UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Bachelor Degree in Informatics Engineering

Computation

Development of a competitive Rocket League bot using reinforcement learning

Daniel Sarrià Pascual

Sergio Alvarez-Napagao (Director - UPC)

Dmitry Gnatyshak (Co-director - BSC)

Joan Sardà (GEP Tutor)

25-04-2022

Rocket League is a popular game where players can compete against each other or against bots. However, from a competitive point of view, there is still room for improvement on the quality of the bots' capabilities. It is possible to develop, using reinforcement learning, a competitive Rocket League bot able to compete against human players and other traditional bots. The goal of this project is to both make a theoretical study of the foundations of reinforcement learning and a practical solution for a real problem: the need for better Rocket League bots.

The theoretical part will go through decision-making problems, to then explain the main reinforcement learning methods designed to solve them. First, simple methods will be shown, capable of solving small problems. These simple problems hardly exist in real life, so these methods serve as a theoretical background and as a foundation for more complex methods needed to solve real problems. Finally, these more complex methods will be presented and analyzed.

For the practical part, the goal is to develop a bot for Rocket League, using reinforcement learning, capable of competing against human players that are experienced in the game and improving the performance of the game stock bots. First, a brief description will be shown of the concepts to take into account when developing a reinforcement learning solution. Then, the implementation of the solution will be presented, giving readers a guideline of how to develop their own reinforcement learning projects.

Additionally, the project planning is also described, showing both the human and material resources, and how they are distributed over time. The planning includes a budget based on those resources.

This text aims to help people better understand the basics of reinforcement learning, theoretically and practically, so they get the courage to start their own projects.

Abstract 1
Index 2
Part I: Project Introduction 7
Chapter 1: Project Contextualization 8
1.1 Definition of the problem the project aims to solve 9
1.2 Target actors of the project 10
Target actors of the bot 10
Target actors of the text 10
1.3 Is this project really needed? 11
1.4 Organization of the text 11
Chapter 2: How ambitious is the project? 12
2.1 Sub-objectives 12
Finding the best reinforcement learning methods to train the bot 12
Bot consistently scoring goals from any place in the most efficient way 12
Bot consistently beating each of the in-game bots' difficulty levels 12
2.2 Obstacles and risks 13
The supercomputer is not available 13
The personal desktop computer is not available 13
The reinforcement learning method will not train the model in time 13
Part II: Project Planning 14
Chapter 3: Methodology and development cycle 15
3.1 Methodology 15
3.2 Development cycle 15
Reward design 15
Reward development and model training 15
Testing phase 15
3.3 Tools to assure planning and methodology success 16
Chapter 4: Time and resources planning 17
4.1 Project Tasks Descriptions 17
PM (Project Management) 17
TD (Technical Documentation) 18
IG (Information Gathering) 18
PD (Project Development) 19
4.2 Resources 19
Human Resources 19
Material Resources 20
4.3 Project Tasks Overview 21
4.4 Gantt Chart 22

4.5 Obstacles and alternative plans 22
The supercomputer is not available 22
The personal desktop computer is not available 22
The reinforcement learning method will not teach the model in time 23
Chapter 5: Budget planning 24
5.1 Personnel costs per task 24
5.2 Material cost 26
5.3 Space cost 26
5.4 Contingencies cost 27
5.5 Incidental costs 27
5.6 Total cost 28
5.7 Budget control 28
Price deviation cost 28
Time deviation cost 28
Total deviation cost 28
Chapter 6: Preliminary sustainability analysis 29
6.1 Self-assessment 29
6.2 Economic dimension 29
6.3 Environmental dimension 29
6.4 Social dimension 30
Part III: Reinforcement Learning Theory 31
Chapter 7: Artificial Intelligence and Machine Learning 32
7.1 Supervised Learning 34
7.2 Unsupervised Learning 36
7.3 Reinforcement Learning 38
7.4 Summary and connection with the practical part of the project 40
Chapter 8: Decision-Making Problems 41
8.1 Understanding the problem 41
8.2 A practical example 43
8.3 Markov decision process (MDP) 45
8.4 Returns: when immediate rewards are not enough 48
8.5 Value function, action-value function and Bellman equation 50
8.6 Summary and connection with the practical part of the project 53
Chapter 9: Policy optimization with Dynamic Programming 54
9.1 Policy evaluation 54
9.2 Policy improvement 57
9.3 Dynamic Programming Control 58
9.4 Summary and connection with the practical part of the project 58

Chapter 10: Reinforcement learning: tabular solution methods **59**
10.1 Exploration vs Exploitation 59
10.2 Policy optimization using Monte Carlo methods 61
Monte Carlo policy evaluation 61
Monte Carlo control 62
10.3 Policy evaluation using Temporal-Difference learning 63
Value function Temporal-Difference policy evaluation 63
Action-value function Temporal-Difference policy evaluation 64
10.4 Control using SARSA 65
10.5 Control using Q-learning 66
10.6 Summary and connection with the practical part of the project 66
Chapter 11: Reinforcement learning: approximate solution methods **67**
11.1 Function Approximation methods 67
The function approximator: a black box 67
Vector of weights and vector of features 68
The objective function 69
Stochastic gradient descent 70
General update rule 71
Monte Carlo update rule 71
Temporal-Difference learning update rule 71
Function approximation control 71
The function approximator: inside the black box 72
Linear combination of features 72
Monte Carlo update rule using linear combination of features 72
Temporal-Difference learning update rule using linear combination of features 72
11.2 Policy-gradient methods 73
The objective function 73
Stochastic gradient ascent 74
Types of policies 75
The Policy Gradient Theorem 76
Monte Carlo Policy-Gradient (REINFORCE) 76
11.3 Actor-critic methods 77
Actor-critic with a baseline: advantage function 78
The reinforcement learning method used in this project: PPO 80
11.4 Summary and connection with the practical part of the project 82

Part IV: Reinforcement Learning Hands-On **83**
Chapter 12: Designing a reinforcement learning solution to a decision-making problem **84**
12.1 Extracting information from the environment 84
12.2 Defining the MDP 85
12.3 Defining the observation space (features) 85
12.4 Defining the action space 87
12.5 Defining the reward function 87
Chapter 13: The Rocket League bot problem **88**
13.1 Rocket League bot objectives 88
13.2 Rocket League state space 89
13.3 Rocket League action space 90
13.4 Methodology to create the bot 91
Chapter 14: Creating a Rocket League bot using reinforcement learning **92**
14.1 Frameworks and libraries used 92
OpenAI Gym 92
Stable-Baselines3 94
RLGym 94
Other libraries 94
14.3 How the system works 95
14.4 Basics of the model creation and training 98
14.5 Building the environment 98
14.6 The learning framework 102
14.7 Other useful tools created: the state setter and the mathematical functions 105
The state setter 105
Mathematical functions 106
14.8 Learning class implementations to create a model capable of bringing the car from a point A to a point B 107
An important concept: angle to the objective point 107
Four different approaches 108
Observation comparison reward functions 109
Action based reward functions 110
Observation based reward functions with positive rewards 112
Observation based reward functions with negative rewards 112
14.9 Code structure and how to run it 113
Code structure 113
How to install and run the code 115

Chapter 15: Experiments **116**
15.1 Boxplots of the results 117
15.2 The best model 119
Part V: Conclusions **120**
Conclusions 1: Planning changes **121**
Conclusions 2: Objective and scope changes **121**
Conclusions 3: Sustainability analysis **122**
Conclusions 4: Technical competences analysis **122**
CCO1.1 122
CCO2.1 and CCO2.4 122
CCO2.2 122
CCO2.6 123
Conclusions 5: Personal impact of the project **123**
Part VI: Annexes **124**
Annex 1: Video Game Bot **125**
Annex 2: Classic Expert Systems **125**
Annex 3: More information about Rocket League **126**
Annex 4: Learning classes code **127**
Part VII: References **136**

Part I:
Project Introduction

This first part of the text goes through the reasons why this project is carried out and its objectives. It also explains the project's organization and the audience it is directed towards.

Chapter 1: Project Contextualization

Rocket League is a highly acclaimed video game that perfectly merges cars and football. It was released on July 7, 2015, developed and published by Psyonix. Its original and main gamemode, “soccar”, takes place in a football-like field, but with a twist. It is an enclosed location with driveable walls. Two teams, orange and blue, participate in the match, with up to four players each. Each of the players control one car. The objective of the game is to score more goals than the opposing team. In order to score, the team has to introduce the ball into the rival’s net. In the figure 1.a you can see one of the Rocket League soccar standard fields.

Figure 1.a. Rocket League soccar gamemode field.
https://rocketleague.media.zestyio.com/Champions-Field1920.f1cb27a519bdb5b6ed34049a5b86e317.jpg

The game is mostly played online, where human players compete against each other for the victory. While it is possible to play against bots (players controlled by the game itself), they hardly represent a threat to players a few hours into the game. The aim of this project is to develop, using reinforcement learning, a Rocket League bot that represents a real challenge for experienced players, that can be enjoyed throughout the game’s lifetime.

This project is carried out within the framework of the Bachelor Degree in Informatics Engineering imparted by Facultat d'Informàtica de Barcelona (FIB), that is part of the Universitat Politècnica de Catalunya (UPC). This project pretends to put into practice all the knowledge acquired during the bachelor degree to solve a real problem and to learn how to cope with the difficulties a professional project presents.

This project has three different goals. The first one is personal, as it is a way to be introduced to the reinforcement learning world, both on a theoretical and a practical level. The second goal is to produce a text that will allow people in the same situation, willing to learn about it, to be introduced in an easy way to reinforcement learning. The third goal is to solve a problem that Rocket League has. This issue has created the perfect situation in order to fulfill the first two goals while solving a real problem at the same time. In the next section, the problem with Rocket League will be discussed.

1.1 Definition of the problem the project aims to solve

As mentioned in the brief introduction, Rocket League has a problem with their bots, due to the fact that they are not competitive enough to be part of the daily gameplay of the average player. As a Rocket League player, you can compete against bots in season, exhibition, private and in unranked online matches (you can find information about this in annex 3). Season and exhibition matches against bots are normally created by new players that are too afraid to compete against human players and want to learn the basics of the game. However, bots have little to offer in terms of teaching new players how to play the game, given their poor capabilities and the silly mistakes they sometimes make (like getting stuck in the net). In unranked games, when a player leaves the game before it ends, the player that left is replaced by a bot until a new player joins. It can be useful for low level lobbies, but for mid to high level lobbies they are more of a nuisance than a help. The game would radically improve if the player could compete in challenging seasons or if the bots that replace human players that leave an unranked match had a similar level to the players in the lobby.

1.2 Target actors of the project

As mentioned previously, this project has several goals. The target of the first one is personal, obviously. For the other two, the targets are quite different. This project can appeal to people interested in the game and the capabilities this bot could bring, as well as people that currently develop bots for it, and would like to take a look at the different perspective this project proposes (more on this in the next section). Last but not least, people that want to learn about reinforcement learning could be interested too.

Target actors of the bot

The target actors of the bot are a part of the Rocket League player base that would enjoy a more competitive bot scene, and even some people that could be attracted by the improved offline experience. This improvement of the video game bots would give Rocket League players a new way of playing it, making it less monotonous and even attracting a new audience. There is a special sector of the Rocket League player base that participates in a community named "RLbot" [1], created to share and play against bots created by themselves (more on this in the next section). People that are curious about bots created by the community could put their abilities into test competing against the bot developed in this project, and even other bot developers could test how good their bots are.

Target actors of the text

As mentioned before, there is a community dedicated to creating, sharing and playing against Rocket League bots created by themselves. Apart from the bot developed in this project, they could take advantage of how it was made. People willing to learn more about developing an agent capable of playing video games by teaching them using reinforcement learning can take a look at the work made here. Apart from that, this text serves as an introduction to reinforcement learning, and for this reason, any person willing to learn the basics of it could read it and get a better understanding on how reinforcement learning works and even start a project themselves.

1.3 Is this project really needed?

One of the questions you have to ask yourself when starting a new project is whether the problem needs to be solved or not, and even in the case it does, you need to know if your proposed solution has already been studied or put into practice. For this reason, in this section we are going to dive into the solutions the community has proposed, and how this approach differs from the rest.

The community has come up with several Rocket League bot designs, some of them capable of beating mid level players. There are even some bots especially designed for freestyling (information about it in annex 3). While Rocket League has not added neither seems to have the interest of adding these bots to the game, the community holds tournaments to see which bot is the best, which is a step ahead to having better bots in Rocket League.

Most of the attempts at creating Rocket League bots have been using Classic Expert Systems (annex 2). In this project, the bot development will take a different approach by teaching it how to play the game by using reinforcement learning. This alternative has been explored a few times, but it did not unleash its full potential. This project will try to go deeper into exploring the capabilities of reinforcement learning when teaching bots to play Rocket League. The project will start from scratch, using OpenAI's gym framework for reinforcement learning.

1.4 Organization of the text

The text is divided into different parts, being this the first one. Each part treats completely different aspects of the project. This first part is about the introduction to the project. The second one treats the planning and budget of it. In the third one, aspects of reinforcement learning theory are discussed. In the fourth, the work on the bot is explained and in the fifth conclusions are drawn. Sixth and seventh are for the annexes and references respectively.

Each part has several chapters that are usually connected. All the chapters of the book share the same numeration. Some chapters might have more than one section.

Chapter 2: How ambitious is the project?

As mentioned in the previous chapters, the project goal is to create a Rocket League bot that is more challenging than the ones currently in the game. For this reason, a way of measuring how successful the bot development is would be to play a series of games between the created bot and the in-game bot with the highest difficulty level. If the created bot can consistently beat the in-game bot, the project goal would be fulfilled. The academic nature of this project sets a deadline for the development of the Rocket League bot, which may put a limit on the learning time of the bot, as well as the study of the different learning possibilities. However, after crossing this deadline, the project can mutate into a personal or even a research one, as the bot will still have a lot of room to improve. Now we are going to dive into the sub-objectives that will serve as a guide during the development, to assess the performance of the bot and to evaluate if the project is moving in the right direction.

2.1 Sub-objectives

Finding the best reinforcement learning methods to train the bot

In this first phase, different reinforcement learning methods will be taken into account and evaluated. The bot will be trained to do certain simple actions with each one of them and this way decide which methods adjusts better to the problem we have in hand. Once the best reinforcement learning method is selected, the actual training can commence, starting then the path to fulfill the next sub-objective.

Bot consistently scoring goals from any place in the most efficient way

After selecting the best reinforcement learning method for this problem, our next sub-objective will be to assure that the bot is capable of consistently scoring from every position, approaching and moving the ball as needed. For this first sub-objective, no rival will be used, as it would be difficult for the bot to train those simple actions while the rival bot is scoring all the time. Scoring (and not getting scored on) is the main objective of a Rocket League game, and that is the reason why this is the starting point for the bot.

Bot consistently beating each of the in-game bots' difficulty levels

The next sub-objectives would be to consistently beat each one of the in-game bots' difficulty levels. The four bot difficulty levels are Beginner, Rookie, Pro and All-Star, from the lowest to the highest. In order to achieve that, the bot’s strategies will need to be improved, like for example not going for the ball if the rival is significantly closer to it.

2.2 Obstacles and risks

The supercomputer is not available

The MareNostrum supercomputer is expected to be available to train the model. If it was not available during the project or at a certain point, the personal desktop computer would be used instead to teach the models.

The personal desktop computer is not available

If the personal desktop computer was not available during the project or at a certain point, a backup laptop would be used instead to teach the models.

The reinforcement learning method will not train the model in time

It is possible that the model does not meet the objectives mentioned above given the fact that there is no time to train the model well enough. This would be due to the fact that the computation power was not enough or that the improvements between models was not good.

Part II:
Project Planning

This second part covers the planning of the project. From the methodology used to carry it out to all the tasks planned, everything is specified here. All the resources (human and material) are accounted for, and a budget based on it is generated.

Chapter 3: Methodology and development cycle

3.1 Methodology

The general methodology to be used to develop the project is Agile [4], in the sense that the development will follow a cycle of actions. These actions correspond to **designing** the rewards, **developing** the code for them and training the model, and **testing** the model against the previous version.

3.2 Development cycle

As mentioned before, the method used to assess progress will be to face the bot with the in-game bots. Given two reinforcement learning models, the one with more wins of a significant set of matches played will be considered as the best so far. This way, the development cycle will include three parts: the design of the rewards given to the agent, the training of the model, and the testing of the model. If the current model's test gives worse results than the previous model, the cycle will start again with that previous model.

Reward design

In this first part, the rewards given to the agent in every state and action pair are designed. A smooth transition has to be assured between the last and the current model in order to make the change of the reward given in similar state-action pairs not change too much.

Reward development and model training

In this second part, the design of the code rewards is converted into code. The model is then trained using these rewards. The agent will move through the environment following the last model and will update it to create the new model given the new rewards it obtains. When it maximizes the reward obtained (the behavior does not change anymore, or does not change much, at least), the model will go into the next phase.

Testing phase

In this phase, the bot will play a series of matches against an in-game bot following the last model trained. If the current model has a smaller win ratio than the previous model, the model will be reverted, and the previous one will be used to start the cycle again.

3.3 Tools to assure planning and methodology success

GitHub [2] will be used as the code storage and control version of the project.

Trello [3] will be used to list all the tasks with their deadlines, this way having a place to organize the project planning.

Figure 3.a. GitHub and Trello logos.

Chapter 4: Time and resources planning

Before starting a project, it is important to set a timeline in order to assess that the progress is going as planned. The project started September 1st of 2021 and it is expected to be finished by January 22nd of 2022. The work on the presentation will be after that date. The planned daily work is 5 hours. With the dates indicated above, the project is expected to last for 144 days, which means 720 hours of work. The first step to successfully planning the project is to define the tasks that have to be performed during the project. After that, the human and material resources are defined. In section 4.4, in this chapter, there is an overview of all the tasks with the time they take to complete, the number of times the tasks are carried out, their dependencies and their material and human resource requirements.

4.1 Project Tasks Descriptions

Given the iterative nature of the project, some of the tasks will be repeated during the project. Tasks are grouped in categories to help understand the similarities between them, but these similar tasks do not have to be necessarily performed at the same time or close in time. Each group of tasks would be performed by a different type of professional if the project was done by a team. Here, only a simple description of each task is given, while in the tasks overview the time and resources they take is specified.

PM (Project Management)

This group of tasks is dedicated to decide how the project will be done, assess the performance and timing of it, resolve problems and conflicts, basically taking global decisions that affect the whole project. Apart from that, some of these processes are included in the report.

- PM.1 (Project Definition): In this task, the goals of the project are defined, along with the methodology used to achieve them. The decisions made are included in the report during this task.
- PM.2 (Project Planning): Here, the timeline of the project is defined. Tasks are explained and their distribution throughout time is planned. The decisions made are included in the report during this task.
- PM.3 (Project Budget): This task specifies the budget of the project, taking into account the personal and material costs, the last one including machines usage cost, energy consumption and space cost. The decisions made are included in the report during this task.
- PM.4 (Project Sustainability): On this occasion the impact the project has on the environment is taken into account. The decisions made are included in the report during this task.

- PM.5 (Documentation Revision): On this task, the written report regarding the project management is revised.
- PM.6 (Meetings): These are the meetings with the project directors to discuss the state of the project.

TD (Technical Documentation)

This group of tasks is dedicated to writing the technical part of the report.

- TD.1 (Theory Documentation): In this task, all the information needed for the reader to understand the project is redacted.
- TD.2 (Development Documentation): Here, all the information regarding all the work done during the project development is written in the report.

IG (Information Gathering)

This group of tasks are dedicated to gathering important information about the technologies and techniques used to develop the project as well as to get the knowledge needed.

- IG.1 (General Reinforcement Learning Info): In this task, the general knowledge about Reinforcement Learning is acquired.
- IG.2 (Model-Free Reinforcement Learning Info): In this task, the knowledge about model-free Reinforcement Learning is acquired.
- IG.3 (Model-Based Reinforcement Learning Info): In this task, the knowledge about model-free Reinforcement Learning is acquired.
- IG.4 (Frameworks and Technologies Info): In this task, the knowledge about the different frameworks and technologies that can be used is acquired.
- IG.5 (Reinforcement Learning Method Info): In this task, the knowledge about a reinforcement learning method is acquired. Given the knowledge gathered in the two previous defined tasks, six reinforcement learning methods are selected to be studied in this task, one for each repetition of this task.
- IG.6 (Reinforcement Learning Final Method Info): In this task, the knowledge about the chosen reinforcement learning method is acquired.

PD (Project Development)

This group of tasks are dedicated to developing the Rocket League bot.

- PD.1 (Environment Preparation): In this task, the python environment and the base code is prepared to start the project.
- PD.2 (Reinforcement Learning Method Testing): In this task, the reinforcement learning method being studied is put into test to see if it is good enough to be used to develop the project.
- PD.3 (Reward Design): In this task, the rewards that will be given to the agent are designed for the current development cycle.
- PD.4 (Model Learning): In this task, the agent automatically learns and improves the model. This is an automatic task that requires no human interaction.
- PD.5 (Model Testing): In this task, the latest model is compared to the previous one to assess whether it is better or not.

4.2 Resources

Human Resources

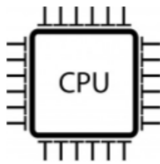
In the projects carried out by a team, usually each person or a group of people has a different role, and each role is responsible for a certain type of tasks. This project is developed by one person, with the help of the director and co-director, but roles will be defined to assign different types of tasks and to have more control on the budget. The defined roles are:

- PM (Project Manager): People in this role are responsible for defining the goals of the project, and making sure that all the team members work together towards it. After speaking with the other members of the team, they define the tasks to perform during the project and make sure that they are accomplished in time. Finally, they have to plan a budget and also make sure that it is respected.
- RE (Machine Learning Researcher): The machine learning researcher is responsible for analyzing the nature of the problem and researching the best technologies that can be used to solve it. People in this role must have a deep understanding of the problem and these technologies to be able to choose the best one.
- EN (Machine Learning Engineer): Machine learning engineers must have a general and practical knowledge on the different technologies and algorithms to be able to use and apply them to solve the problem.

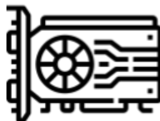
Material Resources

For this project, different material resources are available:

- DC (Personal desktop computer): Personal computer with the following characteristics:



CPU: AMD Athlon X4 750k - 2 cores / 4 threads



Graphics Card: Nvidia GTX 1050 Ti

RAM: ddr3 8gb

Figure 4.a. Personal desktop computer characteristics

- SC (Supercomputer): MareNostrum supercomputer

4.3 Project Tasks Overview

Here we have an overview of the tasks and the resources they need. Given to tasks A and B, A < B means that A has to be done before starting B, and A | B means that they have to be done at the same time.

Task ID	Task Name	Task Time (h)	Task Repeti-tions	Dependencies	Resour-ces	Roles
PM.1	Project Definition	20	1	-	DC	PM
PM.2	Project Planning	20	1	PM.1 < PM.2	DC	PM
PM.3	Project Budget	10	1	PM.2 < PM.3	DC	PM
PM.4	Project Sustainability	10	1	PM.3 < PM.4	DC	PM
PM.5	Documentation Revision	20	1	PM.4 < PM.5	DC	PM
PM.6	Meetings	1	15 approx.	-	DC	PM
TD.1	Theory Documentation	50	1	-	DC	RE
TD.2	Technical Documentation	50	1	-	DC	RE
IG.1	General Reinforcement Learning Info	50	1	-	DC	RE, EN
IG.2	Model-Free Reinforcement Learning Info	20	1	IG.1 < IG.2	DC	RE, EN
IG.3	Model-Based Reinforcement Learning Info	20	1	IG.1 < IG.3	DC	RE, EN
IG.4	Frameworks and Technologies Info	20	1	IG.1 < IG.4	DC	RE, EN
IG.5	Reinforcement Learning Method Info	10	6	IG.1, IG.2, IG.3 < IG.5	DC	RE, EN
IG.6	Reinforcement Learning Final Method Info	30	1	IG.5 < IG.6	DC	RE, EN
PD.1	Environment Preparation	10	1	IG.4 < PD.1	DC	EN
PD.2	Reinforcement Learning Method Testing	10	6	IG.5 | PD.2	DC	EN
PD.3	Reward Design	14	17 approx.	PD.1, PD.2 < PD.3	DC	EN
PD.4	Model Learning	96	17 approx.	PD.3 < PD.4	SC	-
PD.5	Model Testing	1	17 approx.	PD.4 < PD.5	DC	EN

Figure 4.b. Project tasks overview table. Own elaboration.

4.4 Gantt Chart

The Gantt Chart in the figure 4.c shows the distribution of tasks over time. The task PM.6 will be carried out intermittently, but as no concrete dates are set, it is shown as a continuous task. Tasks PD.4, PD.5 and PD.6 will be carried out in a cycle as mentioned in chapter 3, but as no dates can be determined for the cycles, they are shown as a continuous task. PD.5 is in red due to the fact that it does not need human intervention.

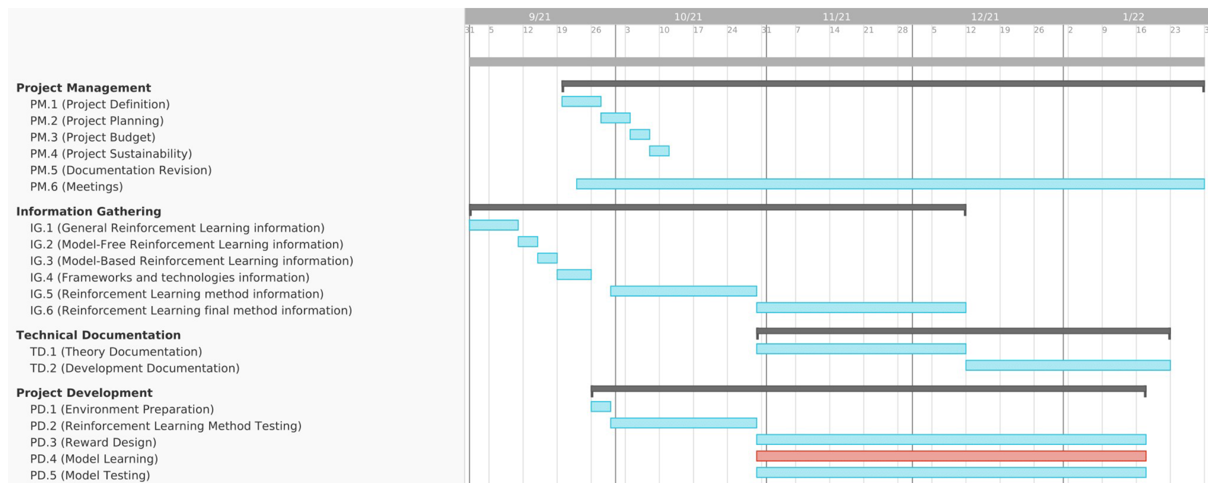


Figure 4.c. Gantt Chart of the project. Own elaboration.

4.5 Obstacles and alternative plans

In case something does not go as planned, the project has to have alternative plans. These plans include extra material resources and new tasks.

