Campos, and M. Ricardo. Traffic-aware multi-tier flying network: Network planning for throughput improvement. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6, April 2018. doi:10.1109/WCNC.2018.8377408.

[13] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network*, 32(2):92–99, March 2018. doi:10.1109/MNET.2017.1700200.

[14] C. Jiang, H. Zhang, Y. Ren, Z. Han, K. Chen, and L. Hanzo. Machine Learning Paradigms for Next-Generation Wireless Networks. *IEEE Wireless Communications*, 24(2):98–105, April 2017. doi:10.1109/MWC.2016.1500356WC.

[15] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks - HotNets '16*, pages 50–56, Atlanta, GA, USA, 2016. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3005745.3005750, doi:10.1145/3005745.3005750.

[16] Mingzhe Chen, Ursula Challita, Walid Saad, Changchuan Yin, and Mérouane Debbah. Machine Learning for Wireless Networks with Artificial Intelligence: A Tutorial on Neural Networks. *arXiv:1710.02913 [cs, math]*, October 2017. arXiv: 1710.02913. URL: http://arxiv.org/abs/1710.02913.

[17] Danish Rafique and Luis Velasco. Machine Learning for Network Automation: Overview, Architecture, and Applications [Invited Tutorial]. *Journal of Optical Communications and Networking*, 10(10):D126, October 2018. URL: https://www.osapublishing.org/abstract.cfm?URI=jocn-10-10-D126, doi:10.1364/JOCN.10.00D126.

[18] Z. Fan and R. Liu. Investigation of machine learning based network traffic classification. In *2017 International Symposium on Wireless Communication Systems (ISWCS)*, pages 1–6, August 2017. doi:10.1109/ISWCS.2017.8108090.

[19] Qianqian Zhang, Mohammad Mozaffari, Walid Saad, Mehdi Bennis, and Merouane Debbah. Machine Learning for Predictive On-Demand Deployment of UAVs for Wireless Communications. *arXiv:1805.00061 [cs, eess, math]*, April 2018. arXiv: 1805.00061. URL: http://arxiv.org/abs/1805.00061.

[20] S. Qian Zhang, F. Xue, N. Ageen Himayat, S. Talwar, and H. T. Kung. A Machine Learning Assisted Cell Selection Method for Drones in Cellular Networks. In *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5, June 2018. doi:10.1109/SPAWC.2018.8445967.

[21] J. Chen, U. Yatnalli, and D. Gesbert. Learning radio maps for UAV-aided wireless networks: A segmented regression approach. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2017. doi:10.1109/ICC.2017.7997333.

[22] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine*, 34(6):26–38, November 2017. doi:10.1109/MSP.2017.2743240.

[23] Hongjia Li, Tianshu Wei, Ao Ren, Qi Zhu, and Yanzhi Wang. Deep Reinforcement Learning: Framework, Applications, and Embedded Implementations. *arXiv:1710.03792 [cs]*, October 2017. arXiv: 1710.03792. URL: http://arxiv.org/abs/1710.03792.

[24] L. Busoniu, R. Babuska, and B. De Schutter. A Comprehensive Survey of Multiagent Reinforcement Learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, March 2008. URL: http://ieeexplore.ieee.org/document/4445757/, doi:10.1109/TSMCC.2007.913919.

[25] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent Reinforcement Learning: An Overview. In Janusz Kacprzyk, Dipti Srinivasan, and Lakhmi C. Jain, editors, *Innovations in Multi-Agent Systems and Applications - 1*, volume 310, pages 183–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. URL: http://link.springer.com/10.1007/978-3-642-14435-6_7, doi:10.1007/978-3-642-14435-6_7.

[26] S. Hayat, E. Yanmaz, and C. Bettstetter. Experimental analysis of multipoint-to-point UAV communications with IEEE 802.11n and 802.11ac. In *2015 IEEE 26th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1991–1996, August 2015. doi:10.1109/PIMRC.2015.7343625.

[27] Metaheuristic vs. deterministic global optimization algorithms: The univariate case | Elsevier Enhanced Reader. doi:10.1016/j.amc.2017.05.014.