The supercomputer is not available

If the supercomputer was not available during the project or at a certain point, the personal desktop computer would be used instead to teach the models. This would mean that the task PD.3 (Model learning) would be significantly longer. As the bot development does not have a defined ending, we can cut at any point, but if the goal of beating the in-game bots still applies, it would probably need more development time. If only counting the academic period of the project, the task cannot be stretched as it is performed non stop automatically since it starts.

The personal desktop computer is not available

If the personal desktop computer was not available during the project or at a certain point, a backup laptop would be used instead to teach the models.

The reinforcement learning method will not teach the model in time

If the best Reinforcement Method found to build the model is not fast enough to meet the objectives set, the plan B is to build a certain part of the bot with a reinforcement learning model and another one with a Classic Expert System. In this case, fewer cycles of model learning (tasks PD.3, PD.4 and PD.5) would be needed (a reduction in time of 50%), but extra tasks of the Classic Expert System programming would be needed. Without counting the PD.4 task, which is performed without human interaction, the time reduction would be from 255 to 150 hours, while the remaining 105 hours would be used in the new ES (Expert System) tasks to make the car perform a series of actions. An additional 95 hours would be needed for these new ES tasks. In total, the ES tasks would amount to 200 hours. This is a brief description of these ES tasks, that are performed by the programer (the same role that the task that it substitutes):

- ES.1 (Ball chasing mechanic) - 40 hours: In this task, an algorithm that makes the car chase the ball is programmed.
- ES.2 (Shadow mechanic) - 40 hours: In this task, an algorithm that makes the car follow the movements of the rival to defend against shots is programmed.
- ES.3 (Movement mechanic) - 40 hours: In this task, an algorithm that makes the car move to a certain position is programmed.
- ES.4 (Shot mechanic) - 40 hours: In this task, an algorithm that makes the car shoot the ball into the goal is programmed.
- ES.5 (Save mechanic) - 40 hours: In this task, an algorithm that makes the car save the goal from scoring into its net is programmed.

Chapter 5: Budget planning

As with the time planning, all projects should have a defined budget to assess whether the costs are out of control or not. If this budget was not defined, the members working on the project would not know if they have spent too much in a certain point in time.

5.1 Personnel costs per task

First of all, it is important to define the salary per hour that would need to be satisfied for each one of the members of the project team. Having this data, it can be applied to calculate the cost of each one of the tasks mentioned in chapter 4, in this part of the text. In the figure 5.a, approximate salaries for each role previously defined are specified. Annual salaries are calculated by multiplying the gross salary by a 1.3 factor (to include the Social Security expenses). Hourly salaries are calculated by dividing annual salary by 1,800.

Roles	Annual Salary (including Social Security)	Hourly Salary (including Social Security)
PM (Project Manager)	67,600€	37.56 €/h
RE (Machine Learning Researcher)	49,400 €	27.44 €/h
EN (Machine Learning Engineer)	58,890€	32.72 €/h

Figure 5.a. Salaries for each role based on [5]. Own elaboration.

With the salaries calculated earlier, the personnel cost for each one of the tasks can be calculated.

TASK ID	TASK NAME	TASK TIME (h)	TASK REPETITIONS	ROLES	COST
PM.1	Project Definition	20	1	PM	751.20€
PM.2	Project Planning	20	1	PM	751.20€
PM.3	Project Budget	10	1	PM	375.60€
PM.4	Project Sustainability	10	1	PM	375.60€
PM.5	Documentation Revision	20	1	PM	751.20€
PM.6	Meetings	1	15 approx.	PM	1690.20€
TD.1	Theory Documentation	50	1	RE	1372.50€
TD.2	Technical Documentation	50	1	RE	1372.50€
IG.1	General Reinforcement Learning Info	50	1	RE, EN	1504.00€
IG.2	Model-Free Reinforcement Learning Info	20	1	RE, EN	601.60€
IG.3	Model-Based Reinforcement Learning Info	20	1	RE, EN	601.60€
IG.4	Frameworks and Technologies Info	20	1	RE, EN	601.60€
IG.5	Reinforcement Learning Method Info	10	6	RE, EN	1804.80€
IG.6	Reinforcement Learning Final Method Info	30	1	RE, EN	902.40€
PD.1	Environment Preparation	10	1	EN	327.20€
PD.2	Reinforcement Learning Method Testing	10	6	EN	1963.20€
PD.3	Reward Design	14	17 approx.	EN	7787.36€
PD.4	Model Learning	96	17 approx.	-	-
PD.5	Model Testing	1	17 approx.	EN	556.24€
-	**TOTAL**	**-**	**-**	**-**	**24090.00€**

Figure 5.b. Cost by each task. Own elaboration.

5.2 Material cost

The next step is to calculate the cost associated with the material used. Every material object used has a building or acquisition cost and a life span. From these, the cost for each hour of usage can be estimated. The amortization cost is the result of the cost per hour times the hours the object is estimated to be used. In the figure 5.c, the amortization costs are calculated for the objects to be used in the project.

Material Resource	Cost	Life Span	Cost per Hour	Usage Time	Amortization
Personal Desktop Computer	400 €	10 years (87600 hours)	4.57×10^{-3}	765 h	3.50 €
Supercomputer	144.68 € per node	5 years (43800 hours)	3.30×10^{-3}	1920 h	6.34 €
TOTAL	**-**	**-**	**-**	**-**	**9.84 €**

Figure 5.c. Amortization for the material used based on [6]. Own elaboration.

5.3 Space cost

The next cost to consider is the one associated with the space occupied by the personnel to do their job. To calculate its cost, the first thing to know is the approximate cost per square meter in the zone where the space is located. In this case, the rental cost is considered, so cost per square meter per month is used. The space to be used is located in Rubí, and having its cost per square meter per month, it is easy to calculate its total cost by multiplying it by the number of months to be used. In the figure 5.d, an overview can be seen.

Space Resource	Cost per m^2 / month	Surface	Monthly cost	Number of months	Total cost
Room	9.8 €	9 m^2	88.20 €	5	441.00 €
TOTAL	**-**	**-**	**-**	**-**	**441.00 €**

Figure 5.d. Cost of the space based on [7]. Own elaboration.

5.4 Contingencies cost

Contingencies are costs budgeted to take into account possible deviations from the original budget. In this case, a 15% extra cost from the base costs is considered as contingencies.

Cost Type	Cost	Contingency Percentage	Contingency Cost
Personnel	24090.00€	15 %	3613.50 €
Material	9.84 €	15 %	1.48 €
Space	441.00 €	15 %	66.15 €
TOTAL	**-**	**-**	**3681.13 €**

Figure 5.e. Cost of the contingencies. Own elaboration.

5.5 Incidental costs

The incidental costs are related to incidents that alter the initial planning. These planning changes can include the alteration of tasks (in length or personnel), the addition and/or removal of tasks, change on material used to carry them out… For each one of the possible incidents, a percentage is estimated, determining the probability of the incident happening. Given the cost that the incident supposes and its probability, the budgeted cost of the incident is those to values multiplied.

Cost Type	Cost	Incident Probability	Cost Given Probability
Additional 95 hours of EN (Machine Learning Engineer) time	3108.40€	0.5	1554.20€
TOTAL	**-**	**-**	**1554.20€**

Figure 5.f. Cost of the incidents. Own elaboration.

5.6 Total cost

In the figure 5.g there is an overview of all the costs budgeted.

Type of Cost	Cost
Personnel	24090.00€
Material	9.84 €
Space	441.00 €
Contingencies	3681.13 €
Incidents	1554.20€
TOTAL	**29776.17€**

Figure 5.g. Total cost. Own elaboration.

5.7 Budget control

As mentioned at the start of this chapter, designing a budget is useful for the project members to know whether the costs are out of control or not. For this reason, it is important to design a mechanism to compare the estimated costs with the real ones. We have three different methods to compute it.

Price deviation cost

In the case of tasks, material resources, space, the price deviation cost is calculated using the following formula:

*(estimated hourly cost - real hourly cost) * hours spent*

Time deviation cost

In the case of tasks, material resources, space, the time deviation cost is calculated using the following formula:

*(estimated hours spent - real hours spent) * real cost*

Total deviation cost

In the case of tasks, material resources, space, the total deviation cost is calculated using the following formula:

(estimated cost - real cost)

Chapter 6: Preliminary sustainability analysis

It is always important to know if the project to be done is really beneficial. To assess so, three dimensions have to be explored regarding the sustainability of it: Economical, Environmental and Social. Before exploring them, a self-assessment of the author of this project has been made about the knowledge about the importance of the sustainability of the projects.

6.1 Self-assessment

During my lifetime, I have been more and more aware of the economical, social and environmental implications of the projects carried out by myself and other people. I am able to assess, in a general manner, whether a project is sustainable in these three dimensions. However, I have always been more focused on the environmental dimension, as it is the one that is more often mentioned when talking about sustainability. This has given me the ability to carry out projects that can help the society in some way without having a negative impact on the environment and economy, and it is my intention and motivation to do so. Once said, it is also true that carrying out a project that perfectly maximizes the benefit for the society, does not harm the environment (or heals it) and that produces a direct or indirect economic benefit is so difficult. I have the intuition of how to do it, but by no means the knowledge on the terminology and techniques used to maximize these benefits.

6.2 Economic dimension

Given the importance that reinforcement learning has, and that will have in the future, further investigation in that field is important as understanding it better can lead to huge economic benefits. During its lifespan, the result of this project does not have a huge cost, only the energy spent on executing it, which is not much.

6.3 Environmental dimension

From an environmental point of view, the cost of this project is related to the waste of energy used to train the AI model. A project of these characteristics can have a significant impact on the environment as the training can last for the equivalent to years of processing if done with a single personal computer. Solutions to minimize the waste of energy are to use more efficient algorithms that require less computational time to create the same AI model, and using computers that are powered by clean energy.

6.4 Social dimension

On a personal level, this project can bring its development team knowledge on how to manage a medium-sized project and overcome the difficulties. From a technical point of view, it is a great opportunity to learn about reinforcement learning, which is one of the most important paradigms of AI. As mentioned in previous chapters, most of the Rocket League bots are developed using classic expert systems, which are a simpler way of creating an AI, and that are better known. Using reinforcement learning is a great opportunity to contribute to the development and better understanding of this technology. While the product itself resulting from this project does not have a meaningful impact on society, the study of reinforcement learning has. Reinforcement learning promises, among other things, to create AI's that aid people to have better and easier lives, and it is a goal of this project to contribute to so.

Part III: Reinforcement Learning Theory

This third part of the text will dive into Reinforcement Learning theory. But before doing so, the text will present the context of Artificial Intelligence and how Machine Learning can help achieve it. Only after this, the problem that reinforcement learning tries to solve will be explained. Then, Reinforcement Learning solutions will be exposed, starting with its basis and ending with more technical and complex aspects of it. The reinforcement learning methods shown in this part do not represent the totality of the methods that exist, but are the basis for most of them. This means that understanding them will help understand the methods used to solve real problems.

Chapter 7: Artificial Intelligence and Machine Learning

Artificial Intelligence [8] is the capability of machines to show and demonstrate intelligence. Of course, this is a broad and open definition and it does not help much, but it is also true that natural intelligence is equally hard to define and delimit. When do we consider an animal as intelligent? When do we consider an algorithm or program as intelligent? There are infinite correct answers to those questions. For the sake of this text, algorithms and programs that analyze their environments and make decisions based on those observations will be considered as Artificial Intelligence.

Artificial Intelligence is not a specific technology or algorithm, it is a concept that can also be considered as a goal, and can be achieved using various technologies and algorithms. For this reason, throughout the years, studies have been conducted to try and find the best algorithms that grant machines with intelligence. Examples of these are Classic Expert Systems, Cognitive Models [9], Machine Learning, etc. As reinforcement learning is one of the three pillars of Machine Learning, we will take a moment to explain what Machine Learning consists of.

Machine Learning is a set of algorithms that aim to learn to recognize patterns in data in order to make predictions or decisions. Predictions are things like determining if the animal in an image is a cat or a dog, deciding if an autonomous car should brake or not in a certain situation, etc. Machine Learning algorithms build models based on the so-called “training data”, and then those models can be used to make those predictions or decisions. As can be seen in the figure 7.a, which represents the training phase, the Machine Learning algorithm takes some training data and uses it to improve the model. It is an iterative process, as data is used several times to make the model more precise in each step.

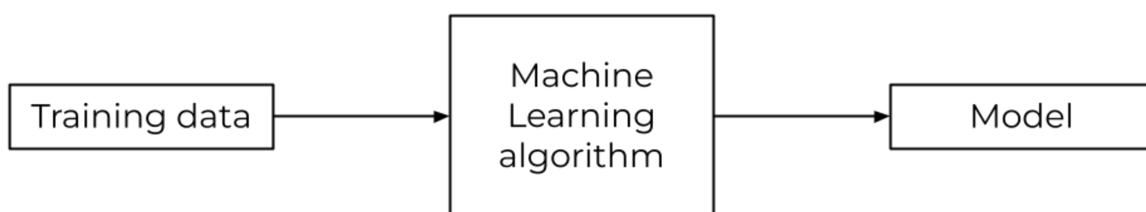


Figure 7.a. Machine Learning training phase diagram

Once trained, the model can be used to predict or make decisions. As seen in the figure 7.b, the data is input to the model, which will determine the output data that it predicted.

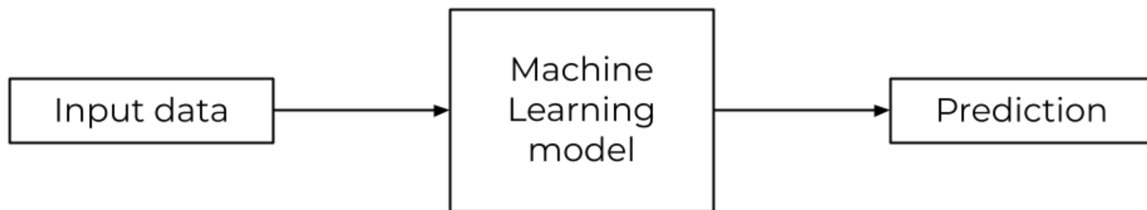


Figure 7.b. Machine Learning predicting phase diagram

Machine Learning has three major approaches: **Supervised learning**, **unsupervised learning** and **reinforcement learning**. While there are other minor approaches and combinations between the three major ones, we are only going to focus on these three. Even if this text focuses on reinforcement learning, the next three sections will give a general overview of these three areas of machine learning, to see their differences and applications, and to justify why reinforcement learning was chosen for this project. In figure 7.c you can see a tree with the different areas of Machine Learning that we will be visiting. Most of them are related to reinforcement learning, as we will go deeper in that area.

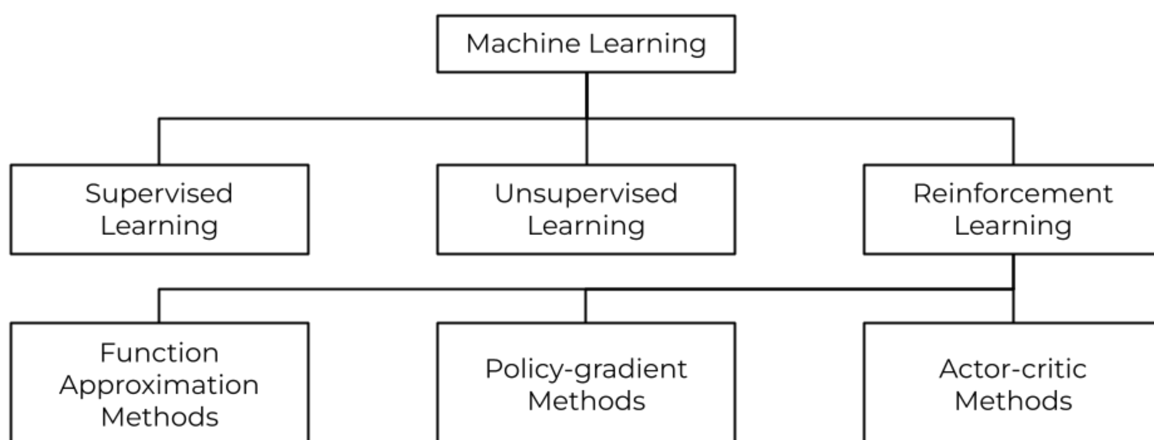


Figure 7.c. Machine Learning tree

7.1 Supervised Learning

Supervised learning [10] is one of the three machine learning major approaches. Its goal is to classify data or predict outcomes, based on a model learnt using training data. To learn that model, the algorithm is given input-output pairs, what is known as labeled data. From this labeled data, the algorithm infers a function that maps the input data to the output data.

The example in the figure 7.d shows how, in order to train the model, a set of input-output data is given to the supervised learning algorithm, which will produce a model as an output. This is a classification problem, which means that the input data is part of one of the several categories in the problem. In this case, there are three categories: square, circle and triangle. Training data are pairs, where the input is a 2D shape, and the output is the category they are in. This is input to the algorithm, which will infer a function in order to be able to recognize these shapes and classify them properly. The learning process generates a model that can be used in the future to classify shapes that have not been seen in the training data.

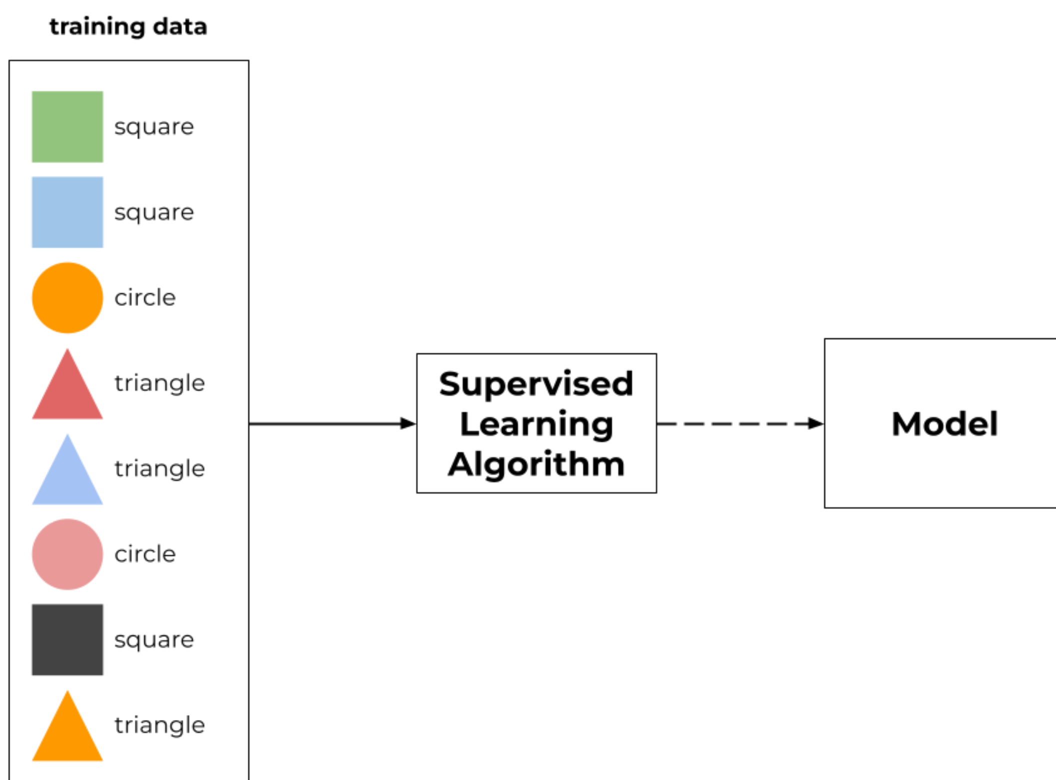


Figure 7.d. Supervised learning training

Once the model is generated, it can be used to classify the shapes, as seen in the figure 7.e. As can be seen, the input data that we want to classify does not need to be the same as the one used in the training phase. Here, several shapes have different colors than the ones used to generate the model. They could also be of slightly different shapes, as for example, the triangle could have different angles. The important part is that the supervised learning algorithm will be able to, if done properly, generalize enough the function in order to classify properly the shapes even if they do not match perfectly the patterns observed in the training phase.

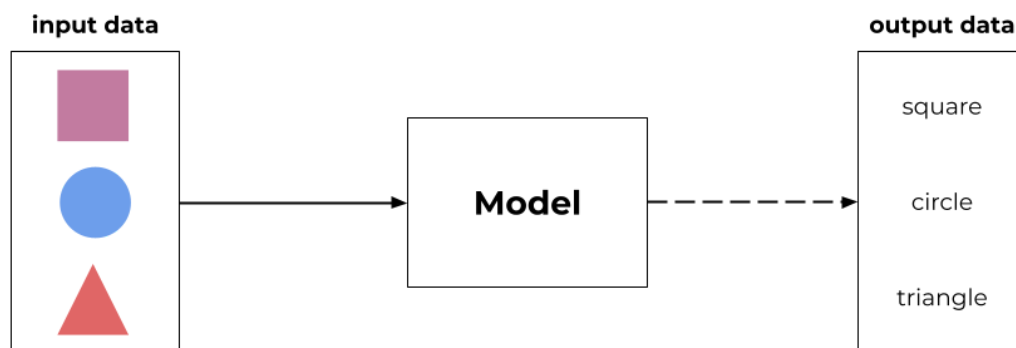


Figure 7.e. Supervised learning prediction.

This was a classification example, but supervised learning can be used to predict different types of things, not necessarily mapping the input to a category. For example, it could be used to predict a person's salary based on some variables such as age, gender, hometown, level of education, etc.

7.2 Unsupervised Learning

Unsupervised learning [11] is another one of the three machine learning major approaches. Its goal is to find patterns in data that is not known yet. The most important difference with supervised learning is that, unlike supervised learning, the input is not labeled.

In the figure 7.f, we have a representation of what could be a bunch of data we do not know much about. It is obvious that there are circles with different colors, because it is just a simple example. But let's imagine that there is not a clear pattern, and we want to find some interesting patterns in order to classify the data. After inputting the data, the unsupervised learning algorithm would identify these patterns and create several categories.

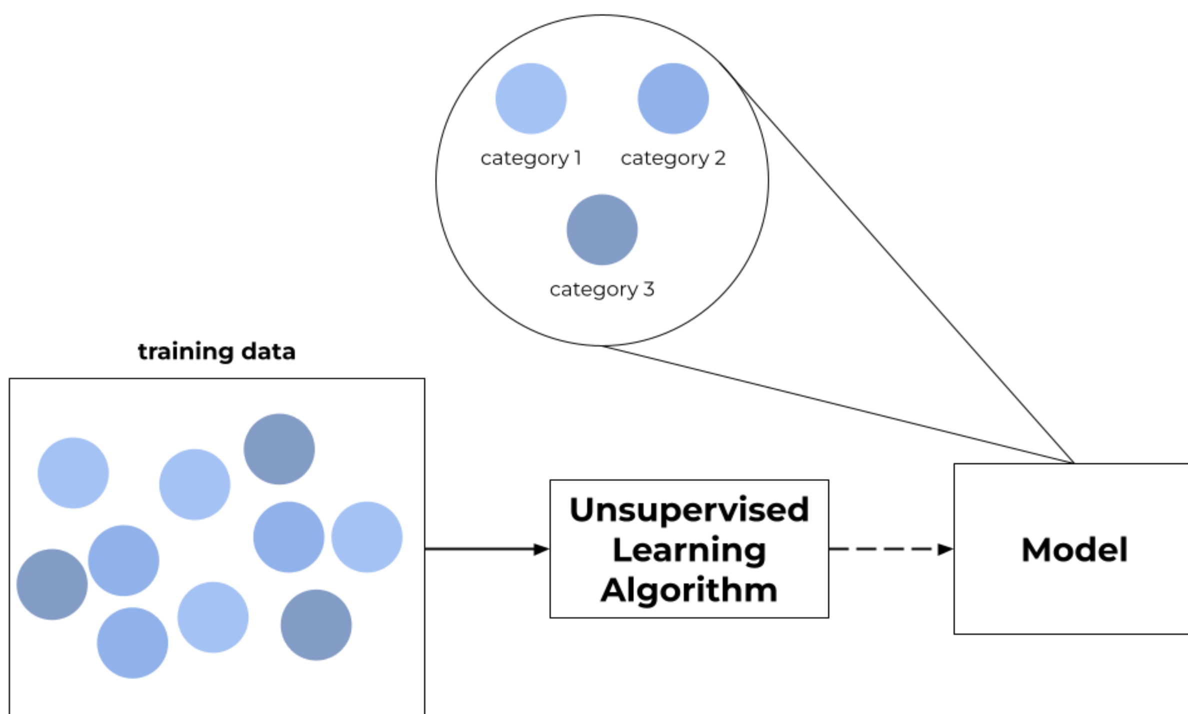


Figure 7.f. Unsupervised learning training

The prediction for unsupervised learning works similarly to supervised learning. When we have the model, we can input some data and it will classify it in a certain category from the ones that the algorithm found in the training data. As mentioned in the supervised learning section, the input data does not necessarily have to be exactly the same as the training data. The function that models the classification solution will determine to which category the input data is closer to, and will classify it as so.

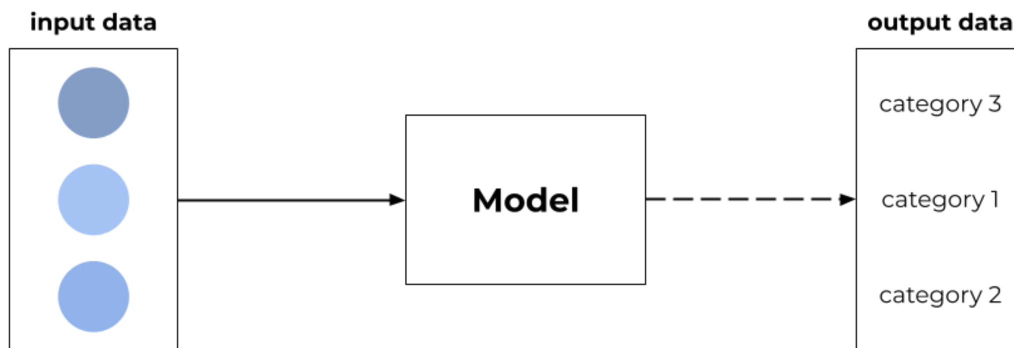


Figure 7.g. Unsupervised learning prediction.

7.3 Reinforcement Learning

Last but not least, reinforcement learning is the third major approach in machine learning. Here we are just going to briefly mention what it is useful for, because after this section, we will start a series of chapters to analyze it in a deeper way.

Reinforcement learning is useful in problems where decisions have to be made. They are what we know as decision-making problems. Reinforcement learning methods help choose the best actions to take in each situation. One of the most distinguishing features of reinforcement learning is that training data is collected while the training is being performed (there are some exceptions to that though).

In the figure 7.h we can see a problem where there are certain situations (green circles) and for each of them, we can take several actions (orange circles). As the problem faces these situations, the algorithm learns from experience and creates a model. The algorithm evaluates the benefits that it has obtained by making each decision, and with that information, it adjusts the probabilities of taking each action.

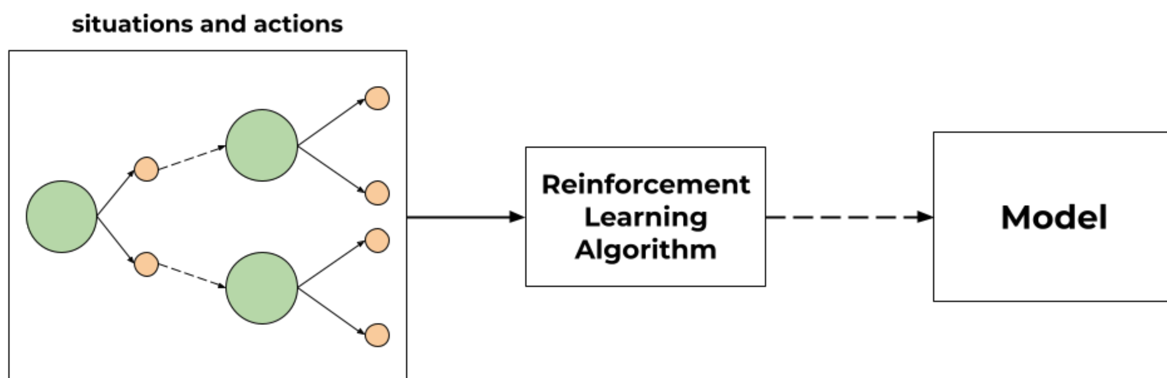


Figure 7.h. Reinforcement learning training

When having to make a decision, you can use the model and input the current situation, and it will output the action to take. The figure 7.i represents a series of situations faced and the actions that the model outputs, forming a decision path.

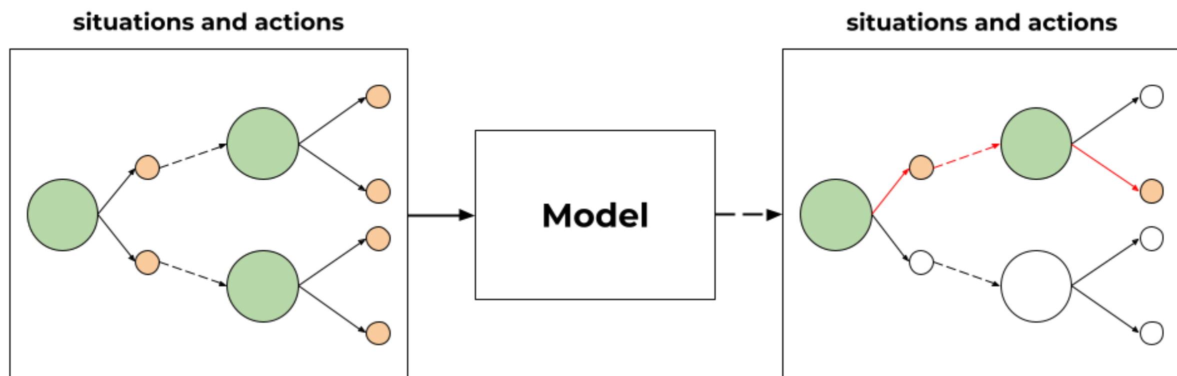


Figure 7.i. Reinforcement learning decision-making.

In the next chapters we are going to analyze the decision-making problems and how to solve them, creating models that help choose the right decision in each situation.

7.4 Summary and connection with the practical part of the project

In this chapter we have seen how we can use different techniques to give intelligence to a system. Depending on what we want to achieve, we should choose one or another. We have reviewed the three major fields in machine learning, which aim to solve different problems. If we analyze the Rocket League bot problem, we see that not all of the machine learning techniques can be used to solve it.

First, if we used a supervised learning solution, we should have training data in the form of input-output pairs, where the input is the situation of the game (location of the cars, ball, etc) and the output would be the correct action to take (like steering). There is no way to know which is the correct output given an input before the model is trained. If this situation does not occur, we cannot use a supervised learning solution.

Unsupervised learning does not make sense either, as we do not intend to find unknown patterns in data.

Reinforcement learning is the one that best fits the problem we face, as the Rocket League bot has to make decisions, so the solution to it is training a model that will tell which action to take in each situation.

The question is, why should we use reinforcement learning instead of the classic expert systems? Classic expert systems are tied to the human (the programmer) knowledge. Creating a classic expert system to implement a bot implies that the programmer needs to know what to do in each situation, so the bot takes the best action in each one. Even the best players in the world (not only in Rocket League, but any discipline) might not know the best actions to take in every one of the situations that can occur (maybe there are millions of them). Reinforcement learning has the potential to learn beyond human knowledge, as demonstrated when it was used to train a model that beat the Go world champion [12]. For this reason, reinforcement learning has been chosen to develop this project.

Chapter 8: Decision-Making Problems

8.1 Understanding the problem

Imagine we are faced with the problem of deciding the best action to take in a certain situation. Which things should we take into consideration? How can we determine if an action is better than another? If we had a way to conclude how much benefit we obtain from taking a certain action, the best one would be the one that yields the most benefit. For the rest of this text, we will refer to benefit as reward, as it is the nomenclature used in reinforcement learning. Figure 8.a is a representation of a decision-making problem, where the green nodes of the graph represent the different situations (often referred as states) that can occur in the problem, and the orange nodes are the actions that can be taken in each situation or state. When a decision is made, and the consequent action is taken, the problem transitions to another state. In each edge connecting an action node and a state node, a label of format **r=*n*** appears, where ***n*** can be any number. This number represents the reward obtained by taking that particular action and transitioning to that particular state.

In the figure 8.a you can see a simple example where the reward of taking each action is coded as a number, where the bigger the number, the bigger the reward. As you can see, the state $\mathbf{S_0}$ (that represents the situation the problem is at) has 4 possible actions. So, in this example, it would be obvious that the best action to take is $\mathbf{A_2}$ as it is the one with the biggest reward. After taking one of the four actions, the problem transitions to the state $\mathbf{S_0}$ again. In this problem there is only one situation that repeats indefinitely.

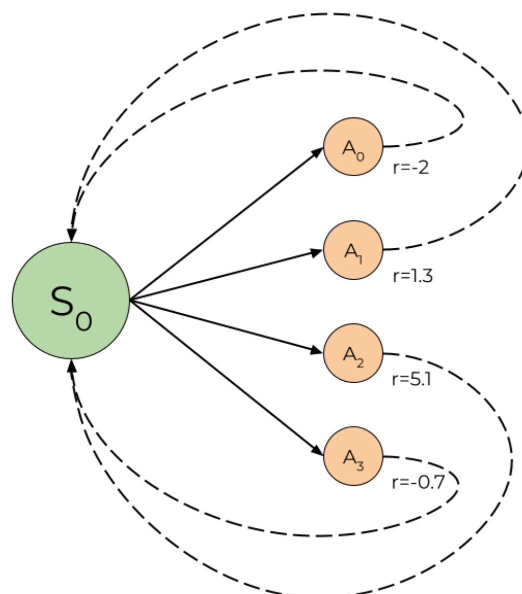


Figure 8.a. Diagram of the state S_0 and its actions.

But, is it that simple? The answer is no. Real problems are not about the actions of a single state, but finding the best action to perform in each state in order to achieve a goal in an effective and efficient way. We can consider two different types of goals: the first one is to reach a certain state of the problem where it is solved, and the second one is to keep taking actions in order to maximize the reward obtained. As an example, in the figure 8.b you can see a problem with three different states. The first one, $\mathbf{S_0}$, has the 4 same actions as before. Taking actions $\mathbf{A_0}$ and $\mathbf{A_1}$ will lead to state $\mathbf{S_1}$, while taking actions $\mathbf{A_2}$ and $\mathbf{A_3}$ will lead to state $\mathbf{S_2}$. Both $\mathbf{S_1}$ and $\mathbf{S_2}$ have two actions leading to $\mathbf{S_0}$.

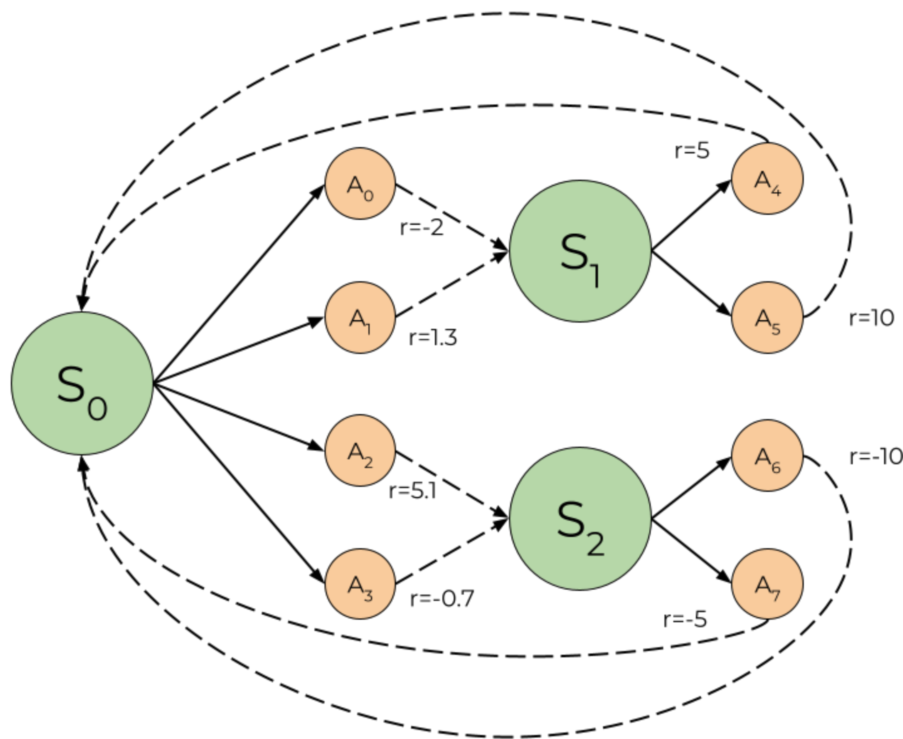


Figure 8.b. Diagram with several states and its actions.

The question now is: given a certain situation or state, is the action with the highest reward the best option? Let's imagine the problem we are trying to solve starts in the state $\mathbf{S_0}$. In the previous example, we saw that the best action to take from $\mathbf{S_0}$ was $\mathbf{A_2}$. If we took it, we would get 5.1 points of reward, but we would end up in state $\mathbf{S_2}$, which yields -5 reward points in the best case and leads to $\mathbf{S_0}$, a terminal state. So, if we followed the action in $\mathbf{S_0}$ that yields the most reward ($\mathbf{A_2}$) we would end up in $\mathbf{S_2}$ and getting 5.1 reward points, and in $\mathbf{S_2}$ we would take the action $\mathbf{A_7}$ (following the same criteria of choosing the action that yields the most reward), getting -5 reward points, adding up to a total of 0.9 reward points. After this, we would be back at $\mathbf{S_0}$.

Obviously, this is not the best option in order to maximize the reward obtained. The conclusion is that, for a certain state, maximizing the immediate reward (that is, the reward obtained for taking an action, without taking into account the future rewards as a consequence of taking that action) does not necessarily mean that the total reward will be maximized.

So, in this example, the best option would be to choose action $\mathbf{A_1}$ in the state $\mathbf{S_0}$, as this way, we would get 1.3 points as immediate reward, and from then on choose action $\mathbf{A_5}$ from state $\mathbf{S_1}$, getting 10 reward points and ending up in the state $\mathbf{S_0}$ again. This adds up to 11.3 reward points, way better than the reward obtained (0.9) just following the action with the most immediate reward. In this problem it is so easy to see which are the best actions to choose in each case, as there are only three states and a total of eight actions, but in bigger problems, with thousands of states, it can be quite difficult.

8.2 A practical example

In the last chapter we learned that decision-making problems are more complicated than just choosing the action that gives more immediate reward. Now we are going to see how this problem occurs in real life, and how it can be represented and treated the same way we did with the graphs in the figures above. Of course, this first example is so simple that it would not need to be solved using reinforcement learning (later on we will see how decision-making problems can be solved using reinforcement learning), but it is useful in order to see how a real problem can be characterized as the type of problems we have seen.

In this problem, we have six different cells. Imagine that a robot has the goal of reaching the green cell making as few movements as possible, while reaching the red one means failing its mission. The movements the robot can make are determined by the arrows in each cell, so the decision-making problem that we face is determining, for each cell, which is the best movement to make.

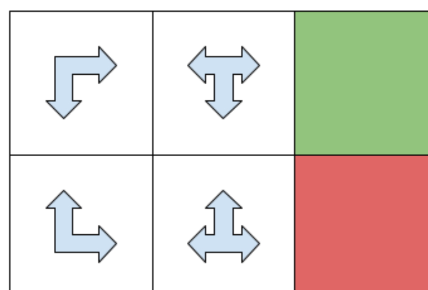


Figure 8.c. Decision-making problem.

The decision-making problem above could be represented as the graph in the figure 8.d. Each state represents a cell and each action one of the movements the robot can make from that cell. The rewards are not directly defined by the problem rules, but they have to make sense and help characterize it. The goal is that the robot reaches the green cell, so it makes sense that the action that makes it reach that cell yields the most reward. Moving to the red cell has to be the action that yields the most negative reward. The problem says that the goal is to reach the green cell in as few movements as possible, so it makes sense to give negative rewards for each action (that does not get the robot to the green cell) as we want to minimize these movements by minimizing the negative reward (and maximizing the reward).

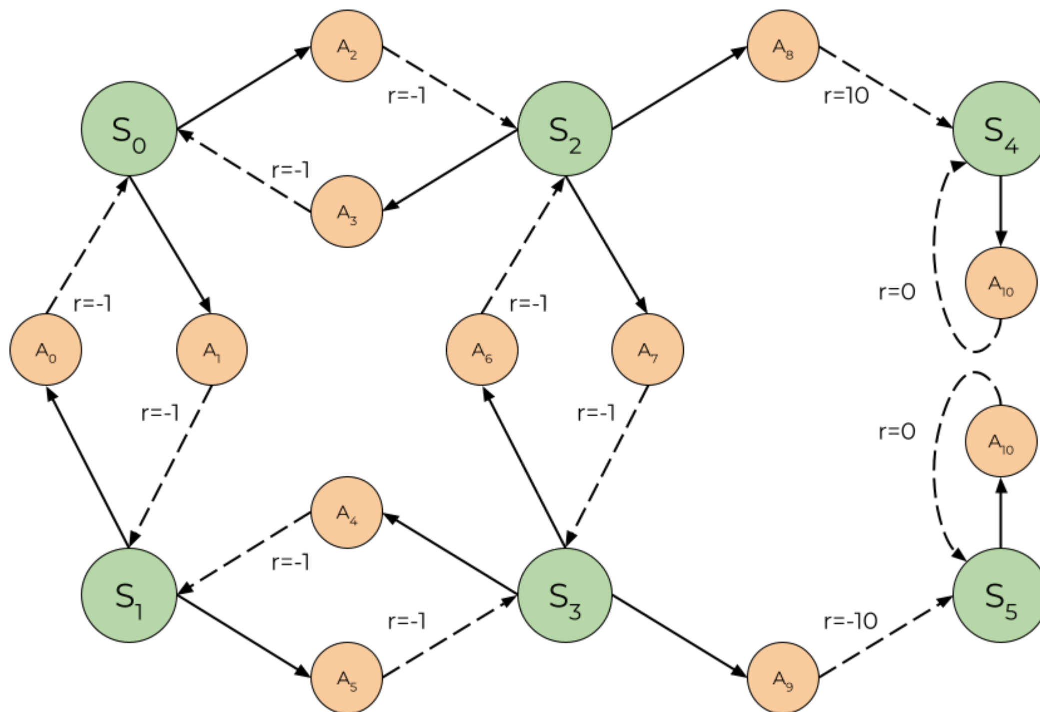


Figure 8.d. Decision-making problem graph.

8.3 Markov decision process (MDP)

Markov decision processes (MDP's) [13] are mathematical frameworks to model decision-making problems where outcomes are partly random and partly under control of the decision maker. Problems that have been analyzed up until this point can be modeled as MDP's, and these MDP's are usually represented with graphs similar to the ones shown in the previous chapters. This process works by time steps. At each time step, the process is in a certain state **s**, and then the decision maker chooses an action **a** available in the state **s**. Then, the process moves to a new state **s'** and returns a reward **R(s,a,s')** to the decision maker. In the previous examples, all pairs state-action mapped to one state, but that is not always the case. Sometimes, a state-action pair can map to more than one state, with a certain probability each. Of course, for each state-action pair, all the probabilities of all the possible resulting states must add up to 1. An example of this can be seen in the figure 8.e.

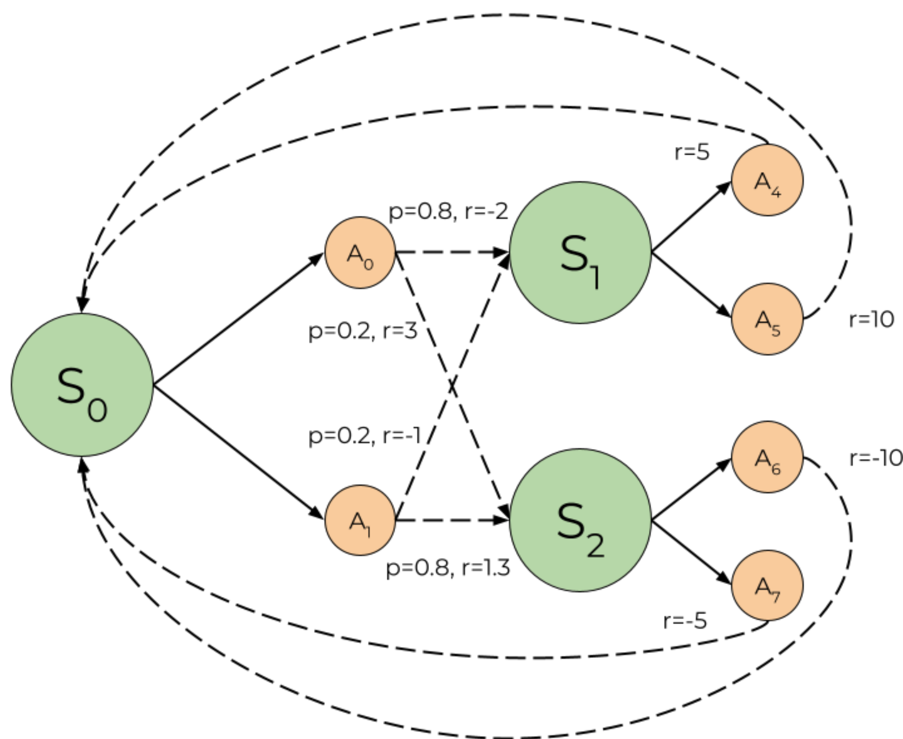


Figure 8.e. Example of a MDP graph.

Another important aspect of MDP's is that the probability of the process transitioning to the state **s'** only depends on the current state **s** and the action **a** taken. It is given by the state transition function **P(s'|s,a)**. Past states of the process and actions taken before **a** do not affect this probability. This is known as the **Markov property**.

We have seen an intuition of what MDP's are and how they can be represented, but now we are going to see which elements they have. An MDP is 4-tuple(S, A, P, R) where:

- **S** is the set of states, also known as the **state space**
- **A** is the set of actions, also known as the **action space**
- **P** = **P(s'|s,a)** = $\mathbf{P(s_{t+1}=s'|s_t=s, a_t=a)}$ is the probability that the action **a** in the state **s** at time step **t** leads to the state **s'** at time step **t+1**
- **R(s,a,s')** is the immediate reward obtained after transitioning from state **s** to **s'** due to the action **a**.

These elements define decision-making problems in the framework of a Markov Decision Process, but still a fifth element is needed in order to define a solution to the problem. This element is what is known as the policy function, often referred to with the letter **π**. Policy functions map the state space to the action space, which means that for each state, it indicates which action to take. Formally, **π** is a mapping from the state space to a probability. If the policy **π** is being used by the decision maker, **π(a|s)** is the probability that the action **a** is chosen if the MDP is at the state **s**, given that $\mathbf{a_t \in A(s_t)}$.

Any policy function is valid to solve the problem, in the sense that any arbitrary policy gives a valid mapping of the state space to the action space so the process can correctly transition between states. But the objective of the methods that try to solve MDP's is not to create a valid policy, but obtain a policy that maximizes the reward yield by the process.

Now that we know what an MDP is, we are going to see how the process develops. Two elements can be highlighted: the agent and the environment. The agent is the decision maker and learner, in case that the goal of running the process is to learn an improved policy. The environment is everything else: responsible for transitioning between states, yielding rewards, etc. The agent and the environment interact with each other, as seen in the figure 8.f.

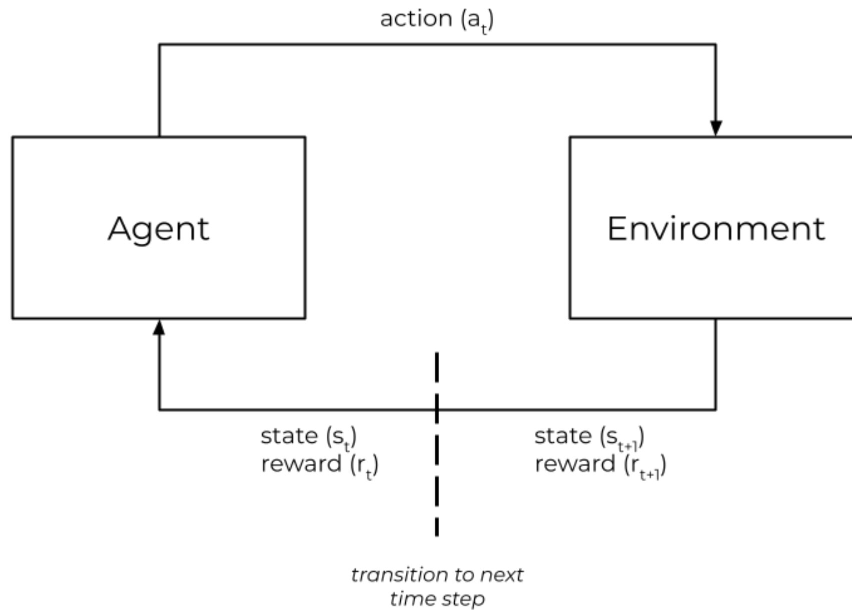


Figure 8.f. Interaction between the agent and the environment in a MDP.

At each time step, the agent receives a state $\mathbf{s_t} \in \mathbf{S}$, and based on some criteria, it chooses an action $\mathbf{a_t} \in \mathbf{A(s_t)}$. The environment will then, as a response, send a new state $\mathbf{s_{t+1}}$ and a reward $\mathbf{r_{t+1}}$ depending on the state $\mathbf{s_t}$ and the action $\mathbf{a_t}$. This creates a sequence like:

$$s_0, a_0, r_1, \ s_1, a_1, \ r_2, \ s_2, a_2, \ r_3, \ s_3, a_3, \ldots$$

Time steps have been mentioned throughout the chapter, but now it is time to focus on them. Even if they can, time steps are not necessarily connected with physics time. Time steps are moments where a decision has to be made, and the consequent state transition occurs. Executions of the MDP are called episodes. Episodes consist of several time steps. Episodes can last a fixed amount of time steps, or until the goal is reached, in case that there's a state considered as goal. These states are called terminal states.

8.4 Returns: when immediate rewards are not enough

Of course, decision-making problems are modeled in order to find a way to make the most optimal decisions. For this reason, MDP's are used as the way to create an optimal policy that, ideally, gives the best action for each state. In MDP's, improving the policy consists in maximizing the reward obtained. As shown earlier, immediate rewards do not necessarily mean that the action that yields the most immediate rewards is the best one. For this reason, it is important to introduce the **return**. In its simplest form, the return in a time step **t** is the sum of all the rewards obtained in the following time steps, as shown in the formula:

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \ ... \ + r_T$$

where **T** is the time where a terminal state is reached.

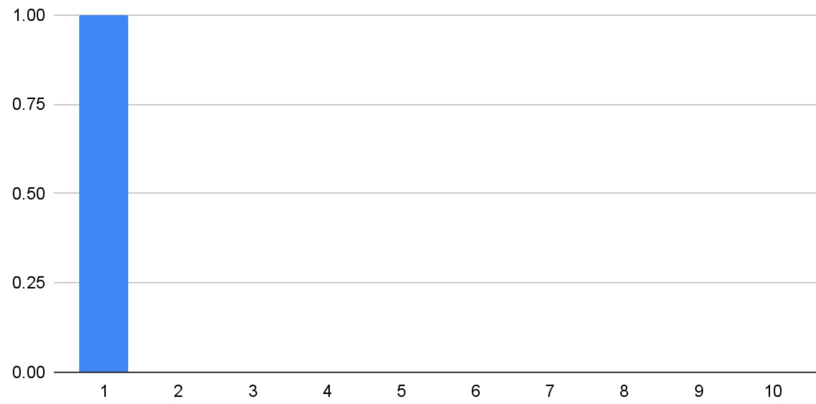
As was mentioned earlier, there are two types of problems: the ones that break naturally in episodes (as for example a match of chess) or the ones that are infinite. The fact that some episodes might be infinite creates the necessity of using a more complex type of return function:

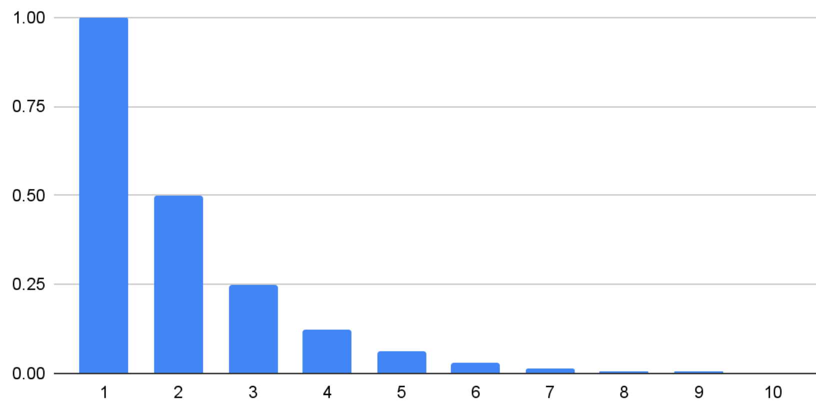
$$G_t = r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \ ...$$

where γ is between 0 and 1. It can be generalized as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

This guarantees that, in the case of having an episode with infinite time steps, the rewards further away in time will have less weight, making these distant rewards tend to 0. γ is a discount factor. In case that the discount was not needed, γ would be equal to 1. So the closer to 1 that γ is, the more important the future rewards are. In the case that γ = 0, only the immediate reward matters. For some problems, immediate rewards might be quite important, and for others, the most important reward is yield in the long run.





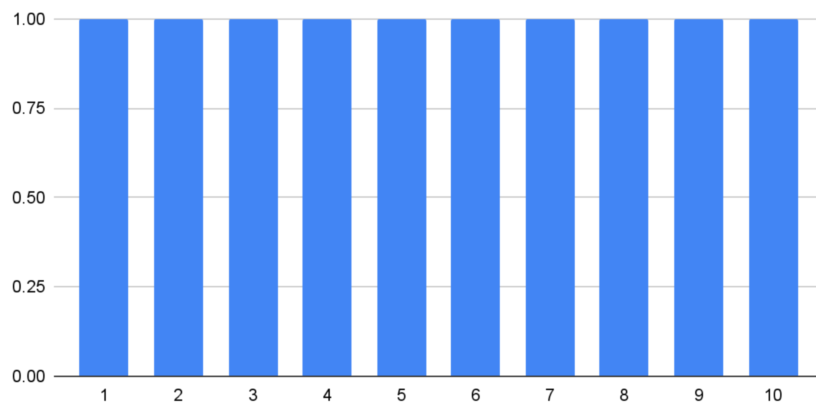


Figure 8.g. Returns on time steps following the time step **t**, with γ=0.0, 0.5 and 1.0.