[28] D.G. Reina, H. Tawfik, and S.L. Toral. Multi-subpopulation evolutionary algorithms for coverage deployment of UAV-networks. *Ad Hoc Networks*, 68:16–32, January 2018. URL: https://linkinghub.elsevier.com/retrieve/pii/S1570870517301713, doi:10.1016/j.adhoc.2017.09.005.

[29] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012. URL: http://ieeexplore.ieee.org/document/6145622/, doi:10.1109/TCIAIG.2012.2186810.

[30] Prateek Jain and Purushottam Kar. Non-convex Optimization for Machine Learning. *Foundations and Trends® in Machine Learning*, 10(3-4):142–336, 2017. arXiv: 1712.07897. URL: http://arxiv.org/abs/1712.07897, doi:10.1561/2200000058.

[31] Alessio Sancetta. Greedy algorithms for prediction. *Bernoulli*, 22(2):1227–1277, May 2016. arXiv: 1602.01951. URL: http://arxiv.org/abs/1602.01951, doi:10.3150/14-BEJ691.

[32] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*, September 2016. arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[33] O. G. Aliu, A. Imran, M. A. Imran, and B. Evans. A Survey of Self Organisation in Future Cellular Networks. *IEEE Communications Surveys Tutorials*, 15(1):336–361, 2013. doi:10.1109/SURV.2012.021312.00116.

[34] Yufei Ye, Xiaoqin Ren, Jin Wang, Lingxiao Xu, Wenxia Guo, Wenqiang Huang, and Wenhong Tian. A New Approach for Resource Scheduling with Deep Reinforcement Learning. page 5, 2018.

[35] Q. Mao, F. Hu, and Q. Hao. Deep Learning for Intelligent Wireless Networks: A Comprehensive Survey. *IEEE Communications Surveys Tutorials*, 20(4):2595–2621, 2018. doi:10.1109/COMST.2018.2846401.

[36] Xiaming. City Cellular Traffic Map (C2tm). Contribute to caesar0301/city-cellular-traffic-map development by creating an account on GitHub, October 2018. original-date: 2014-10-09T10:47:21Z. URL: https://github.com/caesar0301/city-cellular-traffic-map.

[37] nsnam. ns-3. URL: https://www.nsnam.org.

[38] Helder Fontes, Rui Campos, and Manuel Ricardo. A Trace-based ns-3 Simulation Approach for Perpetuating Real-World Experiments. In *Proceedings of the Workshop on ns-3 - 2017 WNS3*, pages 118–124, Porto, Portugal, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3067665.3067681, doi:10.1145/3067665.3067681.

[39] Helder Fontes, Rui Campos, and Manuel Ricardo. Improving the ns-3 *TraceBasedPropagationLossModel* to support multiple access wireless scenarios. In *Proceedings of the 10th Workshop on ns-3 - WNS3 '18*, pages 77–83, Surathkal, India, 2018. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3199902.3199912, doi:10.1145/3199902.3199912.

[40] V. Sharma, M. Bennis, and R. Kumar. UAV-Assisted Heterogeneous Networks for Capacity Enhancement. *IEEE Communications Letters*, 20(6):1207–1210, June 2016. doi:10.1109/LCOMM.2016.2553103.

[41] A G Barto, P S Thomas, and R S Sutton. Some Recent Applications of Reinforcement Learning. page 6, 2017. URL: https://people.cs.umass.edu/~pthomas/papers/Barto2017.pdf.

[42] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. URL: http://www.nature.com/articles/nature16961, doi:10.1038/nature16961.

[43] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv:1511.05952 [cs]*, November 2015. arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952.

[44] Tao Wang, Michael Bowling, and Dale Schuurmans. Dual Representations for Dynamic Programming and Reinforcement Learning. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 44–51, Honolulu, HI, USA, April 2007. IEEE. URL: http://ieeexplore.ieee.org/document/4220813/, doi:10.1109/ADPRL.2007.368168.