In the figure 8.g can be seen how the parameter γ affects the returns for an action **a** taken at time step **t**. All actions in the MDP yield 1 reward point of immediate reward. In the first plot, γ = 0.0, so the return is only the immediate reward of $\mathbf{a_t}$. In the second plot, γ = 0.5, and the further the time step, the less reward it yields as a return for G_t. With this parameter, the return tends to 2, which is the double of the immediate reward. In the third plot, γ = 1.0, so all the time steps after **t** yield 1.0 of reward points as a return for the time step **t**.

8.5 Value function, action-value function and Bellman equation

When trying to optimize a policy, it is important to estimate how good it is to be in a certain state. Using the concept seen in the previous chapter, an estimate of the return can be utilized to have an idea of how good it is to be in a certain state. This is known as the **expected return**, or **value function**. The value function of the state **s** when following the policy π is:

$$v_\pi(s) = E_\pi\left[G_t \mid s_t = s\right] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

There is also what is known as action-value function. It is similar to the value function, but it is useful to estimate how good it is to take the action **a** when in the state **s** and following the policy π:

$$q_\pi(s, a) = E_\pi\left[G_t \mid s_t = s, a_t = a\right] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right]$$

With this, we almost have all the elements to be able to improve the policies that solve an MDP. As said, value functions and action-value functions are estimates of the returns, which means that there is a way to progressively improve these estimates, so they get closer to the real return.

As rewards in the returns, value functions are recursive, in the sense that value functions of a certain state **s** can be calculated from the value functions of the states accessible from **s**. The following formula is used to calculate the value function of a state **s**:

$$v_\pi(s) = E_\pi\left[G_t | s_t = s\right] = E_\pi\left[r_{t+1} + \gamma G_{t+1} | s_t = s\right] =$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a)\left[r + \gamma v_\pi(s')\right], \forall s' \in S,$$

This is known as the **Bellman equation**. To understand how this equation works, we have to remember that every state has one or more actions, and each of these actions has a probability associated. The probability associated to the action **a** in the state **s** is π(a|s), that comes determined by the policy. Then, every action **a** in state **s** has one or more states **s'** which the MDP can transition to when taking the action **a**. Each of these states **s'** has a probability associated, that indicates how probable is that the MDP transitions to it when taking the action **a** in the state **s**. This probability is **P(s'|s,a)**.

The Bellman equation says that, as seen in the figure 8.h, to calculate the value function of a state **s**, for every state **s'** it can transition to, we have to multiply the probability of taking the action **a** that leads to it by the probability of that action **a** leading to **s'**. The result of multiplying these two probabilities has to be multiplied by the sum of the reward yield when taking the action **a** in the state **s** and transitioning to the state **s'** with the discounted value function (using γ) of the state **s'**. So then, these results obtained (for each **s'**) have to be added up.

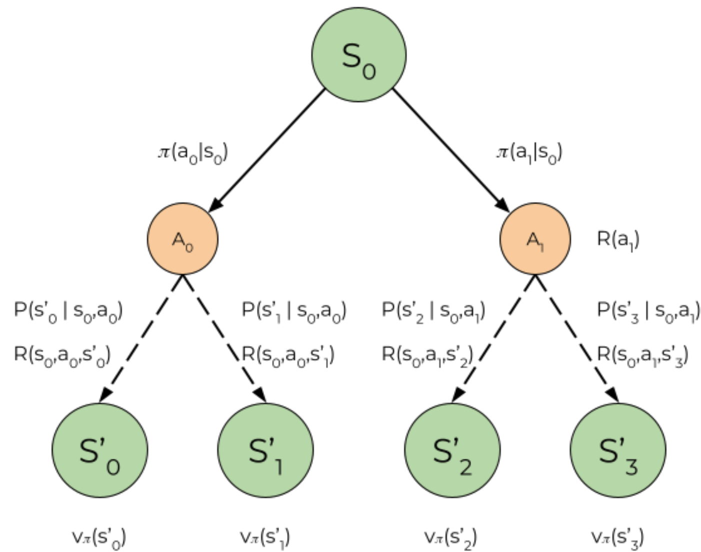


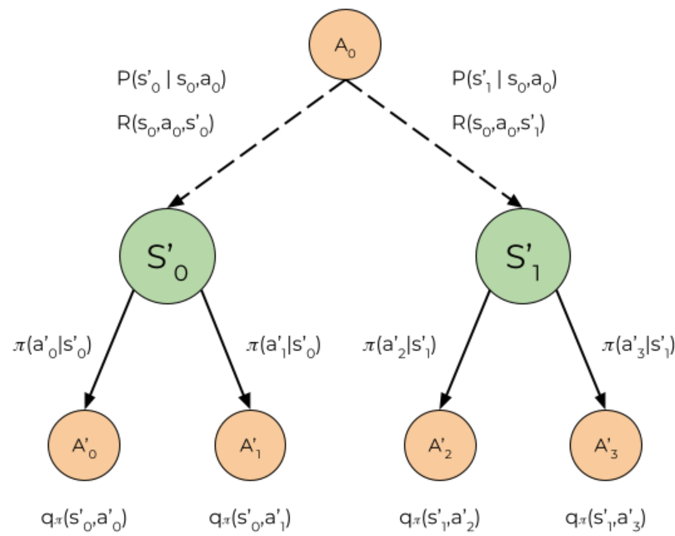
Figure 8.h. Representation of the Bellman equation to calculate value functions.

We also have the Bellman equation to calculate the action-value function:

$$q_{\pi}(s, a) = E_{\pi}\left[G_t | s_t = s, a_t = a\right] = E_{\pi}\left[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a\right] =$$

$$= \sum_{s',r} p(s', r | s, a) \sum_{a} \pi(a'|s')\left[r + \gamma q_{\pi}(s', a')\right], \forall s' \in S$$

The figure 8.i gives a graphical idea of how the Bellman equation to calculate action-value functions work.



$$q_{\pi}(s_0,a_0) =$$

$$= P(s'_0 | s_0,a_0) \cdot \pi(a'_0|s'_0) \cdot (R(s_0,a_0,s'_0) + \gamma\, q_{\pi}(s'_0,a'_0)) +$$

$$+ P(s'_0 | s_0,a_0) \cdot \pi(a'_1|s'_0) \cdot (R(s_0,a_0,s'_0) + \gamma\, q_{\pi}(s'_0,a'_1)) +$$

$$+ P(s'_1 | s_0,a_0) \cdot \pi(a'_2|s'_1) \cdot (R(s_0,a_0,s'_1) + \gamma\, q_{\pi}(s'_1,a'_2)) +$$

$$+ P(s'_1 | s_0,a_0) \cdot \pi(a'_3|s'_1) \cdot (R(s_0,a_0,s'_1) + \gamma\, q_{\pi}(s'_1,a'_3)) +$$

Figure 8.i. Representation of the Bellman equation to calculate action-value functions.

8.6 Summary and connection with the practical part of the project

In this chapter we have seen what decision-making problems are and how we can formalize them as Markov Decision Processes. We explained how these processes consist in an agent that interacts with the environment, taking actions and receiving a state and a reward, which indicates how good it was to take that action in that state. We've seen that a run of this process is called an episode, and that a return is the reward obtained from a certain point in the episode onwards. Under this MDP framework, solving a decision-making problem consists of maximizing the returns obtained. The last thing seen was that the expected return (value) of a state of a state-action pair can be updated using the values of the states or state-action pairs the environment can transition to.

Rocket League bots are in fact a decision-making problem that can be formalized using an MDP. Its states are given by the things such as the coordinates of the cars and ball in the field, their speeds, etc, and the actions of the bot could be throttling, steering, etc. The bot can interact with the environment by sending it those actions and receiving the new state (for example, a state with the updated car coordinates after throttling) and a reward. In the next chapters we will see the theory on how to solve these decision-making problems, which will lay the foundation for solving the Rocket League bot problem.

Chapter 9: Policy optimization with Dynamic Programming

Now we have all the elements in order to optimize policies for MDP's. Reinforcement learning methods are so useful to do so, but in some cases, it is not necessary to use such methods, as it can be achieved using dynamic programming.

For some decision-making problems, all the dynamics of the MDP will be known, which means that all the transitions between states and probabilities are known. But for the most part, they are not. When there is a full understanding of the MDP dynamics, it is possible to optimize the policy by using dynamic programming, and when there is not, reinforcement learning comes to play. So, before we start analyzing reinforcement learning methods for optimizing policies, we are going to see how to optimize them for problems using dynamic programming where the dynamics of the MDP are known. When these dynamics are fully known, we say that we have a complete model of the MDP. In this section, the MDP's are assumed to be finite, in the sense that both the state and the action spaces are finite.

9.1 Policy evaluation

The first step is to know how to evaluate a policy. Policy evaluation consists in calculating $v_\pi(s)$, $\forall s \in S$, when the policy π is followed, ideally finding the true value of the returns, for each state. These value functions are calculated using the Bellman equation, and starting with arbitrary value functions. There are ways to start with value functions that are closer to the true returns, but it will not be treated in this text. So we assume that the starting value functions are arbitrary. Terminal states should have starting value functions of 0.0.

This is an iterative process where we have an array of value functions. At the start of the algorithm, the array is filled with the arbitrary value functions. In each iteration, for each state **s**, we calculate its new value function from its successors, using their value functions from the array, and the new value obtained is stored in the array. In the figure 9.a it can be seen that, to calculate the new value function for state $\mathbf{s_0}$, the old value functions of its successor states (s_1 and s_2) are used.

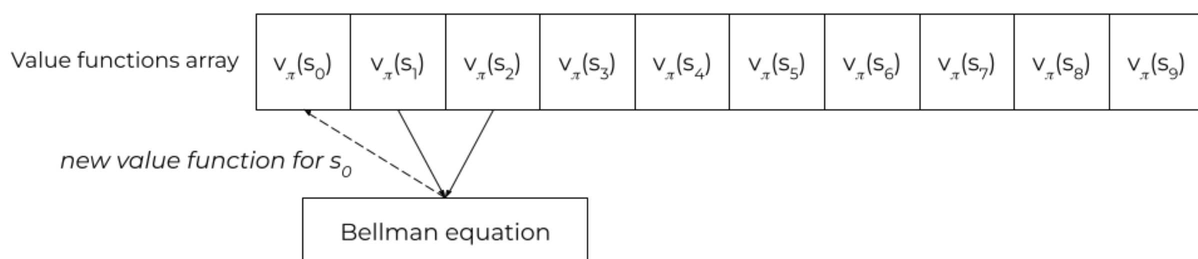


Figure 9.a. Calculation of the new value function for a state during policy evaluation.

It would be ideal to find the true returns for all the states, but it often has a high computation cost. For this reason, the algorithms usually stop when there is not much change in the value functions between iterations, which means that the value functions obtained are close to the real values. In the figure 9.b you can see the pseudocode for the policy evaluation using dynamic programming. Δ is the maximum difference between the value functions of an iteration and the previous one. When this difference is smaller than **θ**, the execution stops.

```
input S, A, P, R, π, γ, θ
array v(s) ← 0, for each s in S

do:
    Δ ← 0
    for each s in S:
        v ← v(s)
        v(s) ← Σ_a π(a|s) Σ_{s',r} p(s', a | s, a)[r + γv_π(s')]
        Δ ← max(Δ, |v - v(s)|)
while Δ >= θ (where θ is a small positive number)
```

Figure 9.b. Pseudocode for Dynamic Programming policy evaluation. Based on [14]

In the problem in the figure 9.c, we have a gridworld, where each state has actions to move to the cells indicated by the arrows. Each action yields -1 reward points. Green cells represent terminal states.

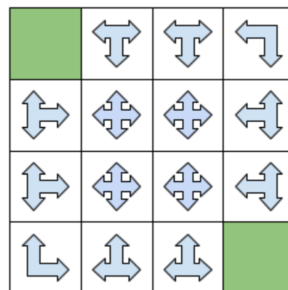


Figure 9.c. Example of policy evaluation. Based on [15]

We are going to execute several iterations of the policy evaluation algorithm, where the policy is equiprobable and where all the value functions start at 0.0. In the figure 9.d we can see how the value functions evolve. As we can see, the states closer to the terminal state had smaller value functions as in its calculation they use the value function of the terminal state (which is always 0). If you analyze the problem, you could expect the value function of the states that connect with terminal states to tend to -1.0, as they are one step away from the goal. Why do they end up with a value function of -14.0? This occurs because, even if we are getting the value functions close to the real returns, we are not using it to improve the policy. The policy still gives an equal probability for each action, when the states adjacent to the terminal states have clearly one good action, that should have 1.0 probability in the optimal policy (it is clear in this problem because it is so simple. In real problems, it is not as easy to see). In the next chapter, we will see how to improve the policy using this policy evaluation.

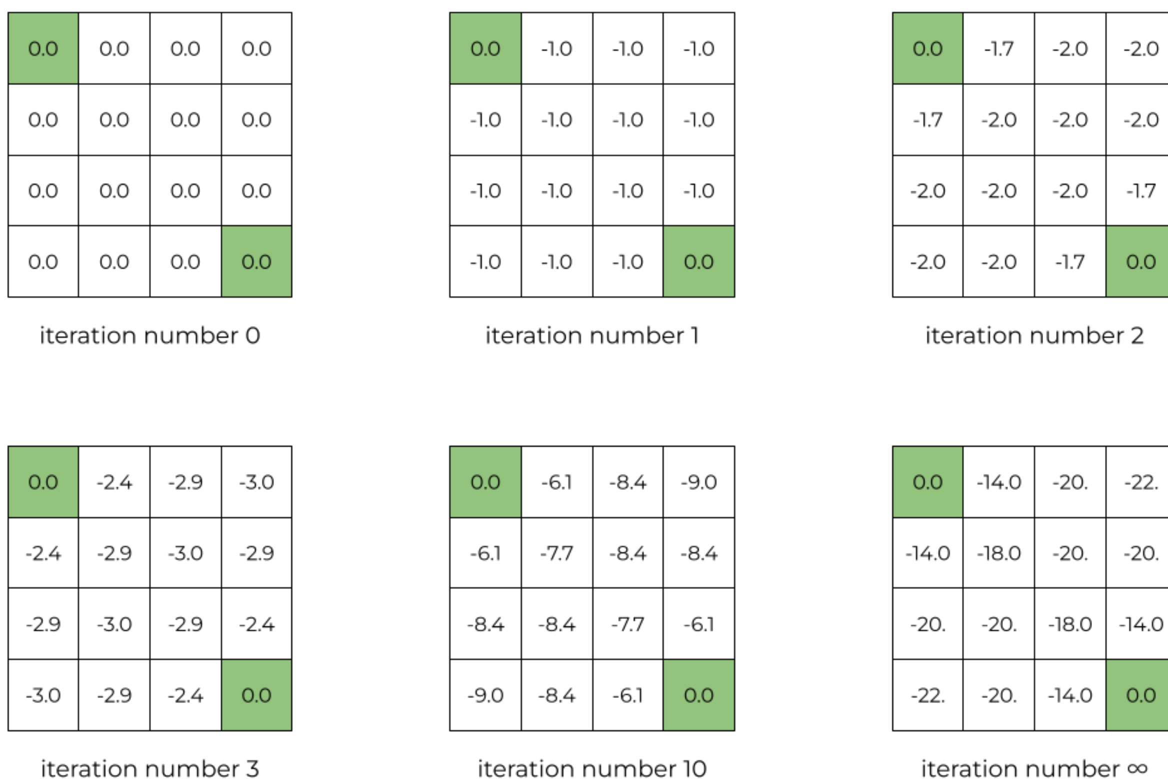


Figure 9.d. Resolution of example of policy evaluation. Based on [16]

9.2 Policy improvement

When the policy has been evaluated, it is possible to improve it. To evaluate the policy, we have used the Bellman equation to calculate the value functions of the states, but in the previous chapter we mentioned that there also exists a Bellman equation to calculate the action-value function, that quantifies how good it is to take an action **a** given a state **s**. It makes sense that, for a state **s**, the action with the biggest action-value function is the one that should be given the whole probability in the policy, as it is the one with the most expected returns.

In the pseudocode in the figure 9.e we can see how, for each state, we update the policy with the action with the biggest action-value function.

```
input S, A, P, R, π, γ, θ, v
for each s in S:

    π(s) ← argmax_a( Σ_{s',r} p(s',r | s,a)[r + γv_π(s')])
```

Figure 9.e. Pseudocode for Dynamic Programming policy improvement.

9.3 Dynamic Programming Control

Control is the process used to repeatedly optimize the policy by using the policy evaluation and policy improvement techniques. As seen in the figure 9.f, the process consists in first executing the policy evaluation and then the policy improvement. When one iteration of the process finishes, we check whether the policy is stable or not. The policy being stable means that the policy did not change in any of the states. In that case, the Dynamic Programming control ends.

```
input S, A, P, R, π, γ, θ
array v(s) ← 0, for each s in S
do:
    do:
        Δ ← 0
        for each s in S:
            v ← v(s)

            v(s) ← Σ_a π(a|s) Σ_{s',r} p(s', a | s, a)[r + γv_π(s')]
            Δ ← max(Δ, |v - v(s)|)
    while Δ >= θ (where θ is a small positive number)

    policy_stable ← true
    for each s in S:
        a ← π(s)

        π(s) ← argmax_a(Σ_{s',r} p(s', r | s, a)[r + γv_π(s')])
        if a ≠ π(s) then policy_stable ← false
while not policy_stable
```

Figure 9.f. Pseudocode for Dynamic Programming control.

9.4 Summary and connection with the practical part of the project

In this chapter we have seen how we could optimize a policy in order to get the maximum returns from a decision-making problem formalized as an MDP. However, this solution only applies to problems where all the dynamics from the MDP are known, so it's used as a theoretical tool more than as a practical solution. For this reason, while this will help us understand how reinforcement learning solutions work, it cannot be applied to solve the Rocket League bot problem.

Chapter 10: Reinforcement learning: tabular solution methods

In this chapter we will see reinforcement learning methods for solving finite MDP's. In this case, we will treat small problems, the data of which can be saved in arrays, as we did with the value functions in dynamic programming. These are tabular problems, solved by tabular solution methods.

When the dynamic programming solution was presented, we had a complete model of the MDP, which let us calculate all the value functions directly from the model. In this case, we do not have that model, and that is when reinforcement learning methods come into play. These methods that do not use a model of the MDP are called **model-free reinforcement learning methods**. Some reinforcement learning methods try to learn the model before optimizing the policy. Those methods are called **model-based reinforcement learning methods**. In this text we will focus on model-free methods.

Unlike in dynamic programming, in reinforcement learning methods the agent learns through experience, which means its interaction with the environment. From this interaction, the agent gets certain rewards for taking certain actions in certain states, which can be used to estimate the value functions or action-value functions. In this chapter we are going to talk about two different types of methods for tabular problems: **Monte Carlo methods** and **Temporal-Difference learning**. But first of all, we are going to talk about a recurring problem in reinforcement learning: exploration vs exploitation.

10.1 Exploration vs Exploitation

Exploration vs exploitation is the problem of deciding whether to keep improving the estimation of the biggest value function or action-value function for a certain state or state-action pair, or explore the smaller ones in hopes that with further estimation improvement, they will become bigger than the current one.

In the dynamic programming method, we followed what is known as a **greedy policy**. A greedy policy gives all the probability to the action with the most expected return, and 0 to all the others. This makes sense because, if you have to choose an action, you would choose the one that will yield the most reward in the long run. While this is correct for dynamic programming, it can cause problems when optimizing policies with reinforcement learning methods.

When doing the policy evaluation with a reinforcement learning method, some actions will get a bigger action-value function than others. Then, if we follow a greedy policy, we will update that policy to give that action all the probability. For this reason, consequent iterations of the policy evaluation will always choose that same action. But we have to ask ourselves: was that policy evaluation iteration representative enough to determine that it is the best action? The answer is no, because that iteration is only a sample, but future experiences could decrease its expected reward and increase the other's. For this reason, it is reasonable to not give all the probability to only the action with the most action-value function, but also give a little bit to all the other actions. This will ensure that these other actions will eventually be explored, giving them the opportunity to improve their estimates and potentially discover an action with a bigger return.

One of the most basic and important strategies to obtain a good trade-off between exploration and exploitation is to use what is known as an **ε-greedy policy**. In ε-greedy policy, the greedy action (the one with the most action-value function) gets the most probability, but the other ones also get some to ensure exploration. The parameter **ε** dictates how much probability the non-greedy actions (combined) get. **|A(s)|** is the number of actions the state **s** has available. Every one of the non-greedy actions will get **ε / |A(s)|** probability, while the greedy action will get **1 - ε + (ε / |A(s)|)**.

So, if we consider **A*** as the greedy action, the general policy update rule would be as follows:

$$\pi(a|s) \leftarrow \left\{ \begin{array}{ll} 1 - \varepsilon + (\varepsilon \,/\, |A(s)|) & if\ a = A^{*} \\ \varepsilon \,/\, |A(s)| & if\ a \neq A^{*} \end{array} \right\}$$

In the next sections we are going to see the reinforcement learning methods mentioned in the chapter introduction, where this ε-greedy policy is used.

10.2 Policy optimization using Monte Carlo methods

In Monte Carlo methods, the value functions are calculated by sampling episodes. The agent interacts with the environment and generates episodes. As mentioned in previous chapters, episodes are sequences as:

$$s_0, a_0, r_1, \ s_1, a_1, \ r_2, \ s_2, a_2, \ r_3, \ s_3, a_3, \ldots, s_{T-1}, a_{T-1}, r_T$$

With this, for each time step, we have the reward obtained, the state, and action taken.

In Monte Carlo, action-value functions are usually used instead of value functions. This is caused by the fact that, if we do not know the dynamics of the environment, we cannot know which action to take to transition to a certain state, or we do not even know the successor states for a certain state **s**. But if we know, for a certain state **s**, the action that has more value, we can update the policy in that direction.

In Monte Carlo methods, an entire episode is generated before it is used to update the action-value functions. These kinds of methods are called **offline methods**. Contrary, **online methods** work with data as soon as it is available. In this case, Monte Carlo methods are offline, as they use a batch of data to work with, in this case, an episode. In next chapters we will discuss the difference between online and offline methods and the pros and cons of each one of them.

Monte Carlo policy evaluation

As we did in the dynamic programming chapter, we are going to first take a look at policy evaluation. As seen in the figure 10.a, two arrays are used. The first one stores the action-value functions for every state-action pair, and the second one stores all the returns obtained for each state-action pair. If we consider an arbitrary policy, we have to generate episodes by following that policy. Every time we generate an episode, we loop through all of its time steps. For each one of these time steps (starting for the last time step of the episode), the accumulated returns are calculated. If we are in time step t, this is a sample of the returns of time step **t**, ($\mathbf{G_t}$, as seen in chapter 8). This is then a sample of the expected return for taking the action $\mathbf{a_t}$ in the state $\mathbf{s_t}$. This sampled return is then appended to the array of returns for the state-action pair corresponding to $\mathbf{s_t}$ and $\mathbf{a_t}$, and all the returns from that pair are averaged, to calculate the new action-value function.

```
input S, A, π, γ
array q(s,a) ← 0, for each s,a in S,A
array returns(s,a) ← empty, for each s,a in S,A

do forever:
    episode ← generate_episode()
    episode_returns ← 0
    for each t in episode (starting for T-1):
        episode_returns ← r_t+1 + γ·episode_returns
        append episode_returns to returns(s_t,a_t)
        q(s_t,a_t) ← average(returns(s_t,a_t))
```

Figure 10.a. Pseudocode for Monte Carlo policy evaluation.

Monte Carlo control

Given the usage of action-value functions, Monte Carlo control is easy. After evaluating the policy, we can improve it. For a given state, we use the ε-greedy policy, giving it the most probability to the greedy action, and a little bit to the rest, to ensure exploration. In Monte Carlo, the policy is updated once per episode, after updating all the action-value functions. We have the complete Monte Carlo method in the figure 10.b:

```
input S, A, π, γ
array q(s,a) ← 0, for each s,a in S,A
array returns(s,a) ← empty, for each s,a in S,A

do forever:
    episode ← generate_episode()
    episode_returns ← 0
    for each step in episode (starting for T-1):
        episode_returns ← r_t+1 + γ·episode_returns
        append episode_returns to returns(s_t,a_t)
        q(s_t,a_t) ← average(returns(s_t,a_t))
    for each s in episode:
        A* ← argmax_a(q(s,a))
        π(a|s) ← { 1 − ε + (ε / |A(s)|)   if a = A*
                 { ε / |A(s)|             if a ≠ A*
```

Figure 10.b. Pseudocode for Monte Carlo control.

10.3 Policy evaluation using Temporal-Difference learning

Temporal-Difference learning [17] is a family of methods that sample from the environment and update its estimates (the value function) based on the estimates of the other states (in this case, of the successor state in the episode). For this reason, it is sometimes said that TD (as it is usually called) is a mix between dynamic programming and Monte Carlo methods. Another difference with Monte Carlo is that TD updates the policy in every time step instead of waiting for the episode to end. For this reason, TD methods are online.

Value function Temporal-Difference policy evaluation

So, in policy evaluation, the general rule to update the value functions is the one that follows:

$$v(s_t) \leftarrow v(s_t) \; + \; \alpha\left[r_{t+1} + \; \gamma \cdot v(s_{t+1}) \; - \; v(s_t)\right],$$

$$where\; 0 \; \leq \; \alpha \; \geq \; 1$$

In it, the value function of the state $\mathbf{s_t}$ of the current time step is updated taking into account the value function of the next step of the episode. To the old value function of the state $\mathbf{s_t}$, a quantity is added. This quantity is a fraction of the difference between the expected return $(r_{t+1} + \gamma \cdot v(s_{t+1}))$ taking into account the reward in the next time step and the discounted value function of the next state in the episode, and the current value function of the state $\mathbf{s_t}$. There is a parameter α is what is known as the **step size**. It determines the rate of learning. With a step size of 0, no new information will be learned. With a step size of 1, all the new information replaces the old one.

The formula above is for a specific TD method called TD(0), which is a special case of TD(λ). TD(0) will be the only one studied in this introduction text. In the figure 10.c we can see the pseudocode for the TD(0) policy evaluation:

```
input S, A, π, γ, α
array v(s) ← 0, for each s in S

for each episode:
    for each step in episode:
        aₜ ← π(sₜ)
        rₜ₊₁,sₜ₊₁ ← get_next_time_step(aₜ)
        v(sₜ) ← v(sₜ) + α[rₜ₊₁ + γ·v(sₜ₊₁) − v(sₜ)]
        sₜ ← sₜ₊₁
        if sₜ is terminal then break
```

Figure 10.c. Pseudocode for TD(0) policy evaluation using value functions.

Action-value function Temporal-Difference policy evaluation

We also have the version of the formula and pseudocode that use action-value functions:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) \ + \ \alpha\left[r_{t+1} + \gamma \cdot q(s_{t+1}, a_{t+1}) - q(s_t, a_t)\right],$$

$$where\ 0 \leq \alpha \ \geq 1$$

```
input S, A, π, γ, α
array q(s,a) ← 0, for each s,a in S,A

for each episode:
    aₜ ← π(s₀)
    for each step in episode (starting from t=1):
        rₜ₊₁,sₜ₊₁ ← get_next_time_step(aₜ)
        aₜ₊₁ ← π(sₜ)
        q(sₜ) ← q(sₜ,aₜ) + α[rₜ₊₁ + γ·q(sₜ₊₁,aₜ₊₁) − q(sₜ,aₜ)]
        sₜ ← sₜ₊₁
        aₜ ← aₜ₊₁
        if sₜ is terminal then break
```

Figure 10.d. Pseudocode for TD(0) policy evaluation using action-value functions.

There are two main ways of doing TD control: **SARSA** and **Q-learning**. SARSA is what is known as an **on-policy method**. On-policy methods [18] are the ones that use the policy being optimized to generate experience. In **off-policy methods** instead, we have the policy being optimized (π), what is called the target policy, and one or more policies (β, …) called behavior policies, that are used to generate the experience. Q-learning is the off-policy version of TD.

10.4 Control using SARSA

The control using SARSA is done by using the TD(0) policy evaluation, that estimates action-value functions, and then using the ε-greedy policy, as seen in the figure 10.e. It is not compulsory to store a policy. The action to take can be chosen ε-greedily every time it is needed, based on the action-values of the current state and each one of its actions.

```
input S, A, γ, α
array q(s,a) ← 0, for each s,a in S,A

for each episode:
    s_t ← select starting state
    a_t ← select action ε-greedily
    for each step in episode (starting from t=1):
        r_t+1,s_t+1 ← get_next_time_step(a_t)
        a_t+1 ← select action ε-greedily
        q(s_t,a_t) ← q(s_t,a_t) + α[r_t+1 + γ·q(s_t+1,a_t+1) − q(s_t,a_t)]
        s_t ← s_t+1
        a_t ← a_t+1
        if s_t is terminal then break
```

Figure 10.e. Pseudocode for SARSA control using action-value functions.

10.5 Control using Q-learning

Control using Q-learning is similar to SARSA, but in this case, the action with the biggest action-value is used to update the action-value function. This means that the update is independent of the transition made, as the next state-action pair is not used to calculate it.

```
input S, A, γ, α
array q(s,a) ← 0, for each s,a in S,A

for each episode:
    sₜ ← select starting state
    for each step in episode (starting from t=1):
        aₜ ← select action ε-greedily
        rₜ₊₁,sₜ₊₁ ← get_next_time_step(aₜ)
        q(sₜ,aₜ) ← q(sₜ,aₜ) + α[rₜ₊₁ + γ·maxₐq(sₜ₊₁,a) − q(sₜ,aₜ)]
        sₜ ← sₜ₊₁
        if sₜ is terminal then break
```

Figure 10.f. Pseudocode for Q-learning control using action-value functions.

10.6 Summary and connection with the practical part of the project

In this chapter we saw the first reinforcement learning methods for solving decision-making problems. First of all, we analyzed how important it is to find the right balance between exploiting the options that promise to bring the most returns and exploring others that could potentially bring even more. Knowing that, we saw that we could get the expected returns of the states or state-action pairs by sampling returns from the environment following a certain policy and updating the value or action-value functions. Once these value functions were precise enough, we could update the policy in order to give more importance to the states or action-state pairs that promised bigger returns, always giving some room for exploration.

While it seems a promising way to solve the Rocket League bots problem, the methods shown in this chapter only work with finite state and action spaces, and if they are not too big. In the Rocket League case, both the state and action spaces are continuous, so it is impractical to solve them this way. However, these are the foundations for the methods that will be able to solve the problem.

Chapter 11: Reinforcement learning: approximate solution methods

In this chapter we are going to analize reinforcement learning solutions to problems that cannot be solved by tabular solution methods. These problems have state spaces too big to be stored in memory, and their policies would take too much time to optimize. These state spaces can be things like coordinates in a map (which can be infinite if we consider continuous coordinates), velocities, etc. One of the biggest problems with these large state spaces is that most of the states would not be visited while gathering experience through the methods studied in the last chapter, so it would be impossible to obtain correct results. So the objective of the approximate solution methods, studied in this chapter, is to generalize the learned experience, so when learning new data for a visited state, it also applies to states similar to it. For example, if the state space of a certain problem is the coordinates of a robot, the value of two states representing two coordinates separated by 1mm is expected to be similar. There are three main families of algorithms: **function approximation methods**, **policy-gradient methods** and **actor-critic methods**.

11.1 Function Approximation methods

In function approximation methods, the value function is not represented as an array or table, but as a parameterized function. This function has a weight vector **w** $\in$ $\mathbf{R}^{\mathbf{d}}$. $\hat{v}_w(s) \approx v_\pi(s)$ is the approximate value of the state **s** given the weight vector **w**. The action-value function is represented with $\hat{q}_w(s, a) \approx q_\pi(s, a)$. The dimensionality of **w** (the number of weights in the vector) is smaller than the number of states, so changing one of the weights affects the value of several states. This also allows partially observable states by the agent, which means that the agent might only see some of the variables that define the state. In this case, instead of getting the current state, the agent gets an **observation** of the state.

The function approximator: a black box

As seen in the figure 11.a, for now, we can imagine the function approximator as a black box where we can input a state-action pair and it outputs the approximate action-value function in that state and action. We will consider this approximator as a black box until the end of the section (this black box is $\hat{q}_w(s, a)$), as it is better to explain first the things that are common for all of them.

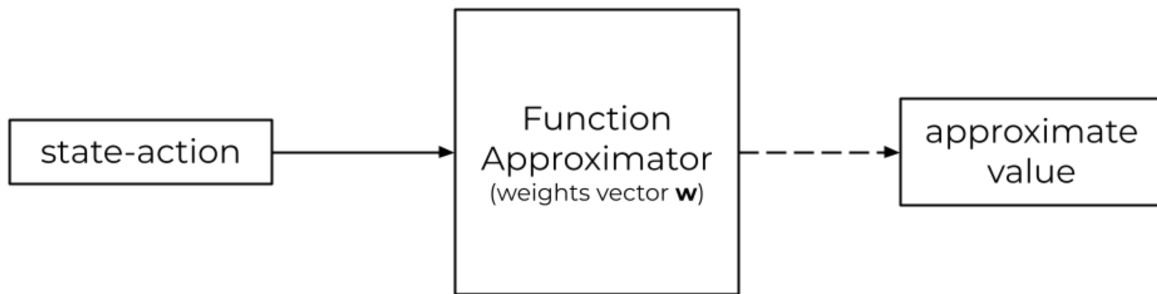


Figure 11.a. Function approximation diagram.

Vector of weights and vector of features

The weights of the function approximator can be adjusted. The objective is to adjust the weights vector **w** in a way that, for each state **s** and action **a**, $\hat{q}_w(s, a)$ is as close as possible to $q_\pi(s, a)$ (the real action-value function in the pair **a**-**s**).

Before explaining how to approximate the value function, we are going to see how states are represented. States are represented with what is known as a feature vector. Feature vectors can be formed by real numbers that can represent lots of different things: coordinates, distances, velocities, rgb colors, whether a light is on or not, etc. We can see the representation of the feature vector below.

state feature vector

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

state-action feature vector

$$x(s,a) = \begin{pmatrix} x_1(s,a) \\ \vdots \\ x_n(s,a) \end{pmatrix}$$

To connect with what was said in the last chapters, we have to realize that the states of tabular problems can be also represented as a feature vector. For a certain state **s**, all features will be 0 except for the feature corresponding to that state **s**.

$$x(s) = \begin{pmatrix} \text{if } s = s_1 \text{ then } \mathbf{1} \text{ else } \mathbf{0} \\ \vdots \\ \text{if } s = s_n \text{ then } \mathbf{1} \text{ else } \mathbf{0} \end{pmatrix}$$

The weights are:

$$w = (w_1, \ldots, w_n)$$

The objective function

An objective function is the function that we want to minimize or maximize. In this case, the objective of function approximation methods is to approximate $\hat{q}_w(s, a)$ as much as possible to $q_\pi(s, a)$. To achieve that, we have to minimize the mean squared error. For this reason, the objective function is the minimization of the mean squared error of $q_\pi(s, a)$ and $\hat{q}_w(s, a)$.

The error is the difference between the true value $q_\pi(s, a)$ and the approximated error $\hat{q}_w(s, a)$. The square is added to penalize more the big errors (and implicitly converts all errors to a positive number). So, the mean squared error is as follows:

$$J(w) = E_\pi\left[(q_\pi(s, a) - \hat{q}_w(s, a))^2\right]$$

Stochastic gradient descent

When this mean squared error is minimized, it means that the approximated action-value function is equal to the real one. This hardly happens, but we can try to make it pretty close. The technique used to minimize this mean squared error is called stochastic gradient descent. The function $J(w)$ has as many parameters as the dimensionality of the vector **w**. For this reason, the gradient of $J(w)$ is:

$$\nabla_w J(w) = \begin{pmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_n} \end{pmatrix}$$

Doing stochastic gradient descent means moving the parameters **w** in the gradient direction that minimizes the $J(w)$ function. The amount of variation we have to apply to the parameters **w** comes from:

$$\Delta w =_{(1)} -\frac{1}{2}\alpha \nabla_w J(w) =_{(2)} -\frac{1}{2}\alpha \nabla_w E_\pi\left[\left(q_\pi(s,a) - \hat{q}_w(s,a)\right)^2\right]$$

$$=_{(3)} -\frac{1}{2}\alpha E_\pi\left[2\left(q_\pi(s,a) - \hat{q}_w(s,a)\right) \cdot \nabla_w - \hat{q}_w(s,a)\right]$$

$$=_{(4)} \alpha E_\pi\left[\left(q_\pi(s,a) - \hat{q}_w(s,a)\right) \cdot \nabla_w \hat{q}_w(s,a)\right]$$

In the equality (1), on the right side we have a negative symbol, which means the gradient is followed in the descending direction. The **1/2** is just to make the final equation cleaner, as when deriving, a **2** is generated (and it is canceled with the **1/2**). Finally, the α is the step size (the rate of learning). On the right side of the second equality, we just substitute the J(w) equation. In the equality (3) we use the derivative chain rule, and in the fourth one we move the negative and the **1/2** inside.

General update rule

Instead of calculating the expectation Δw will be calculated by taking samples. For this reason, the update rule for Δw is:

$$\Delta w = \alpha(q_{\pi}(s, a) - \hat{q}_w(s, a))\nabla_w \hat{q}_w(s, a)$$

Until now, when calculating the error (the difference between the real action-value and the approximated one), we used the real action-value. In reality, we cannot know it (that is the reason we are using an approximator). For this reason, in order to obtain samples and calculate the changes in the weights (Δw) we need to use different methods. In this text, we are going to see the same ones we saw in the tabular solutions chapter.

Applying this update rule in each state sampled will adjust the weights in a way that the approximated action-value gets closer to the real value each iteration. This way, this can be used as policy evaluation.

Notice that for the next subsections, the red part of the equations correspond to the target for the corresponding method used to obtain the samples, and the blue and purple parts depend on the approximator used. For some methods, the red part may also depend on the approximator used.

Monte Carlo update rule

Monte Carlo update rule uses the returns obtained by sampling an episode, as seen in chapter 10:

$$\Delta w = \alpha(G_t - \hat{q}_w(s, a))\nabla_w \hat{q}_w(s, a)$$

Temporal-Difference learning update rule

Temporal-Difference learning update rule uses the returns obtained by sampling an episode, as seen in chapter 10:

$$\Delta w = \alpha(r_{t+1} + \gamma \hat{q}_w(s_{t+1}, a_{t+1}) - \hat{q}_w(s_t, a_t))\nabla_w \hat{q}_w(s_t, a_t)$$

Function approximation control

To do function approximation control, we simply have to follow the same pattern we saw in the Monte Carlo, SARSA and Q-learning algorithms, where we followed an ε-greedy policy.

The function approximator: inside the black box

Earlier in this section we said that we would treat the function approximator. If we open the black box, we can see that there are different types of function approximators [19]. The most common ones are the following:

- Linear combinations of features
- Neural networks
- Decision trees
- Nearest neighbor
- Fourier / wavelet bases

Different ones can be used, and neural networks are one of the most commonly utilized, but in this case we will only explain the simplest one: the linear combination of features.

Linear combination of features

Linear combination of features is the simplest way to construct a function approximator. In this case, the vector of weights has the same dimensionality as the vector of features. We define $\hat{q}(s, a, w)$ as:

$$\hat{q}_w(s, a) = x(s, a)^T w = \sum_{j=1}^{n} x_j(s, a) w_j$$

We can also calculate its gradient:

$$\nabla_w \hat{q}_w(s, a) = \nabla_w x(s, a)^T w = x(s, a)$$

Now we can fully define the Monte Carlo and Temporal-Difference learning update rules:

Monte Carlo update rule using linear combination of features

$$\Delta w = \alpha(G_t - \sum_{j=1}^{n} x_j(s_t, a_t) w_j) x(s_t, a_t)$$

Temporal-Difference learning update rule using linear combination of features

$$\Delta w = \alpha((r_{t+1} + \gamma \sum_{j=1}^{n} x_j(s_{t+1}, a_{t+1}) w_j) - \sum_{j=1}^{n} x_j(s_t, a_t) w_j) x(s_t, a_t)$$

11.2 Policy-gradient methods

Policy-gradient methods are a different approach, where we directly parameterize the policy, instead of the value function or the action-value function. This means that these parameters directly affect the probability of choosing an action given a state and the parameters. We can define:

$$\widehat{\pi}_\theta(s, a) = P(a|s, \theta)$$

These methods have a few advantages over the function approximation methods. They are more practical when working with continuous action spaces, as with policy gradient methods you do not have to find the one with the most action-value to update the policy greedily. Another advantage is that with policy-gradient methods, we can build stochastic policies, where not only one action is the optimal one, but all of them can have a certain probability to be chosen. This can be useful to solve certain problems.

The objective function

In policy-gradient methods, there are different objective functions, depending on the nature of the problem.

For episodic problems, we can consider the objective function:

$$J(\theta) = v_{\widehat{\pi}_\theta}(s_0) = E(r_\tau)$$

Where s_θ is the starting state in the episode and v_{π_θ} is the true value function for the policy π determined by the parameters θ.

For continuing problems, we have two different objective functions:

$$J(\theta) = \sum_s d_{\widehat{\pi}_\theta}(s) v_{\widehat{\pi}_\theta}(s)$$

$$J(\theta) = \sum_s d_{\widehat{\pi}_\theta}(s) \sum_a \widehat{\pi}_\theta(s, a) R(s, a)$$

Stochastic gradient ascent

Computing the policy gradient and using it to do policy gradient ascent is useful to adjust the parameters θ in the direction that maximizes the objective function. The following is the gradient of the objective function:

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

The objective is to find the change in the parameters θ in the direction of the gradient of the objective function, multiplying it by a learning rate α.

$$\Delta\theta = \alpha \nabla_\theta J(\theta)$$

First of all, we are going to explore what is known as the **score function**.

If we have a differentiable policy $\nabla_\theta \widehat{\pi}_\theta(s, a)$, we have that:

$$\nabla_\theta \widehat{\pi}_\theta(s, a) = \widehat{\pi}_\theta(s, a) \frac{\nabla_\theta \widehat{\pi}_\theta(s,a)}{\widehat{\pi}_\theta(s,a)} = \widehat{\pi}_\theta(s, a) \nabla_\theta log \widehat{\pi}_\theta(s, a)$$

The gradient of the logarithm of the policy is what is known as the score function.

$$\nabla_\theta log \widehat{\pi}_\theta(s, a)$$

This gradient tells in which direction the parameters θ have to be adjusted in order to get more probability for a certain action **a**. This score function is useful when using policy-gradient methods, as we will see.

Types of policies

We have different types of policies (for now, we have seen the greedy and ε-greedy policies). Depending on the policy used, the score function will be different.

First we are going to see the **softmax policy**. In softmax policy, we use a linear combination of features, as seen in the previous chapter. From these combinations, we can build the policy:

$$\widehat{\pi}_\theta(s, a) = \propto e^{x(s,a)^T \theta}$$

which means that the policy is proportional to (∝) the exponentiation of the combination of features. The state-action pairs with a bigger exponentiated combination of features will have a bigger probability. The score function for the softmax policy is:

$$\nabla_\theta log \widehat{\pi}_\theta(s, a) = x(s, a) - E_{\pi_\theta}[x(s, \cdot)]$$

So, the score function for the softmax policy is the difference between the combination of features of the action we took and the average combination of features value. It tells how good it was to take a certain action compared to the other actions of the state.

We also have the **Gaussian policy**. The Gaussian policy is more useful for continuous action spaces as the steering degree of a car. We calculate the mean using a combination of features $y(s) = x(s)^T \theta$, and we also have a variance σ^2 that may be fixed or also parametrized. The policy is a Gaussian:

$$a \sim N(y(s), \sigma^2)$$

The mean action is the one with the most possibilities to be chosen, and the further away an action is from the mean, the less probability it has to be chosen. The score function for the Gaussian policy is:

$$\nabla_\theta log \widehat{\pi}_\theta(s, a) = \frac{(a - y(s))x(s)}{\sigma^2}$$

So again, the score function is the difference between the combination of features of the action we took and the mean action, and scaled by the variance.

The Policy Gradient Theorem

To calculate the gradient of the objective function, and with that, the adjustment of the parameters θ in order to maximize that objective function, we can use the Policy Gradient Theorem:

$$\nabla_{\theta} J(\theta) = E_{\pi_\theta}\left[\nabla_{\theta} log \hat{\pi}_{\theta}(s, a) q_{\pi_\theta}(s, a)\right]$$

The Policy Gradient Theorem says that the gradient of the objective function is equal to the expectation of the gradient of the score function of the actions taken multiplied by the action-value function of these state-action pairs. This means that, if an action has a positive score function (meaning that its combination of features is bigger than the mean in that state) we adjust the parameters in the direction of that action (proportional to its action-value function), making it more probable.

Monte Carlo Policy-Gradient (REINFORCE)

This is the most basic policy-gradient algorithm. In it, we sample several episodes, and for each time step of them, we adjust the parameters θ using the Policy Gradient Theorem, but changing the theoretical action-value function for the sampled return. A learning rate is used to determine how much we move in the direction of the gradient.

```
input S, A, π, γ, θ

for each episode {s1,a1,r2,s2,a2,r3,s3,a3,...,sT-1,aT-1,rT} ~ π̂θ do:
    returns ← 0
    for each t in episode (starting in t-1):
        returns ← rt+1 + γ·returns
        θ ← θ + α∇θ log π̂θ(st, at)·returns
return θ
```

Figure 11.b. Pseudocode for Monte Carlo Policy-Gradient (REINFORCE) algorithm.

11.3 Actor-critic methods

Actor-critic methods combine the two main families of reinforcement learning methods: function approximation and policy-gradient methods. In these methods, the **critic** is the part that approximates the action-value function. The **actor** part is the one that approximates the policy. They both work together in the sense that the **critic** approximates the action-value function that follows the policy approximated by the **actor**, and the actor approximates the policy using the approximated action-value function by the **critic**.

The critic is then the approximation of the action-value function following the actor's approximation:

$$\hat{q}_w(s, a) \approx q_{\pi_\theta}(s, a)$$

And the actor is the parametrized policy:

$$\hat{\pi}_\theta(s, a) = P(a|s, \theta)$$

In actor-critic methods we keep two different sets of parameters, **θ** and **w**. The update rule for the parameters **w** is equal to the ones seen in the function approximation section (it depends on the method and approximator used), while the update rule for the parameters **θ** use the Policy-Gradient Theorem, but with the approximated action-value from the critic, instead of using returns.

The pseudocode in the figure 11.c is a basic actor-critic method that uses a Temporal-Difference learning update rule and linear combination of features as a critic.

```
input S, A, R, π, γ, w, θ

select initial s_t from S
a_t ← π̂_θ(s_t)

for each t do:
    r_t+1 ← R(s_t,a_t)
    s_t+1 ← sample next state
    a_t+1 ← π̂_θ(s_t+1)
    θ ← θ + α∇_θ log π̂_θ(s_t, a_t)·q̂_w(s_t, a_t)
    w ← w + α(r_t+1 + γq̂_w(s_t+1, a_t+1) − q̂_w(s_t, a_t))∇_w q̂_w(s_t, a_t)
    s_t ← s_t+1
    a_t ← a_t+1
return θ
```

Figure 11.c. Pseudocode for the Actor-Critic algorithm.

Actor-critic with a baseline: advantage function

When the action-values are estimated, their values depend on the magnitudes used when designing the rewards. If the rewards are of the order of magnitude of 10, the action-values will be so different that if they are of the order of magnitude of 1,000,000. This means that, when doing the policy parameters update, the amount these parameters change depends on it. For this reason, it is interesting to introduce a baseline. With this, the amount of change of the parameters is not directly proportional to the action-value, but is also affected by the baseline. With it, we can also have baselines that make the parameters move towards the action if it is estimated to be good, and move away from it if it is estimated to be bad. If we define the baseline as $b(s)$, the update rule for the **θ** parameter would be:

$$\alpha \nabla_{\theta} log \widehat{\pi}_{\theta}(s, a)(q_{w}(s, a) - b(s))$$

One of the most widely used baselines is the value function. The subtraction of the value function to the action-value function is known as the **advantage function**:

$$a = \widehat{q}_{w}(s, a) - \widehat{v}_{v}(s)$$

The idea behind the advantage function is that it represents how good it is to take an action with respect to how good it is to be in the state where that action is taken from. So, if the advantage function for a certain action is positive, it means that taking that action is better than expected from the state-value point of view. For this reason, when we have a positive advantage function for an action, we adjust the parameters in the direction of that action, proportional to how better it is to take that action. If the advantage function is negative, we move the parameters in the opposite direction, giving the action less probability.

One of the ways to achieve that would be to have another approximator for the value function, apart from the one for the action-value function. That would mean more parameters to update, and more complexity. We can achieve the same by considering only the value function approximation. This way, for the pseudocode example we have seen, we could substitute the action-value for the TD target:

$$\hat{q}_w(s_t, a_t) \approx r_{t+1} + \hat{v}_v(s_{t+1})$$

This is true because $\hat{q}_w(s_t, a_t)$ is equal to $r_{t+1} + \hat{v}_w(s_{t+1})$ averaged for each one of the possible transition states of **s**. This means that, taking enough samples, all the different transitions from **s** to **s'** taking action **a** will be made, and the action-value would approximate to the real one. For this reason, it is possible to just approximate the value function instead of the action-value function, and then the advantage function would be:

$$A(s, a) = r_{t+1} + \hat{v}_v(s_{t+1}) - \hat{v}_v(s)$$

```
input S, A, R, π, γ, w, θ

select initial sₜ from S
aₜ ← π̂_θ(sₜ)

for each t do:
    rₜ₊₁ ← R(sₜ,aₜ)
    sₜ₊₁ ← sample next state
    aₜ₊₁ ← π̂_θ(sₜ₊₁)
    θ ← θ + α∇_θ log π̂_θ(s_t, a_t)·(r_{t+1} + v̂_w(s_{t+1}) − v̂_w(s))
    w ← w + α(r_{t+1} + γv̂_w(s_{t+1}) − v̂_w(s_t))∇_w v̂_w(s_t)
    sₜ ← sₜ₊₁
    aₜ ← aₜ₊₁
return θ
```

Figure 11.d. Pseudocode for the Actor-Critic algorithm with baseline.

The reinforcement learning method used in this project: PPO

PPO, which stands for Proximal Policy Optimization, is the reinforcement learning method chosen to develop the practical part of the project. To start to understand it, we will first take a look at the concept of probability ratio:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$$

This is the ratio in the probability of taking an action **a** when in the state **s**, for the old policy, and the updated one.

We can consider a basic objective function as maximizing:

$$J(\theta) = E\left[r(\theta) \cdot A_{\theta_{old}}(s, a)\right]$$

This maximization of the objective function is achieved when, for actions with a positive advantage (they are better than the average for their state), we have a ratio as big as possible, which means that the probability of the action has increased. When the action has a negative advantage (it is worse than the average for its state), the objective function maximization is achieved when the ratio is as small as possible (so we get a negative number as close to 0 as possible).

Doing this might sometimes cause problems, as when trying to maximize, the updates might be too big, and an action that only needed a small probability increment ends up with all of it. It is for this reason that, for the PPO objective function we use what is known as a clip function. A clip function limits how small or how big a number can get:

$$clip(x, min, max) = \begin{pmatrix} \boldsymbol{min} & if\, x < min \\ \boldsymbol{x} & if\, min < x < max \\ \boldsymbol{max} & if\, max < x \end{pmatrix}$$

We can use this clip function in the objective function in order to limit how much we update the policy. Specifically, the objective function is the maximization of the minimum between the probability ratio multiplied by advantage, and the clipped probability ratio multiplied by the advantage:

$$J(\theta) = E\left[min(r(\theta) \cdot A_{\theta_{old}}(s, a),\ clip(r(\theta),\ 1 - \varepsilon,\ 1 + \varepsilon)A_{\theta_{old}}(s, a))\right]$$

where ε is a parameter that determines the clip range.

Given this objective function, the update rule is:

$$\theta_{k+1} = argmax_{\theta}(J(\theta_k))$$

which means that we set as new parameters the ones that maximize the objective function (that uses the sampled state and action to calculate the objective function and the probability ratio).

11.4 Summary and connection with the practical part of the project

In this chapter we have seen how to solve real problems with approximate solutions. We analyzed two different approaches, and an additional one that combines both of them. Function approximation methods work better with discrete action spaces, which is not the case of the Rocket League bots, as we will see in the next chapters. For this reason, it was discarded in favor of policy gradient and actor-critic methods. After trying several methods, PPO was the one that gave the best results, and that is the reason it was chosen to develop the project.

Part IV: Reinforcement Learning Hands-On

This fourth part will go through the main things to consider when designing a reinforcement learning solution to a decision-making problem. First, a general description of how to face the problem will be given, to then give an explanation of how the practical part of this project, creating a Rocket League bot, was carried out.

Chapter 12: Designing a reinforcement learning solution to a decision-making problem

In the previous part of the text, we have seen a theoretical explanation of reinforcement learning. In this part, we will give some hints on how to create a reinforcement learning solution for a decision-making problem. In this part of the text we assume that we will use the existing reinforcement learning methods (some of them described in the previous part) rather than designing ourselves. Even if we are not going to talk about methods design, it is important to know the basis of reinforcement learning, and for this reason it is recommended to read the previous part of the text. This chapter will talk about decision-making problems in general while in the next one we will dive into how the practical part of this project, the creation of a Rocket League bot, was made. In the next sections we will see different aspects to consider when designing a reinforcement learning solution for decision-making problems.