[45] Pieter Abbeel. Markov Decision Processes and Exact Solution Methods:. page 34. Accessed 2019. URL: https://people.eecs.berkeley.edu/~pabbeel/cs287-fa12/slides/mdps-exact-methods.pdf.

[46] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the Complexity of Solving Markov Decision Problems. *arXiv:1302.4971 [cs]*, February 2013. arXiv: 1302.4971. URL: http://arxiv.org/abs/1302.4971.

[47] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 1–6, New York, NY, USA, 1987. ACM. URL: http://doi.acm.org/10.1145/28395.28396, doi:10.1145/28395.28396.

[48] Yuxi Li. Deep Reinforcement Learning. *arXiv:1810.06339 [cs, stat]*, October 2018. arXiv: 1810.06339. URL: http://arxiv.org/abs/1810.06339.

[49] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. page 28, 1992.

[50] Vijay R. Konda and John N. Tsitsiklis. On Actor-Critic Algorithms. *SIAM Journal on Control and Optimization*, 42(4):1143–1166, January 2003. URL: http://epubs.siam.org/doi/10.1137/S0363012901385691, doi:10.1137/S0363012901385691.

[51] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. page 10, 2014.

[52] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, September 2015. arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

[53] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *arXiv:1502.05477 [cs]*, February 2015. arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[54] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, February 2016. arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[55] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample Efficient Actor-Critic with Experience Replay. *arXiv:1611.01224 [cs]*, November 2016. arXiv: 1611.01224. URL: http://arxiv.org/abs/1611.01224.

[56] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation. *arXiv:1708.05144 [cs]*, August 2017. arXiv: 1708.05144. URL: http://arxiv.org/abs/1708.05144.

[57] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, July 2017. arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[58] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *arXiv:1706.02275 [cs]*, June 2017. arXiv: 1706.02275. URL: http://arxiv.org/abs/1706.02275.

[59] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, January 2018. arXiv: 1801.01290. URL: http://arxiv.org/abs/1801.01290.

[60] Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva Tb, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. DISTRIBUTED DISTRIBUTIONAL DETERMINISTIC POLICY GRADIENTS. page 16, 2018.

[61] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. *arXiv:1802.09477 [cs, stat]*, February 2018. arXiv: 1802.09477. URL: http://arxiv.org/abs/1802.09477.

[62] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. page 7, 2000.

[63] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. URL: http://www.nature.com/articles/nature14236, doi:10.1038/nature14236.

[64] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. page 9, 2013.

[65] J. Kim, B. Kim, J. Yoon, M. Lee, S. Jung, and J. y Choi. Robot Soccer Using Deep Q Network. In *2018 International Conference on Platform Technology and Service (PlatCon)*, pages 1–6, January 2018. doi:10.1109/PlatCon.2018.8472776.

[66] Martin Lauer and Martin Riedmiller. An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*, pages 535–542. Morgan Kaufmann, 2000.

[67] OpenAI. Gym: A toolkit for developing and comparing reinforcement learning algorithms. URL: https://gym.openai.com.

[68] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014. arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[69] Kayri. Data Optimization with Multilayer Perceptron Neural Network and Using New Pattern in Decision Tree Comparatively. *Journal of Computer Science*, 6(5):606–612, May 2010. URL: http://www.thescipub.com/abstract/10.3844/jcssp.2010.606.612, doi:10.3844/jcssp.2010.606.612.

[70] Murat Taşkıran, Zehra Gülru Çam, and Nihan Kahraman. An Efficient Method to Optimize Multi-Layer Perceptron for Classification of Human Activities. *International Journal of Computing, Communication and Instrumentation Engineering*, 2(2), December 2015. URL: https://iieng.org/siteadmin/upload/9004ER1215104.pdf, doi:10.15242/IJCCIE.ER1215104.

[71] Zhang Hao. Activation Functions in Neural Networks, May 2017. Accessed 2019. URL: https://isaacchanghau.github.io/post/activation_functions/.

[72] Luke B. Godfrey. Parameterizing and Aggregating Activation Functions in Deep Neural Networks. 2018.