12.1 Extracting information from the environment

First of all, it is important to know the environment. We can find ourselves in a problem where the environment is part of the real world, and sometimes it is digital instead. Even if it is part of the real world (like for example training a robot to walk), it is interesting to create a simulation for it to learn. This has several advantages. In the example of the walking robot, training it in real life means that every time it fails and falls, it could break. You also need to put it in the starting position every time you end an episode. But one of the most important advantages of using a simulation is the possibility of training much faster, as, if well done, the simulation should be able to run at a speed faster than physics time.

No matter what the environment is, it is important to know how to extract information from it. In digital environments, it should be possible to get any information from the environment. In the real world, it can be more complicated. We can use cameras, sensors, or any instrument capable of getting data. In case of using a simulation, it is important to only get data from it that can be also obtained in the real environment, as it will be needed when using the learned policies.

When working with an environment to solve a decision-making problem, it is important to be able to interact properly with it, in the sense that it has to provide a mechanism for working with steps, which reinforcement learning methods need. This environment needs to be able to stop at a certain moment, provide the information required, and then accept an action that potentially alters the state of the environment.

12.2 Defining the MDP

In reinforcement learning, decision-making problems are framed into an MDP. As seen in the theoretical part of the text, MDP's consist in a state space, an action space, a probability function, and a reward function. The probability function is what defines the transitions between states, affected by the dynamics of the environment and the actions taken. In real life, these dynamics are the physics of the world we live in. In a digital world, it depends on how it is implemented. This means that this probability function is not something we have to define when creating a solution for a decision-making problem. We can consider the state space as all the possible combinations of all the possible information that could theoretically be extracted from the environment. Only a subset of this is used in practice (the features), and it is known as the observation space (the information observable for the agent). The action space has to be defined, as it is the reason to create a solution (to decide which action to take in each situation). Finally we have the reward function, that determines the reward yield when executing an action **a** in the state **s** and transitioning to the state **s'**. For this reason, in order to solve a decision-making problem, first we have to define its observation space (features), action space and reward function.

12.3 Defining the observation space (features)

Once we have an environment we can interact with, it is important to realize that it is impractical to try to get all the information it can offer. First of all, environments of real problems might have an infinite amount of information that characterizes its state. For example, if we are trying to train a robot to walk, we could consider infinite information, such as each foot's distance to the ground, robot's torso angle towards the ground, distance between feet, speed, the distance to each one of the obstacles in a room, … and the list could go on. The second reason to not use all the information available in the environment is that, the more information used, the more computation will be needed to train the model.

For this reason, it is crucial to only use the needed information to represent the states. The set of information from the environment used while training the agent is known as features. It is also important that these features are able to characterize the state in a reasonably good way. Of course, there is a tradeoff between the capability of them to represent the state precisely and the time needed to train the model. If there are too many features, the model will take too long to be trained, and if the features are not representative enough of the state of the environment, the model could learn erroneous behaviors.

Now we are going to analyze this last problem. When using too few features, two apparently similar or equal states (judging by the features) could be really different in reality. In the figure 12.a we can see a decision-making problem example where an agent (it could be a robot) has to get from the starting points (red circles) to the goal position (green rectangle). Let's imagine that the agent receives a negative reward proportional to the distance between it and the goal, in each state. This will encourage the agent to move to states that are closer in order to reduce the negative reward, and to do it as fast as possible. The agent receives a positive reward when reaching the goal.

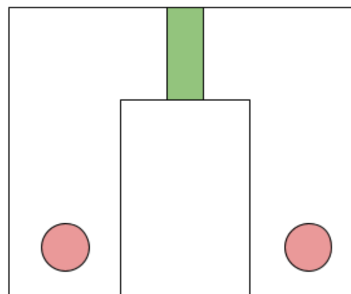


Figure 12.a. Example of the importance of selecting good features.

To start, we could consider only one feature: the distance from its position to the goal. The problem with having only this feature is that the agent could not differentiate between the two sides of the course. This means that the agent would surely learn that it has to go up, but at the moment of having to move horizontally, there would be a conflict. If we used a deterministic policy, the agent would learn to go either to the right or to the left when in the top corners (as based on the only feature, they look as if they were the same state), which would cause it to get stuck in one of the two sides. If using a stochastic policy, the agent could learn to go either left or right with the same probability, which would make it enter in an infinite loop. There are other problems with only having this feature, but they will be discussed in the next set of features presented for this problem, as this same problem arises with them.

The second set of features considered consists in two signed values: the horizontal and the vertical distances. A negative horizontal distance means that the goal is to the left, and a positive to the right. The same happens with the vertical distance, with the goal being down and up respectively. This can solve the problem, as now the top left corner has close to 0 vertical distance, and a positive horizontal distance, while the horizontal distance in the top right corner is negative. Now, these two positions in the course have different values in the features, which means that the agent can interpret them as different states.

12.4 Defining the action space

Defining the action space is equally important. These actions are the ones the agent can take in each time step. Actions can basically be discrete or continuous. Discrete actions can be things like choosing whether to go forward, backward, left or right. Continuous actions can be things like the amount of throttle that we apply to a car. It is also worth mentioning that some reinforcement learning methods allow you to take several actions at the same time step, for example going forward and left (creating a diagonal movement) or accelerating the car and steering at the same time.

12.5 Defining the reward function

Last but not least, we have to define the reward function. When programming, it can be defined as a regular function that receives the state, the action, and the state the environment has transitioned to due to that action. The output of the function is the reward yield. There are lots of different ways of designing a reward function. Some reward functions are sparse (in the sense that they only yield reward when reaching the goal, or only in some specific situations), and other ones are dense, yielding reward more often.

Sparse reward functions are good because they are easy to design and they always converge to a good policy if the goal is easy to reach. The bad thing about them is that they need more time to do so, and sometimes the goal cannot be achieved. When reaching the goal, all the state-action pairs that have led to achieving the goal receive part of the reward as a return. In case of having a discount factor of 1 (to get more information about it, read the chapter 8, section 8.4), all the state-pairs involved in that episode will receive that reward. In case of having a discount factor bigger than 0 and smaller than 1, the state-action pairs visited earlier will get less part of that reward that the ones visited right before reaching the goal. If the goal is hard to get with the starting (probably random) policy, the reward would never be yielded and the policy will never be updated.

In these cases where the goal is hard to get, it is a good idea to design a denser reward function that helps guide the agent towards the goal. These intermediate rewards can, for example, depend on the current state (the one before or after transitioning due to the action taken), depend on the state-action pair, or the difference between two or more features of the previous and the current state. These are only a few ideas on how to design a reward function. The best way to know which one is the most appropriate for a given problem, is to try all of them.

Chapter 13: The Rocket League bot problem

Before starting to solve the Rocket League bot problem, we have to analyze it. First of all, we will describe the objectives of the bot, in order to know what we want to achieve when creating the solution. Then, we will analyze the state space, in the sense of which information can be obtained from the Rocket League game match. Then we will see the actions available to be taken by the bot, and finally, we will see the methodology that will be followed to create the bot.

13.1 Rocket League bot objectives

The objective for a Rocket League bot is to win a match. This is a simple objective to understand, but a complex objective to to achieve if not divided into less complex ones. Winning a match consists in scoring more goals than the opponent. For this reason, we can divide the main objective into two: scoring a goal in the opponent's net and saving a goal in the own net. To simplify things, we will call these objectives **scoring a goal** and **saving a goal** respectively. Each of these could be divided into more objectives. In the diagram in the figure 13.a we can see one of the possible objective trees for a Rocket League bot.

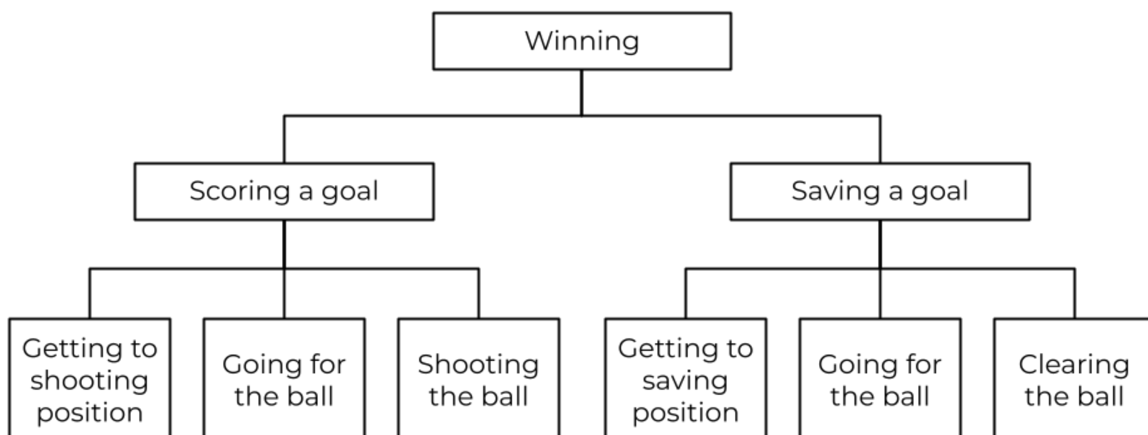


Figure 13.a. Possible Rocket League bot objective tree.

By no means it tries to be the only possible objective tree for a Rocket League, or even the correct one. This is only a way of showing how the objectives can be divided into smaller ones in order to have easier problems to solve.

13.2 Rocket League state space

Before implementing a solution to train a Rocket League bot, we have to know its state space. Basically, we want to know which information can be obtained from a Rocket League match in order to be able to design the features. Of course, in a game like Rocket League, we could extract endless information, but we will analyze the most important. In Rocket League we have static and dynamic information.

Static information are things like:

- Coordinates of the blue team's goal
- Coordinates of the orange team's goal
- Coordinates of the back and side walls
- Height of the ceiling
- Coordinates of the boost pads

Of course, there is much more static information to be extracted from a Rocket League match, but these are the main ones. Most of them refer to elements of the field, which does not change at all during a match.

Now, if we consider the dynamic information, we can find the following:

- Blue team's score
- Orange team's score
- Time left until each boost pad is available
- Ball position
- Ball linear velocity
- For each car:
    - Position
    - Rotation
    - Linear velocity
    - Angular velocity
    - Boost amount
    - Whether it has a jump available or not

More information could be obtained, but this is the most important. With this, interesting features could be created that would help the bot how to achieve the objectives better.

13.3 Rocket League action space

In Rocket League, both players and bots have several actions that they can perform. When training the bots' models, the objective is to find the best actions to take in each situation. The actions available are the following:

- Throttle: Continuous action. It indicates the amount of forward or backwards throttle applied to the car. Can be represented as a real number between -1 and 1, where [-1,0) interval means throttling backwards, while the interval (0,1] means throttling forward.
- Steer: Continuous action. It indicates the amount of steering left or right applied to the car. Can be represented as a real number between -1 and 1, where [-1,0) interval means steering left, while the interval (0,1] means steering right.
- Pitch: Continuous action. It indicates the amount of pitching up or down applied to the car. Can be represented as a real number between -1 and 1, where [-1,0) interval means pitching down, while the interval (0,1] means pitching up.
- Yaw: Continuous action. It indicates the amount of yawing left or right applied to the car. Can be represented as a real number between -1 and 1, where [-1,0) interval means yawing left, while the interval (0,1] means yawing right.
- Roll: Continuous action. It indicates the amount of rolling left or right applied to the car. Can be represented as a real number between -1 and 1, where [-1,0) interval means rolling left, while the interval (0,1] means rolling right.
- Jump: Binary action. It indicates whether to jump or not.
- Boost: Binary action. It indicates whether to boost or not.
- Handbrake: Binary action. It indicates whether to handbrake or not.

13.4 Methodology to create the bot

As mentioned earlier, the main objective of winning the game can be broken down into simpler objectives. This means that, instead of creating a complex model that captures all the bot's behavior, we could create several models that interact with each other. If we take a look in the figure 13.b, the objectives in the leaves can be solved with models that use all or part of the action space mentioned in the previous section. The objectives that are not leaves can be solved with models that take the models below as the action space.

This means that, if we have a model that tries to solve the problem of **winning**, depending on the state it will decide whether to try to score a goal or save it. For example, if the ball is near to the opponent's goal, it might decide that **scoring a goal** is the best action. At this point, the model that tries to solve the **scoring a goal** problem comes into play. This model will decide between the actions of **getting to shooting position**, **going for the ball** or **shooting the ball** depending on the situation. For example, if the car is not aligned with the ball and the opponent's goal, the model might decide that the best action to take is **getting to shooting position**. Once the **getting to shooting position** model kicks in. Its actions will be things like throttling, steering, etc.

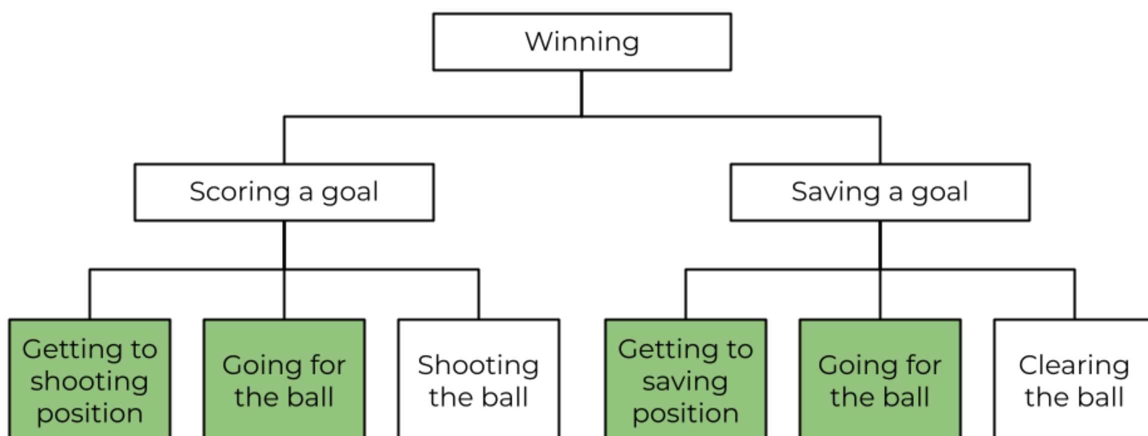


Figure 13.b. Objectives solved (green) with the models created in this project.

After examining the Rocket League problem, it was obvious that creating a winning bot in such a short amount of time would be too complex. For this reason, it was decided to focus on creating a model that was able to solve several objectives (the green objectives in the tree). This model tries to bring the car to any position on the field, whether that position is dynamic or not. This means that the position can be for example the goal center (static) or the ball (dynamic). In the next chapter we are going to see the system created to train and test this model.

Chapter 14: Creating a Rocket League bot using reinforcement learning

In this chapter we will see how the practical part of this project was implemented. First of all, we will discuss the main frameworks and libraries used in order to give a little bit of background and so the explanation is easier to understand. The remaining sections will analyze the system built to train the models. This project has been carried out using the programming language **Python**, and from this point onwards its terminology, and programming terminology in general, will be used.

14.1 Frameworks and libraries used

OpenAI Gym

OpenAI Gym is one of the most widely used frameworks for reinforcement learning solutions. We are not going to discuss in-depth how it works, but we are going to give a few ideas in order to understand the rest of the text and to encourage people its use.

In the next links we can visit its website and its source code:

https://gym.openai.com/

https://github.com/openai/gym

OpenAI Gym provides a python template to build environments that reinforcement learning methods will use to train the models. Its core class Env has the basic structure shown in the figure 13.a.

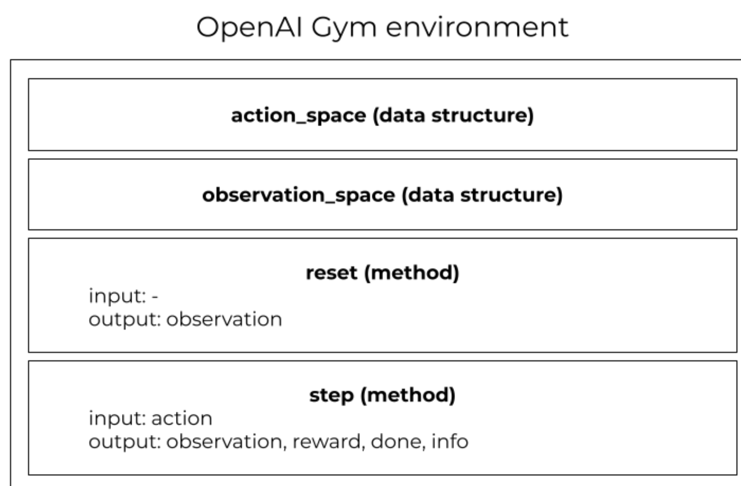


Figure 14.a. Basic structure of OpenAI Gym Env class.

The Env class is an abstract class, which means that it is like a blueprint that must be used to create the specific environment for the specific problem to solve. For this reason, when building the environment (section 13.3), the ***reset*** and ***step*** methods must be implemented.

In the ***reset*** method, the environment must be reset to start a new episode. In this method, it is important to get the data necessary to create an observation for the starting step. If we are training a video game bot for example, we should extract the necessary data of the current state of the game. If we are training a bot in real life, we should get the data from the process that is gathering the data from cameras, sensors, etc. When we have the observation, we return it as the output.

In the ***step*** method, the environment receives an action, and it must generate the next observation, depending on that action. In the previous example, if we are creating a game bot, an action could be going forward. This way, we would send that action to the game, and the game would return the next observation. We also have to calculate the reward yield. To achieve it, we can use a reward function where we input the action and the observation generated. It is possible to store the previous observation, so this way we have the observations before and after taking the action. We also have to calculate the **done** signal which indicates whether the environment is in a terminal state or not. Finally, the info field is to return additional information. It can be empty.

Both the ***action_space*** and ***observation_space*** attributes have to be initialized. It is usually done in the constructor of the class instance. OpenAI gym provides an easy way to declare these action and observation spaces. There are different formats of spaces:

- discrete (action and observation spaces): The discrete space has the specified number of possible values. In case of being used as an action space, only one of the actions will be chosen at a time.
- multi_discrete (action space): Multi discrete action spaces are a series of different discrete action spaces. When selecting the action, one action of each of the discrete spaces will be chosen.
- box (action and observation spaces): Bidimensional space of real numbers. The number of elements in each dimension can be chosen. If it is used as an action space, a certain value for each element is selected.
- multi_binary (action and observation spaces): Bidimensional space of binary elements. The number of elements in each dimension can be chosen. If it is used as an action space, a certain value for each element is selected.
- dict (observation space): A dictionary. It lets us use several different types of spaces, and encapsulate them into one.

Stable-Baselines3

Stable-Baselines3 is a set of libraries that implements several reinforcement learning methods.

In the next links we can visit its website and its source code:

https://stable-baselines3.readthedocs.io/en/master/

https://github.com/DLR-RM/stable-baselines3

In the project, several reinforcement learning methods were tried, all implemented in Stable-Baselines3, but the only one that gave good results and was mostly used was **PPO** (an actor-critic reinforcement learning method. To learn more about it, see chapter 11 section 11.3). Stable-Baselines3 reinforcement learning methods work with environments that follow the OpenAI Gym environment template.

RLGym

RLGym is a framework created by the RLBot community that uses the OpenAI Gym template to create a Rocket League environment.

In the next links we can visit its website and its source code:

https://rlgym.org/

https://github.com/lucas-emery/rocket-league-gym

It implements the **Env** class mentioned earlier, and has a mechanism to communicate with the game. This communication lets the game process send the framework process data about the game state, and the framework can send back information about the actions taken, and other data. At the start of the project, it was used as an environment, but it had several limitations, as both the action and the observation spaces were predefined. For this reason, it was decided to not use anything of the framework related to reinforcement learning, and only use it to start the game process and communicate with it. For this reason, we are not going to explain how the framework works, as it is not important for the study of reinforcement learning.

Other libraries

Other libraries that have been used are **math** and **numpy**. These two libraries provide mathematical functions, and the latter provides tools to work with big amounts of data in an easy way.

14.3 How the system works

As seen in the figure 14.b, we have two different processes. The first one is the learning process itself, and the second one is the Rocket League process. When we execute the learning process, the ***main*** function starts. This ***main*** function creates the Rocket League process (using the part of the RLGym framework dedicated to that). After that, the ***main*** function creates the environment and the model (or loads it in case of using an existing one). Finally, the ***main*** function starts the learning method, which updates the model and uses the environment.

As seen in the diagram, the environment is the only one that communicates with the Rocket League process. It sends actions to it, and the Rocket League process sends the game state back when asked. The Rocket League process is capable of pausing the game. This way, when the environment sends an action, it will update the game state (for example moving the car according to the actions, updating the game based on physics, etc) and then it will pause until more actions are sent. Apart from that, we can consider the Rocket League process as a black box.

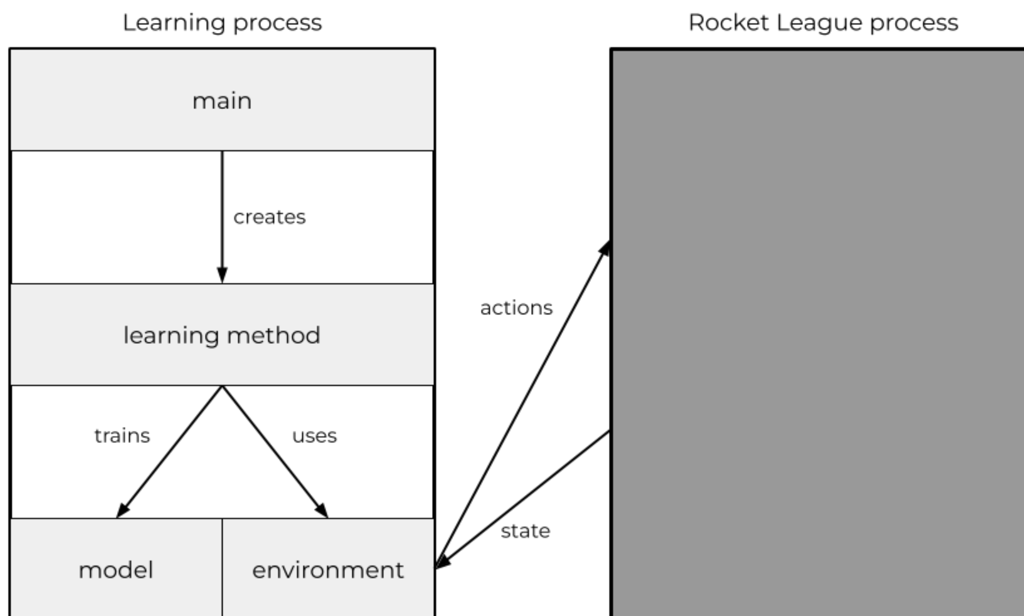


Figure 14.b. Structure of the training system.

In the figure 14.c we can see the code for the main function. We enter the function and create the Rocket League process, saving a reference to it.

In line **03** we enter a code section depending on the argument of the program. When we want to train a model, we have to use the argument '***training***'. When we are in the training part of the code, we first create the environment (that implements the OpenAI Gym Env class mentioned in section 14.1, and that will be discussed in section 14.5), inputting the reference to the Rocket League process to it, so they can communicate with each other. After it, we create the model (or load it, in case we want to continue training a model previously created), inputting the environment to it. The model is created (or loaded) using one of the Stable-Baselines3 libraries, in this case PPO (after the name of the PPO reinforcement learning method). We will analyze the model creation in the next section. Finally, we enter an infinite loop where the model is trained for several episodes and then saved.

If we have used the argument '***predicting***' instead, we enter in the piece of code used to test the trained models. In it, we create the predicting environment (similar to the training environment, but a bit simpler, because some things do not need to be done). Then, we load the model we want to test. In this case, we are going to run some episodes and count the number of seconds per episode (counting the steps per episode and dividing by the number of steps per second). In line **15** we open a file where we will write the number of seconds per episode. After that, we enter a loop of 1000 episodes. For each one of them, we reset the episode, obtaining an observation, we initialize to 0 the step counter, we mark the **done** signal as ***False*** and we start a loop, where each iteration is a step. This loop ends when a terminal state is reached. In each iteration, we select an action using the last observation, and we use that action to advance one step in the environment and get a new observation and a **done** signal. At the end of the iteration, we increase the step counter. At the end of each episode, we write its number of seconds in the file. When we have predicted all the episodes, we close the file and the game process and pipe (via the environment **close** method).

```
01- if __name__ == '__main__':
02- game_process = launch_rocket_league(
     CommunicationHandler.format_pipe_id(0),
     'epic')

03- if sys.argv[1] == 'training':
04-   environment = TrainingEnvironment(game_process=game_process)
05-   if exists('models/model.zip'):
06-     model = PPO.load('models/model.zip', env=environment)
07-   else:
08-     model = PPO(
          policy='MultiInputPolicy',
          batch_size=64,
          n_epochs=32,
          env=environment)
09-   while True:
10-     model.learn(total_timesteps=10 * 2048, n_eval_episodes=10)
11-     model.save('models/model.zip')

12- elif sys.argv[1] == 'predicting':
13-   environment = PredictingEnvironment(game_process=game_process)
14-   model = PPO.load('models/model.zip', env=environment)
15-   data_file = open('data/model_stats.txt', 'w')
16-   for _ in range(1000):
17-     observation = environment.reset()
18-     steps = 0
19-     done = False
20-     while not done:
21-       actions = model.predict(observation)[0]
22-       observation, done = environment.step(actions)
23-       steps = steps + 1
24-     data_file.write(str(steps/15) + '\n')
25-   data_file.close()
26-   environment.close()
```

Figure 14.c. Main function.

14.4 Basics of the model creation and training

Stable-Baselines3 provides several libraries to create and train reinforcement learning models. In this case, after testing several, it was decided to use the PPO library, which provides a way to create a model and train it using the PPO reinforcement learning method. This method is an actor-critic method (read chapter 11, section 11.3 to get more information about it). As seen in the previous section, the model is created using the PPO class, which accepts several parameters. We will not go through the parameters, as they are well explained in the documentation that can be found in:

https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

It is important to note though, that the environment created has to be passed as an argument. When training the model (with ***model.learn***, where ***model*** is the model created), we have to indicate the number of time steps and episodes to train the model for.

When the learning method is training the model, it will input an action to the environment, and the environment will output the current observation, the reward obtained and the **done** signal (which indicates whether the current state is a terminal state or not). With this information, the learning method can update the model properly.

14.5 Building the environment

In the previous section we saw that the learning method uses the environment to retrieve useful information to update the model. Stable-Baselines3 is compatible with environments that follow the OpenAI Gym template (environments that implement the abstract class Env from OpenAI Gym). For this reason, in this section we explain the implementation of the environment made for this project.

In the figure 14.d we can see the method ***__init__*** of the ***TrainingEnvironment*** class. This method is called when the environment is created. The lines **03** and **04** stores in class attributes the objects needed to communicate with the Rocket League process. In line **05** we can see we declare the attribute ***learning_object***. This is one of the most important aspects of the environment. The ***learning_object*** is an instance of the ***LEARNING_CLASS***, which is an implementation of a framework created for this project. This framework will be explained in the next section, but basically, it provides the action and observation spaces for the current learning strategy, a method to build the observation from the state, a reward function and a method to check whether the state is terminal or not. This way, we can change the ***LEARNING_CLASS*** used in the environment to rapidly change the rewards yields, the observations, the spaces, etc. In the next line, we use the static methods of the ***LEARNING_CLASS*** to get the observation and action spaces. Finally, we open a communication pipe with the Rocket League process to be able to communicate with it, and

we send it a message to configure the match. In this configuration we can see that different attributes can be specified. The first one specifies the amount of members for each team (in this case 1). The second attribute indicates if the bot to be trained will be playing against itself. The third one specifies whether the bot will face an opponent team or not. The fourth one indicates the number of game updates (or ticks) that the game will perform after receiving an action. This means that, if we specify a ***tick_skip=8***, when we send an action of going forward, the game will move the car forward during 8 ticks. At last, the ***game_speed*** attribute indicates the speed of the physics in the match. With ***game_speed=1***, one second in the game will be 1 second in real time. If the value of the ***game_speed*** attribute is bigger, the physics will be simulated faster, which will allow a faster training.

```
01- def __init__(self, game_process):
02-   super().__init__()

03-   self.game_process = game_process
04-   self.comms = CommunicationHandler()
05-   self.learning_object = None
06-   self.previous_observation = None
07-   self.observation_space, self.action_space =
        LEARNING_CLASS.get_spaces()
08-   self.comms.open_pipe(CommunicationHandler.format_pipe_id(0))
09-   self.comms.send_message(
        header=Message.RLGYM_CONFIG_MESSAGE_HEADER,
        body=list({
          'team_size': 1,
          'self_play': 0,
          'spawn_opponents': 0,
          'tick_skip': 1,
          'game_speed': 1}.values()))
```

Figure 14.d. __init__ method of the TrainingEnvironment class.

When starting an episode, the learning method will call the ***reset*** method of the ***TrainingEnvironment*** class. In it, we create an initial state to be sent to the Rocket League process. In line **02**, we create a default initial state, using the tool provided by the RLGym framework. In line **03** we update the initial state with custom values, for cars and ball positions, velocities, etc. It uses a tool created specifically for this project, and will be discussed in the next sections. In the next line we send the initial state to the Rocket League process so it can situate each object in the field. In line 05 we create an instance of the ***LEARNING_CLASS*** and save it in the ***learning_object*** attribute of the environment. This will be used, as mentioned earlier, to get the observations, the reward yield and the ***done*** signal. In the next line we get the current state of the game. This state is equivalent to the initial state we sent to the Rocket League process, but for format reasons, we cannot use the one we created earlier (as the initial state created and sent has a different format than the one received). In line 07 we create an observation from this state, according to the ***learning_object*** observation building method, which extracts features from the state, and we save it as the previous observation (so when we start with the first time step, that is the observation previous to taking the action). Finally, we return that previous observation to the learning method.

```
01- def reset(self):
02-   initial_state = StateWrapper(blue_count=1, orange_count=0)
03-   StateSetter.reset(
        initial_state,
        cars_position='random',
        cars_speed='random',
        cars_yaw='random',
        ball_position='random',
        ball_speed='random')

04-   self.comms.send_message(
        header=Message.RLGYM_RESET_GAME_STATE_MESSAGE_HEADER,
        body=initial_state.format_state())
05-   self.learning_object = LEARNING_CLASS()
06-   current_state = GameState(self.comms.receive_message(
        header=Message.RLGYM_STATE_MESSAGE_HEADER)[0].body)
07-   self.previous_observation =
        self.learning_object.get_observation(current_state)

08-   return self.previous_observation
```

Figure 14.e. reset method of the TrainingEnvironment class.

In each time step, the learning method will call the ***step*** method of the ***TrainingEnvironment*** class, inputting the action selected depending on the policy and value function of the model. First, we save the previous observation in an auxiliary variable, as its value will be changed. Then, we format the actions so they are easier to use when computing the reward. In line **04** we send those actions to the Rocket League process, so the game can update the state. Then, the current state is received, and the observation is created using that state. In line **07**, the current observation is saved in the ***previous_observation*** attribute so it can be used in the next step. Finally, we compute the reward yield and the done signal using the methods from the learning object. The current observation, the reward yield, the done signal, and additional information (that can be empty) are returned to the learning method.

```
01- def step(self, raw_actions: Any):
02-   previous_observation = self.previous_observation
03-   actions = self.learning_object.format_actions(raw_actions, 'step')
04-   self.comms.send_message(
        header=
          Message.RLGYM_AGENT_ACTION_IMMEDIATE_RESPONSE_MESSAGE_HEADER,
        body=self.learning_object.format_actions(actions, 'send'))
05-   current_state = GameState(self.comms.receive_message(
        header=Message.RLGYM_STATE_MESSAGE_HEADER)[0].body)
06-   current_observation =
        self.learning_object.get_observation(current_state)
07-   self.previous_observation = current_observation
08-   reward = self.learning_object.get_reward(
        previous_observation, actions, current_observation)
09-   done = self.learning_object.is_terminal_step(current_state)

10-   return current_observation, reward, done, {}
```

Figure 14.f. step method of the TrainingEnvironment class.

14.6 The learning framework

Now we are going to see the learning framework in detail. The following methods are part of the ***LEARNING_CLASS***, used in the environment, as seen before. The code used as an example is one of the possible implementations of this ***LEARNING_CLASS***, as it can be implemented as wished.

The first method is ***get_spaces***. It is a static function that returns two OpenAI Gym spaces, which are mentioned in section 14.1. In the next link you can visit the code of the OpenAI Gym spaces, which contain an explanation of how to use them:

https://github.com/openai/gym/tree/master/gym/spaces

In this method we define the observation and action spaces. In this example, the observation space is a Dict. This means that in each Dict element, another space can be fit. So, in this case, we create a python dictionary with two elements, each one of them containing a Box space. These Box spaces are unidimensional with one element, so they are basically an array with one element. All of this would be equivalent to creating a unidimensional Box with two elements. The only difference is that with the dictionary we can name the feature, so it is easier to understand the code this way. This is the methodology followed for observation spaces in the project development. When we have the python dictionary, we create a Dict space with it.

In box spaces, the ***low*** and ***high*** parameters indicate the minimum and maximum value of its elements, while the ***shape*** indicates the length of each dimension. Finally, ***dtype*** indicates the type of the elements in the box. They have to be a numpy type, where numpy is a mathematical library. In this case, the action space is a Box. As seen in the figure 13.g, the ***low*** and ***high*** parameters are numpy arrays instead of numbers, which means that each element of the box has a minimum and maximum. Here, the shape of the Box is equal to the shape of the ***low*** and ***high*** vectors (which should have the same shape).

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['angle_to_ball'] =
        spaces.Box(low=-180.0, high=180.0, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['distance_to_ball'] =
        spaces.Box(low=0.0, high=20000, shape=(1,), dtype=np.float32)

05-   return
        spaces.Dict(observation_space_dictionary),
        spaces.Box(
          low=np.array([-1.0, -1.0, -1.0, -1.0, -1.0]),
          high=np.array([1.0, 1.0, 1.0, 1.0, 1.0]),
          dtype=np.float32)
```

Figure 14.g. Example of the get_spaces method of the LEARNING_CLASS.

The second method is ***get_observation***. This method receives the state and returns the observation (features). The observations returned need to have the same format specified in the ***get_spaces*** function. For this reason, in the example below, we create a dictionary, and put the corresponding information in an array. In this case, there are two features. The first one is the angle between the velocity vector of the car and the vector from the car to the ball. The second feature is the distance between the car and the ball. Both features are calculated using the functions in the ***calcs*** file. This file has been created for this project and contains mathematical functions useful to work with vectors and coordinates. We will discuss it later. The features can be calculated using any information available in the state.

```
01- def get_observation(state: GameState):
02-   observation = dict()
03-   observation['angle_to_ball'] =
        [calcs.get_velocity_angle_to_point(
          state.players[0].car_data.position,
          state.players[0].car_data.linear_velocity,
          state.ball.position)]

04-   observation['distance_to_ball'] =
        [calcs.get_2d_distance(
          state.players[0].car_data.position,
          state.ball.position)]

05-   return observation
```

Figure 14.h. Example of the get_observation method of the LEARNING_CLASS.

The next method is ***get_reward***. In it, we receive the previous observation, the action taken and the current observation as parameters. From this information, we have to return a reward. We will not analyze this in depth as we did it in the previous chapter.

```
01- def get_reward(previous_obs, actions, current_obs):
02-     reward = 1000 if current_obs['distance_to_ball'][0] <= 200 else 0

03-     reward += - np.abs(current_obs['angle_to_ball'][0]) / 10
04-     reward += - current_obs['distance_to_ball'][0] / 1000

05-     return reward
```

Figure 14.i. Example of the get_reward method of the LEARNING_CLASS.

The next method is ***is_terminal_step***. We receive the current observation, and we have to determine whether the step is terminal or not. We return a bool. In this case, the step is terminal if the ball is close enough.

```
01- def is_terminal_step(current_obs):
02-     return True if current_obs['distance_to_ball'][0] <= 200 else False
```

Figure 14.j. Example of the is_terminal_step method of the LEARNING_CLASS.

The last function is ***format_actions***. We will not analyze it as it is not interesting for the matter of this text.

14.7 Other useful tools created: the state setter and the mathematical functions

The state setter

This is a simplified version of one of the state setters used to modify the starting state for an episode. In it, we modify the default values of the cars and ball, such as their position, rotation, linear and angular speed, boost amount, etc. In section 14.9 you can find more information about the code and visit the Github to find the full file with the complete code.

```
01- class StateSetter:
02-   def reset(state_wrapper: StateWrapper, cars_position='random',
        cars_yaw='random', cars_speed='zero', ball_position='random',
        ball_speed='zero'
      ):
03-     StateSetter._reset_cars(state_wrapper, cars_position, cars_yaw,
          cars_speed)
04-     StateSetter._reset_ball(state_wrapper, ball_position, ball_speed)

05-   def _reset_cars(state_wrapper,cars_position,cars_yaw,cars_speed):
06-     for car in state_wrapper.cars:
07-         # Boost
08-         car.boost = random.random()
09-         # Position
10-         car.position=[
              random.uniform(-3072.0, 3072.0),
              random.uniform(-4096.0, 4096.0), 17.0
            ]
11-         # Yaw
12-         car.rotation = [0.0, random.uniform(-np.pi, np.pi), 0.0]
13-         # Speed
14-         car_yaw_vector=calcs.get_vector_from_rotation(car.rotation[1])
15-         car_yaw_unit_vector =
              car_yaw_vector/ np.linalg.norm(car_yaw_vector)
16-         speed = random.uniform(0.0, 2300.0)
17-         car.linear_velocity = [
                  car_yaw_unit_vector[0] * speed,
                  car_yaw_unit_vector[1] * speed, 0.0]

18-   def _reset_ball(state_wrapper, ball_position, ball_speed):
19-     # Position
20-     state_wrapper.ball.position = [
          random.uniform(-3072.0, 3072.0),
          random.uniform(-4096.0, 4096.0), 93.0]
21-     # Speed
22-     state_wrapper.ball.linear_velocity =
          [random.uniform(0.0, 1000.0), random.uniform(0.0, 1000.0), 0.0]
```

Figure 14.k. Simplified version of the StateSetter class in default_state_setter.py.

Mathematical functions

This is a simplified version of the calcs.py file, which has mathematical functions to help obtain the features from the state. It provides algorithms to calculate distances, vectors, angles, etc. In section 14.9 you can find more information about the code and visit the Github to find the full file with all the functions.

```
01- def get_2d_distance(position1, position2) -> float:
02-   return np.sqrt(np.power(position2[0] - position1[0], 2) +
        np.power(position2[1] - position1[1], 2))

03- def get_3d_distance(position1, position2) -> float:
04-   return np.sqrt(np.power(position2[0] - position1[0], 2) +
        np.power(position2[1] - position1[1], 2) + np.power(position2[2]
        - position1[2], 2))

05- def get_2d_vector(position1, position2) -> np.ndarray:
06-   return np.array([position2[0] - position1[0], position2[1] -
        position1[1]])

07- def get_3d_vector(position1, position2) -> np.ndarray:
08-   return np.array([position2[0] - position1[0], position2[1] -
        position1[1], position2[2] - position1[2]])

09- def get_angle_between_vectors(vector1, vector2) -> float:
10-   return np.rad2deg(math.atan2(vector1[0] * vector2[1] - vector1[1] *
        vector2[0], vector1[0] * vector2[0] + vector1[1] * vector2[1]))

11- def get_vector_from_rotation(theta):
12-   return np.dot(np.array([[np.cos(theta), -np.sin(theta)],
        [np.sin(theta), np.cos(theta)]]), [1, 0])

13- def get_velocity_angle_to_point(o1_pos, o1_vel_vector, o2_pos):
14-   o1_to_o2_vector = get_2d_vector(o1_pos, o2_pos)
15-   velocity_angle_to_point =
        - get_angle_between_vectors(o1_vel_vector, o1_to_o2_vector)
16-   return 180 if o1_vel_vector[0] == 0.0 and o1_vel_vector[1] == 0.0
        else velocity_angle_to_point

17- def get_yaw_angle_to_point(state: GameState, point_pos):
      car_yaw_vector =
        get_vector_from_rotation(state.players[0].car_data.yaw())
      car_to_point_vector =
        get_2d_vector(state.players[0].car_data.position[0:2], point_pos)
      car_angle_to_point =
        get_angle_between_vectors(car_to_point_vector, car_yaw_vector)
      return car_angle_to_point
```

Figure 14.l. Simplified version of the calcs.py file.

14.8 Learning class implementations to create a model capable of bringing the car from a point A to a point B

In this section we are going to analyze different learning classes that were implemented in order to train models that are capable of bringing the Rocket League car from its location A to an arbitrary location B, which can be dynamic. As mentioned earlier, we built a system that is capable of training models interacting with an environment, which needs a learning class to get the observation and action spaces, as well as to get the features from the state, and calculate the reward yield. The most important aspect of these learning classes is the reward function, but we are also going to see the observation and action spaces. There are other aspects of the learning classes that will not be analyzed, such as how to extract the features from the state, because it is too specific of this problem, and it is a matter of working with vectors and coordinates rather than a reinforcement learning matter. In the next section we will give information about the code and where to find it, so you can check the full implementation.

We are going to see the 16 implementations that have been tried for this project. Each one of them is named 'atobX', where **atob** means "A to B" (moving the car from position A to B) and X is a number from 1 to 16. Some of the implementations were designed to just go from the car position to the ball, and that is why their features have names relating to the ball (and if they were changed, the model created would not work anymore), but they have been generalized to work to go to any coordinate of the field.

An important concept: angle to the objective point

Before analyzing the implementations of the learning classes, we are going to see a key concept that will be helpful: the angle to the point. Consider that we have two vectors: the vector that goes from the car to the point it is headed to (in the example image, the ball), and the vector parallel to the orientation of the car. The angle to the point is the angle between these two vectors. The angle can take values from -180 to 180 (when working with degrees), where a negative value means that the car is to the "left" of the point, and a positive value means that it is to the "right". Values close to 0 mean that the car is facing the point, while values near 180 (or -180) mean that it is facing the opposite direction. Instead of the vector parallel to the orientation of the car, we could consider its linear velocity vector instead.

Figure 14.m. Angle to the objective point.

Four different approaches

If we group the learning classes implementations based on the type of reward function, we can see four different groups:

- Observation comparison reward functions
- Action based reward functions
- Observation based reward function with positive rewards
- Observation based reward function with negative rewards

For each one of them, we will see the general idea behind them, and a tree will be shown with all the learning classes implemented for that approach. Implementations on top of the tree are the base for the ones below, so the tree shows how the implementations evolved. As 15 different learning classes have been implemented, we will not review them one by one, but we will review the most important changes and how they evolved. The code for all of the implementations is in annex 4.

Observation comparison reward functions

In observation comparison reward functions we compare the observations before and after taking an action. If the observation after taking the action is better than the observation before, the reward yield is positive, while if it is worse, the reward yield is negative. This means that we do not care about which actions improved or worsened the situation, but their result. If they made the situation worse, a negative reward will be yielded and the actions will be less likely to be taken. If they improved the situation, they will be more likely to be taken. Two learning classes were implemented following this approach, as seen in the figure 14.n

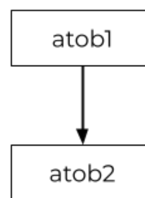


Figure 14.n. Observation comparison reward functions tree.

atob1 is the first learning class implemented. In it, we have two features: the distance and the angle to the point. For this learning class, we only consider three actions: throttle, steer and boost. In its reward function, 10 reward is given when the angle to the point is smaller than 2 degrees (absolute value), and an additional reward proportional to the distance reduced from last observation. The idea is to keep the car in that small angle space by rewarding the agent in that situation.

atob2 adds a huge reward when the car is close enough to the point it wants to reach. This way, when reaching the point, a huge reward is yield that will affect the action-values visited before reaching that point.

Action based reward functions

In action based reward function, we yield positive or negative rewards depending on the previous observation and the rewards taken. If an action taken is expected to improve the situation, a positive reward is yielded, while if they are expected to worsen the situation, a negative reward is yielded.

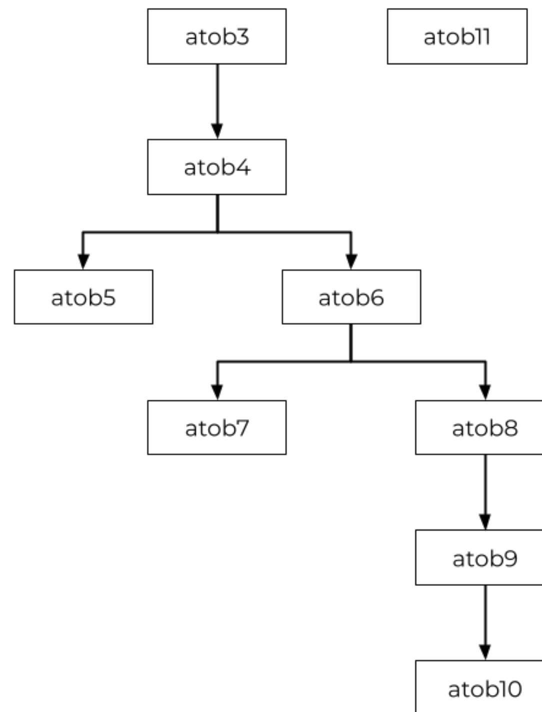


Figure 14.o. Action based reward functions tree.

atob3 is the first learning class following this paradigm. It only has one feature: the angle to the point. It still has the same three actions. When the car has a small angle to the point, it yields reward to encourage it to stay with that low angle (making it go straight to the point). When the angle is bigger, there are two options:

- The agent takes an action that will reduce the angle, either moving to the right when the angle is negative or moving to the left when the angle is positive. It also needs to throttle forward. This will yield a positive reward, but smaller than if the car was at a small angle with respect to the objective point.
- The agent takes an action that will increase the angle, either moving to the right when the angle is positive or moving to the left when the angle is negative. This will yield a negative reward.

Consequent learning classes are also based on the same concept, but introducing slight changes. Some of the implementations introduce new features, so the agent has more information about the environment. We will discuss some interesting new concepts added.

atob6 introduces a reward when the angle is small. The reward is given if there is no steering at all. This encourages the car to maintain the small angle.

atob7 yields a huge reward when reaching the objective. It is the only implementation of this approach that uses it. The distance to the objective point is added as a feature in order to be able to check this fact, and so the agent has more information.

atob8 encourages throttling forward and not steering when the angle is small in order to maintain the angle and advance to the objective. This is done by giving positive rewards proportional to the forward throttle applied (and negative when throttling backwards) and giving negative rewards proportional to the steering.

atob9 changes a little bit how the rewards work depending on the angle, and adds a fourth action: the handbrake. It allows to steer faster, reducing the angle faster.

atob10 adds yet another action: the jump. Jumping two times in less than a second and a half while pitching forwards makes the car do a front flip, which can give it additional speed. In this implementation, the speed feature is added, yielding positive reward when proportional to the speed. Big angles will reduce the reward obtained, as this speed is not used entirely to get closer to the objective.

atob11 is a different and drastic implementation, as it rewards each one of the actions (positively or negatively) depending on whether the car has a small angle with respect to the objective point or not. It also introduces a concept that we will see in the next approach: giving reward depending on how good the state is, based on the observation.

Observation based reward functions with positive rewards

In observation based reward functions with positive rewards, we give reward proportional to how good it is to be in a certain state (given the current observation). This exploits the return rewards, as ending up in good states will bring a good reward for actions taken in order to reach it, so they will be selected more often.

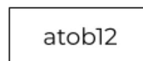


Figure 14.p. Observation based reward functions with positive rewards tree.

In this case, only the learning class **atob12** was implemented following this approach. The only change in the actions is that the car can only throttle forward. The reward function gives a huge reward when reaching the objective. It yields reward inversely proportional to the angle with the objective point, making situations with less angle better. When the angle is small enough, it yields reward inversely proportional to the distance, making the closer states to be better.

Observation based reward functions with negative rewards

Observation based reward functions with positive rewards are similar to the previous one, but the rewards yield are negative (only a positive reward is yield when reaching the goal). This is usually used when there is a need for the agent to reach the goal as fast as possible. This way, the agent goes to better states (that yield less negative reward) as fast as possible.

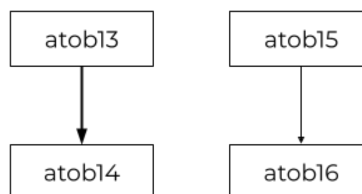


Figure 14.p. Observation based reward functions with negative rewards tree.

atob13 was the first learning class following this approach. It has the same actions as the last implementation seen. The reward function gives a huge reward when reaching the objective. It yields a negative reward proportional to the angle and the distance, making the agent try to reduce them as fast as possible.

atob14 does the same, but introducing a logarithm, that makes the rewards yielded change faster when the angle is small, giving it more precision in these cases.

atob15 gets rid of the rewards related to the distance, and focuses on the angle, using this time a square root.

atob16 adds more features. The most important is the relative velocity of the dynamic target (in case it is dynamic). This adds a degree of prediction of the dynamic target, in hopes to intercept it instead of chasing it. It also increases the reward yielded when achieving the goal, and it yields a constant negative reward in each time step. This is expected to make the agent less focused in the angle, and make it try to reach the goal as fast as possible.

14.9 Code structure and how to run it

In this final section we will do a quick review of the code available in:

https://github.com/Daniel98SP/rlrl

Code structure

In the folder tree below we can see the code organization. First, we have a folder with the slightly modified code of the RLGym framework. Then, we have a folder with the models used (**atob** is the folder containing the models studied in this text). The other folders inside **models** correspond to models created with learning classes that go beyond the analysis made in this text. The **data** folder contains statistics of the predicted episodes. Next we have the **learning_classes** folder, where the **atob** folder contains the classes studied in this text. As mentioned, there are some learning classes not discussed in this text because they were attempts to create models to achieve other objectives, but they are still in the works. In this **learning_classes** folder, we also have the file **model.zip**, which is the model that by default will be trained or predicted (if it does not exist, a file with this name is created when a new model is trained). In the **state_setters** folder we have the two types of state setters created, where the **default_state_setter.py** is the one used to train and test all the models discussed in this text. As files, we have the **main.py** and **training_environment.py** (which contains the class **TrainingEnvironment**, discussed at the start of the chapter). The **predicting_environment.py** contains the class **PredictingEnvironment**, which is similar to **TrainingEnvironment**, but adapted to predict, so it is simpler. Last but not least, we have the **calcs.py** file which contains the mathematical functions discussed in section 14.7 and the **values.py** file, which contains static information about the Rocket League field. The **requirements.txt** file is used to install the package dependencies of the code. The remaining files are work in progress. In general, files and folders highlighted in green are the ones explained in this text, or that are involved or a result of things explained in the text.

```
[project]
    [rocketleaguegym]
    [models]
        [atob]
        [levels]
        [turn]
        model.zip
    [data]
    [learning_classes]
        [atob]
        [levels]
        [shooting]
        [turn]
    [state_setters]
        default_state_setter.py
        shooting_practice.py
    main.py
    training_environment.py
    predicting_environment.py
    calcs.py
    values.py
    requirements.txt
    training_environment_2.py
    testing_environment.py
```

Figure 14.r. Code structure.

How to install and run the code

First of all, you need to have the Rocket League game on Steam or Epic Games video game distribution systems. If you do not have, it is only available on Epic Games for free, visiting this link:

https://store.epicgames.com/en-US/p/rocket-league

After installing it, you need to install a mod to it called BakkesMod. This mod allows you to use plug-ins inside Rocket League. One of the plug-ins (from RLGym) is the one responsible for sending and receiving data from the game process, as well as stopping the game in order to wait for the actions we send from the learning process. The mod can be obtained for free visiting:

https://www.bakkesmod.com/

After installing both the game and the mod, you have to execute the command “pip install -r requirements.txt” to install all the packages needed.

Finally, to start the program, you have to execute the main file using either the “training” or “predicting” arguments.

Chapter 15: Experiments

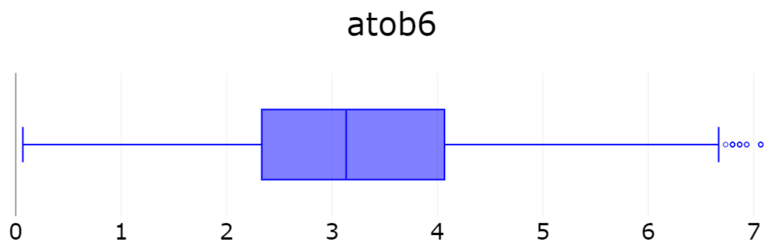
In this last chapter we will review some experiments to assess which learning class produced the best model in order to solve the problem of going from one point to another in the field. First of all, we trained all the models. To do so, we trained the models to go to the ball position, which is dynamic. In each episode, the ball and car positions were random as well as their velocities (not too fast for the ball). If the models learn how to get to the ball (dynamic position), they should also be able to reach any static position in the field.

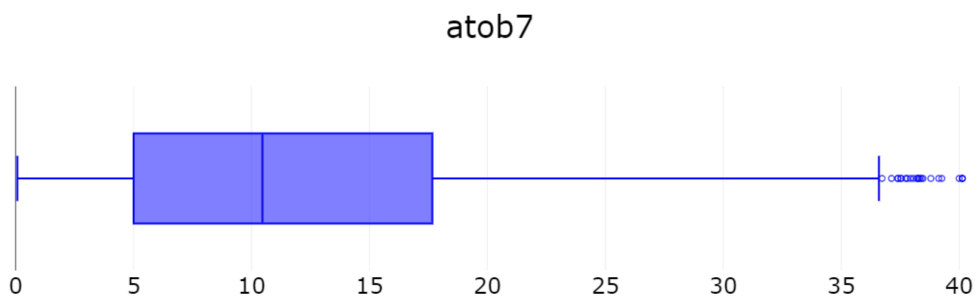
After training them, we predicted all the models that could achieve the objective consistently (from atob6 to atob15). We did predictions for 1000 episodes each model, using the same random conditions as when training the models. We calculated how many steps it took for each episode to finish (to reach the ball). We calculated the mean number of steps per episode and its standard deviation. Each second has 15 steps, so the average time that the car takes to get to the ball is the average number of steps per episode divided by 15. In the table 15.a we can see the results for each model. To calculate the statistics, we got rid of the extreme outliers. These outliers are caused by a couple of reasons. These are common for all the models, so it is safe to get rid of all of them. The first of the reasons is that, as the ball has some speed, sometimes it can end up in the net (scoring a goal), so it respawns in the center of the field. When this happens, the car has to change its trajectory suddenly. The other reason is that the car might sometimes get stuck in the net, but it is rare, and it happens for all the models.

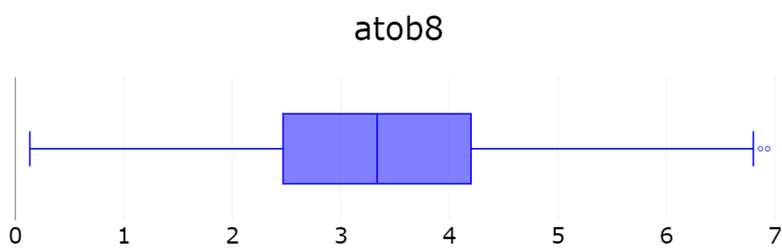
Model name	Number of samples (excluding extreme outliers)	Mean time per episode (s)	Standard Deviation
atob6	941	3.26	1.29
atob7	949	12.49	9.36
atob8	975	3.38	1.27
atob9	985	3.82	1.61
atob10	985	5.62	3.16
atob11	986	6.76	3.24
atob12	932	5.47	2.82
atob13	956	4.68	2.35
atob14	965	5.60	3.10
atob15	971	3.14	1.22
atob16	961	3.41	1.45

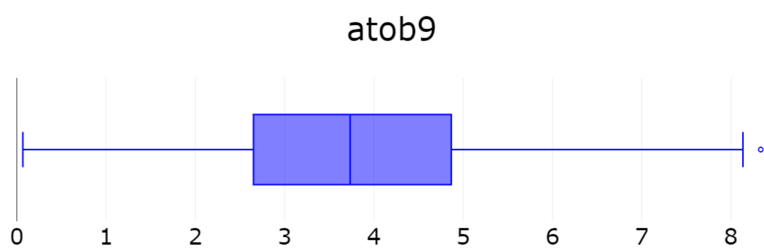
Figure 15.a. atob9 get_spaces and get_reward methods.

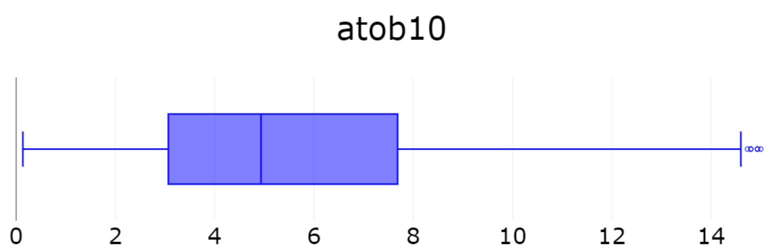
15.1 Boxplots of the results



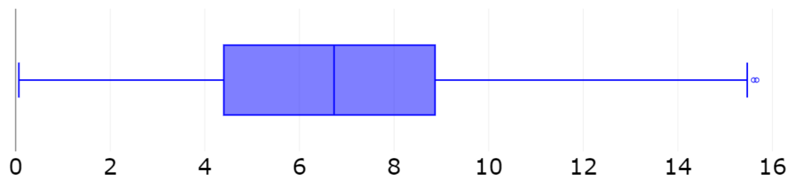




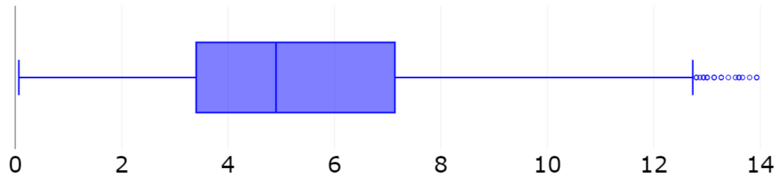




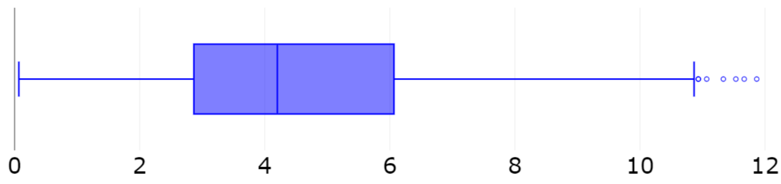
atob11



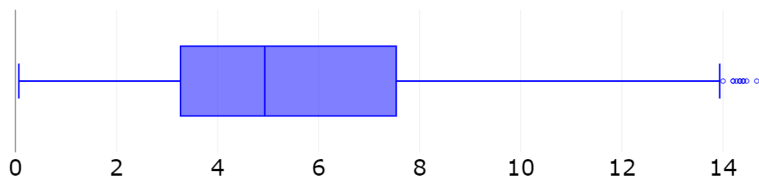
atob12



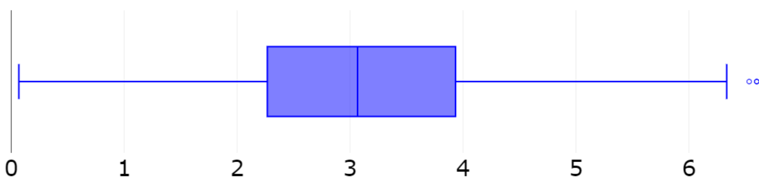
atob13



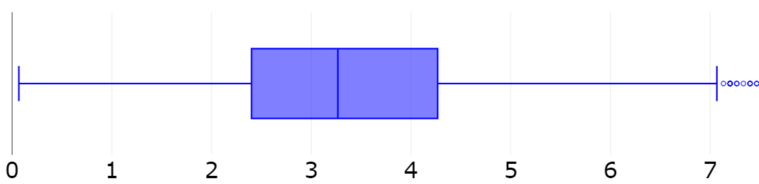
atob14



atob15



atob16



15.2 The best model

Several of the trained models have shown good results, especially **atob6**, **atob8**, **atob9**, **atob15** and **atob16**. **atob15** is the one that has the best quantitative results, as using this model, the bot takes less time to get to the target on average. **atob16** or a model inspired by promises to be the best one, as it includes more features that lets the agent know more about the environment, and as mentioned, better predict the movement of the dynamic target. This model is more complex so it needs more time to be trained. As it is still not fully trained, it is possible that at some point it obtains better results than the rest of the models trained. But, for now, we can say that **atob15** is the best model yet. In the following you can watch a video of the bot following the model **atob15**:

https://drive.google.com/file/d/1gRM6KCJnOLOdAsewiSg7gfRW8_L3L4DG/view?usp=sharing

Even if this is the best model so far, it is not perfect. As we mentioned, we need a model that can predict where the dynamic targets will move to, in order to intercept them instead of chasing them. The fact that this model is not able to predict the target's movement well, makes it miss the objective sometimes. Rocket League players, when they need to go fast and do not have boost, front flip to gain more speed. This model is not capable of doing so, as it is a relatively complex movement. In the next models, all of this will be addressed.

Part V:
Conclusions

This fifth part of the text reflects about the work done, and compares the results to what was expected. It also gives a more personal take on how the project impacted the knowledge and the working strategies previously had.

Conclusions 1: Planning changes

The development of the project has shifted from the original planning. It was expected to be finished at the end of January, but the deadline was extended until the end of April. This happened due to some issues that prevented the author from dedicating as many hours as needed, and not having the best organization.

All of this caused some of the tasks specified in the planning stage of the project to be delayed. Despite it, the tasks defined were maintained and their order respected, so it did not significantly change the cost of developing the project. The biggest cost change could be the space one. The space used to develop the project cost 88.20€ per month. With these three extra months, the total cost increase would be 264.60€.

The electricity cost was not taken into account in the initial budget, and it should have been. If we estimate that the power of the computer and display used to develop the project is about 500W, the normal price of the kWh is around 0.13€, the cost per hour would be about 0.065€. The project needed 720 hours of dedication to be completed. This adds up a total of 46.80€ of electricity cost. This is with the electricity cost at the time the project was planned. Now, the war in Ukraine and other factors have raised the electricity cost, an increase of around 2.5 times. An estimate of a third of the project will be developed under these circumstances, which means that one third of the ours will cost 0.16€, when it comes to electricity. The total estimated electricity cost is around 480h · 0.065€ + 240h · 0.16€ = **69.60€**.

Conclusions 2: Objective and scope changes

The objectives of this project changed when it comes to the practical part. This is due to the fact that they were too optimistic given the lack of knowledge at the start of the project. The objective was to create a bot capable of competing and winning against the current Rocket League bots. As it was not possible to achieve, the objectives switched to being able to lay the foundation for a Rocket League reinforcement learning bot.

Given the fact that there was no previous knowledge about reinforcement learning, the practical part of the project was a task too complex to solve in this amount of time. It would have been better to first solve an easier problem in order to grab more knowledge without being overwhelmed by the complexity of the problem.

Conclusions 3: Sustainability analysis

From an economic point of view, the project had the expected impact. Its development had some costs not calculated during the planning stage, but it did not affect how it was carried out. The project does not have a long term economic impact, as its main objective was to obtain knowledge while doing it, and resulting in something that can help others obtain it too.

From an environmental point of view, the project went as expected. The energy spent is the most important concern, as it is only a digital product that has no impact in the real world environment.

Finally, from a social point of view, the result of the project can be helpful for people that want to know more about this subject, but have not found the right material they were looking for. This project offers them a simple yet good way to have their first contact with reinforcement learning that lets them start to understand the most basic concepts, while at the same time provides them the basic knowledge to be able to start their own projects.

Conclusions 4: Technical competences analysis

CCO1.1

During the development of this project, several types of decision-making problems have been presented. Some of them were simpler (they are treated in chapter 10) and could be solved using basic reinforcement learning methods. Others, on the other hand, were more complex (they are treated in chapter 11) and required more sophisticated methods. The complexity of these problems has not been analyzed in depth, but an intuition of why some methods can solve them and why others not has been explained.

CCO2.1 and CCO2.4

This text has discussed the different machine learning paradigms (chapter 7), and why the Rocket League bot problem required a reinforcement learning solution. When it comes to reinforcement learning, this text has discussed all the basic foundations of this paradigm. The text has also shown how to develop a system capable of training a model to play a video game, or part of it (chapter 14).

CCO2.2

This project has used human knowledge to develop a system that interacts with its environment. The knowledge of the objectives that the agent has to achieve have been formalized as reward functions, which helped train models that provide intelligence to the system (chapter 14).

CCO2.6

No graphic applications have been designed, but they have been worked on and used during the project.

Conclusions 5: Personal impact of the project

On a personal level, this project has had a big and positive impact. It did not only give me theoretical and practical knowledge about reinforcement learning, but also created a great interest in me for this field that, before starting the project, was unknown to me. I started from scratch in this field, to the point that I did not know about its existence. Now, even if I still have lots of things to discover about it, I feel like I understood the basics. I'm willing to keep working on this project and others that I have in mind.

Part VI: Annexes

This sixth part of the text explains some concepts that are important to know in order to fully understand the text, but that would break its dynamics if placed along with the rest of the content.

Annex 1: Video Game Bot

A video game bot [20] is a character or a non-human player controlled by the computer. These bots follow a cycle (similar to human players) where they observe the environment and choose which actions to perform. Bots can be, for example, side characters that will act depending on the human players' actions and the changes of the environment. Bots can also, as in this project's case, try to emulate what a human player would do and even outperform them. This means that the set of actions that the bot can perform are the same as the human player's. In this case, the actions that a human player can perform are related to the buttons of the keyboard, mouse, controller, and other input devices that they can use. For this reason, the actions that bots will perform are similar to those on the input devices. The most difficult task is for the computer to choose which actions to perform given a state of the environment. There are different methods to determine it, like classic expert systems or reinforcement learning models, among others.

Annex 2: Classic Expert Systems

An Expert System [21] is a computation system that emulates the behavior of a human. Usually they consist of a set of conditions (if-then-else clauses in programming languages for the classic expert systems) of the environment in order to execute an action. In the context of Rocket league, an example would be that if the player is facing the direction towards the ball, the bot has to execute the action of accelerating. Of course this is a simplification, but would be a valid condition to execute an action.

Annex 3: More information about Rocket League

Rocket League has several options to play matches. One of them is the Session, where the player plays a ligue-like tournament against bots. In the case of having teammates, these are also bots. The team with the most wins becomes the champion of the season. Exhibition is a match option where the human player plays a game or a series of games with and against bots, and where some match settings can be changed, such as the gravity, ball size, etc. Private match is similar to Exhibition but several human players can join the game. Ranked matches are online games where only human players can participate. A victory gives you points and a defeat makes you lose them. The amount of points a player has (what is known as MMR) maps to a rank. In Unranked matches, players also gain and lose points, but no rank is shown to the player. If a player leaves the game, it is substituted by a bot until another human player enters. Finally, Tournaments are a series of online games only for human players where each team goes through a series of eliminatory rounds. Finally, it is good to know the concept of freestyling. Freestyling is performing a series of fancy movements where the spectacularity of the actions prevails above the efficiency of the movements. However, a good freestyle movement should be effective (resulting in a goal in case that is the desired outcome of the action).

Annex 4: Learning classes code

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['distance_to_point'] =
        spaces.Box(low=0, high=50000, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['angle_to_point'] =
        spaces.Box(low=-180, high=180, shape=(1,), dtype=np.float16)
05-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(3, ))

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 0
08-   if -2 < current_obs['angle_to_point'][0] < 2:
09-     reward = 10 + previous_obs['distance_to_point'][0] -
                  current_obs['distance_to_point'][0]
10-   return reward
```

Figure a4.a. atob1 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['distance_to_point'] =
        spaces.Box(low=0, high=50000, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['angle_to_point'] =
        spaces.Box(low=-180, high=180, shape=(1,), dtype=np.float16)
05-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(3, ))

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 1000 if current_obs['distance_to_point'][0] < 200
        else 0
08-   if -2 < current_obs['angle_to_point'][0] < 2:
09-     reward += 10
10-     reward += previous_obs['distance_to_point'][0] -
                current_obs['distance_to_point'][0]
11-   return reward
```

Figure a4.b. atob2 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['angle_to_point'] =
        spaces.Box(low=-180, high=180, shape=(1,), dtype=np.int16)
04-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(3, ))

05- def get_reward(previous_obs, actions, current_obs):
06-   if -2 < current_obs['angle_to_point'][0] < 2:
07-     return 10
08-   elif (previous_obs['angle_to_point'][0] < 0 and
        actions['steer'] > 0 and actions['throttle'] > 0) or
        (previous_obs['angle_to_point'][0] > 0 and
        actions['steer'] < 0 and actions['throttle'] > 0):
09-     return 1
10-   else:
11-     return -1
```

Figure a4.c. atob3 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['angle_to_point'] =
        spaces.Box(low=-180, high=180, shape=(1,), dtype=np.int16)
04-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(3, ))

05- def get_reward(previous_obs, actions, current_obs):
06-   if -0.5 < current_obs['angle_to_point'][0] < 0.5:
07-     return 10
08-   elif (previous_obs['angle_to_point'][0] < 0 and
        actions['steer'] > 0 and actions['throttle'] > 0) or
        (previous_obs['angle_to_point'][0] > 0 and
        actions['steer'] < 0 and actions['throttle'] > 0):
09-     return 1
10-   else:
11-     return -1
```

Figure a4.d. atob4 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['angle_to_point'] =
      spaces.Box(low=-180, high=180, shape=(1,), dtype=np.int16)
04-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(3, ))

05- def get_reward(previous_obs, actions, current_obs):
06-   if -0.5 < current_obs['angle_to_point'][0] < 0.5:
07-     return 10
08-   elif (-5 < previous_obs['angle_to_point'][0] < 0 and
      0 < actions['steer'] < 0.2 and actions['throttle'] > 0) or
      (0 < previous_obs['angle_to_point'][0] < 5 and
      -0.2 < actions['steer'] < 0 and actions['throttle'] > 0):
09-     return 2
10-   elif (previous_obs['angle_to_point'][0] < 0 and
      actions['steer'] > 0 and actions['throttle'] > 0) or
      (previous_obs['angle_to_point'][0] > 0 and
      actions['steer'] < 0 and actions['throttle'] > 0):
11-     return 1
12-   else:
13-     return -1
```

Figure a4.e. atob5 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['rotation_angle_to_ball'] =
      spaces.Box(low=-180, high=180, shape=(1,), dtype=np.int16)
04-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(3, ))

05- def get_reward(previous_obs, actions, current_obs):
06-   if -0.5 < current_obs['rotation_angle_to_ball'][0] < 0.5:
07-     reward = 10
08-     if actions['steer'] == 0.0:
09-       reward += 10
10-   elif (previous_obs['rotation_angle_to_ball'][0] < 0 and
      actions['steer'] > 0 and actions['throttle'] > 0) or
      (previous_obs['rotation_angle_to_ball'][0] > 0 and
      actions['steer'] < 0 and actions['throttle'] > 0):
11-     reward = 1
12-   else:
13-     reward = -1
14-   return reward
```

Figure a4.f. atob6 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-    observation_space_dictionary['distance_to_point'] =
      spaces.Box(low=0, high=50000, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['angle_to_point'] =
      spaces.Box(low=-180, high=180, shape=(1,), dtype=np.float16)
05-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(3, ))

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 1000 if current_obs['distance_to_point'][0] < 200 else 0
08-   if -0.5 < current_obs['angle_to_point'][0] < 0.5:
09-     reward = +10
10-     if actions['steer'] == 0.0:
11-       reward += 10
12-   elif (previous_obs['angle_to_point'][0] < 0 and
      actions['steer'] > 0 and actions['throttle'] > 0) or
      (previous_obs['angle_to_point'][0] > 0 and
      actions['steer'] < 0 and actions['throttle'] > 0):
13-     reward += 1
14-   else:
15-     reward += -1
16-   return reward
```

Figure a4.g. atob7 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['rotation_angle_to_ball'] =
      spaces.Box(low=-180, high=180, shape=(1,), dtype=np.float16)
04-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(3, ))

05- def get_reward(previous_obs, actions, current_obs):
06-   reward = 0
07-   if -0.5 < current_obs['rotation_angle_to_ball'][0] < 0.5:
08-     reward = +10
09-     reward += actions['throttle'] * 5
10-     reward += - np.abs(actions['steer']) * 5
11-   elif (previous_obs['rotation_angle_to_ball'][0] < 0 and
      actions['steer'] > 0 and actions['throttle'] > 0) or
      (previous_obs['rotation_angle_to_ball'][0] > 0 and
      actions['steer'] < 0 and actions['throttle'] > 0):
12-     reward += 1
13-   else:
14-     reward += -1
15-   return reward
```

Figure a4.h. atob8 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['rotation_angle_to_ball'] =
        spaces.Box(low=-180, high=180, shape=(1,), dtype=np.float32)
04-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(4, ), dtype=np.float32)

05- def get_reward(previous_obs, actions, current_obs):
06-   reward = 0
07-   if -2 < current_obs['rotation_angle_to_ball'][0] < 2:
08-     reward = +10
09-     reward += actions['throttle'] * 20
10-     reward += - np.abs(actions['steer']) * 20
11-   elif previous_obs['rotation_angle_to_ball'][0] < 0 and
        0 < actions['steer'] <
        np.abs(previous_obs['rotation_angle_to_ball'][0])/45 and
        actions['throttle'] > 0:
12-     reward += 1
13-   elif previous_obs['rotation_angle_to_ball'][0] > 0 and
        - np.abs(previous_obs['rotation_angle_to_ball'][0]) / 45 <
        actions['steer'] < 0 and actions['throttle'] > 0:
14-     reward += 1
15-   else:
16-     reward += -1
17-   return reward
```

Figure a4.i. atob9 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['angle_to_ball'] =
        spaces.Box(low=-180.0, high=180.0, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['speed'] =
        spaces.Box(low=0.0, high=1000, shape=(1,))
05-   return spaces.Dict(observation_space_dictionary),
               spaces.Box(-1, 1, shape=(5, ), dtype=np.float32)

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 0
08-   if -5 < current_obs['angle_to_ball'][0] < 5:
09-     reward = +5
10-     velocity_penalty = np.abs(current_obs['angle_to_ball'][0]) if
          1 < np.abs(current_obs['angle_to_ball'][0]) else 1
11-     reward += current_obs['speed'][0] / velocity_penalty

12-   if previous_obs['angle_to_ball'][0] < 0 and
        0 < actions['steer'] <
        np.abs(previous_obs['angle_to_ball'][0])/45
        and actions['throttle'] > 0:
13-     reward += 1
14-   elif previous_obs['angle_to_ball'][0] > 0 and
        - np.abs(previous_obs['angle_to_ball'][0]) / 45 <
        actions['steer'] < 0 and actions['throttle'] > 0:
15-     reward += 1
16-   else:
17-     reward += -1
18-   return reward
```

Figure a4.j. atob10 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['angle_to_ball'] =
        spaces.Box(low=-180.0, high=180.0, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['close_to_ball'] = spaces.Discrete(2)
05-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(5, ), dtype=np.float32)

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 10 / np.abs(current_obs['angle_to_ball'][0])
08-   if -2 < current_obs['angle_to_ball'][0] < 2:
09-     reward += actions['throttle'] * 10
10-     reward += 20 if actions['boost'] else 0
11-     reward += 100 if actions['jump'] and
          current_obs['close_to_ball'] == 0 else 0
12-     reward += - np.abs(actions['steer']) * 10
13-     reward += -20 if actions['handbrake'] else 0
14-   else:
15-     reward += -1 if actions['boost'] else 0
16-     reward += -10 if actions['jump'] else 0
17-   return reward
```

Figure a4.k. atob11 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary['angle_to_ball'] =
        spaces.Box(low=-180.0, high=180.0, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['distance_to_ball'] =
        spaces.Box(low=0.0, high=20000, shape=(1,), dtype=np.float32)
05-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(5, ), dtype=np.float32)

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 1000 if current_obs['distance_to_ball'][0] <= 200 else 0
08-   reward += 10 / np.abs(current_obs['angle_to_ball'][0])
09-   if -2 < current_obs['angle_to_ball'][0] < 2:
10-     reward += 2000 / current_obs['distance_to_ball'][0]

11-   return reward
```

Figure a4.l. atob12 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary[angle_to_point] =
        spaces.Box(low=-180.0, high=180.0, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary[distance_to_point] =
        spaces.Box(low=0.0, high=20000, shape=(1,), dtype=np.float32)
05-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(5, ), dtype=np.float32)

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 1000 if current_obs['distance_to_point'][0] <= 200 else 0
08-   reward += - np.abs(current_obs['angle_to_point'][0]) / 10
09-   reward += - current_obs['distance_to_point'][0] / 1000
10-   return reward
```

Figure a4.m. atob13 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary[angle_to_point] =
        spaces.Box(low=-180.0, high=180.0, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary[distance_to_point] =
        spaces.Box(low=0.0, high=20000, shape=(1,), dtype=np.float32)
05-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(5, ), dtype=np.float32)

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 1000 if current_obs['distance_to_point'][0] <= 200 else 0
08-   reward += - np.log10(np.abs(current_obs['angle_to_point'][0]) * 10)
09-   reward += - current_obs['distance_to_point'][0] / 1000
10-   return reward
```

Figure a4.n. atob14 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary[angle_to_point] =
        spaces.Box(low=-180.0, high=180.0, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['distance_status'] =
        spaces.Discrete(2)
05-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(5, ), dtype=np.float32)

06- def get_reward(previous_obs, actions, current_obs):
07-   reward = 500 if current_obs['distance_status'] == 0 else 0
08-   reward += - np.sqrt(4 * np.abs(current_obs['angle_to_point'][0]))
09-   return reward
```

Figure a4.o. atob15 get_spaces and get_reward methods.

```
01- def get_spaces():
02-   observation_space_dictionary = dict()
03-   observation_space_dictionary[angle_to_point] =
        spaces.Box(low=-180.0, high=180.0, shape=(1,), dtype=np.float32)
04-   observation_space_dictionary['distance_to_point'] =
        spaces.Box(low=0.0, high=20000.0, shape=(1,), dtype=np.float32)
05-   observation_space_dictionary['point_relative_velocity'] =
        spaces.Box(low=-6000, high=6000,shape=(1,), dtype=np.float32)
06-   return spaces.Dict(observation_space_dictionary),
             spaces.Box(-1, 1, shape=(4, ), dtype=np.float32)

07- def get_reward(previous_obs, actions, current_obs):
08-   reward = 1000 if current_obs['distance_to_point'][0] < 200 else 0
09-   reward += -np.sqrt(4 * np.abs(current_obs['angle_to_point'][0]))-10
10-   return reward
```

Figure a4.p. atob16 get_spaces and get_reward methods.

Part VII: References

This seventh part of the text indicates the sources of information that have helped put this text together.

[1] RLbot, 2021,
http://rlbot.org

[2] GitHub, 2022,
https://github.com/

[3] Trello, 2021,
https://trello.com

[4] Ben Aston, 2021, 9 Of The Most Popular Project Management Methodologies Made Simple
https://thedigitalprojectmanager.com/project-management-methodologies-made-simple

[5] Glassdoor, 2021, Glassdoor Salary Comparator
https://www.glassdoor.es/Sueldos/index.htm

[6] BSC, 2021, MareNostrum
https://www.bsc.es/marenostrum/marenostrum

[7] Idealista, 2021, Rental Price in Rubí
https://www.idealista.com/sala-de-prensa/informes-precio-vivienda/alquiler/cataluna/barcelona-provincia/rubi/

[8] Wikipedia, 2021, Artificial Intelligence
https://en.wikipedia.org/wiki/Artificial_intelligence

[9] Quora, 2015, What are the major areas of AI other than machine learning?
https://www.quora.com/What-are-the-major-areas-of-AI-other-than-machine-learning

[10] Wikipedia, 2022, Supervised learning
https://en.wikipedia.org/wiki/Supervised_learning

[11] guru99, 2022, Unsupervised Machine Learning: Algorithms, Types with Example
https://www.guru99.com/unsupervised-machine-learning.html

[12] DeepMind, 2022, AlphaGo
https://www.deepmind.com/research/highlighted-research/alphago

[13] Wikipedia, 2022, Markov Decision Process
https://en.wikipedia.org/wiki/Markov_decision_process#Optimization_objective

[14] Reinforcement Learning: An Introduction, 1992 Dynamic Programming (page 61)

[15] Reinforcement Learning: An Introduction, 1992 Dynamic Programming (page 61)

[16] Reinforcement Learning: An Introduction, 1992 Dynamic Programming (page 65)

[17] Wikipedia, Temporal difference learning, 2021
https://en.wikipedia.org/wiki/Temporal_difference_learning

[18] Stack Exchange, 2019, What is the relation between online (or offline) learning and on-policy (or off-policy) algorithms?
https://ai.stackexchange.com/questions/10474/what-is-the-relation-between-online-or-offline-learning-and-on-policy-or-off

[19] Youtube - DeepMind, 2015, RL Course by David Silver - Lecture 6: Value Function Approximation (12’20’’)
https://www.youtube.com/watch?v=UoPei5o4fps&list=PLqYmG7hTraZBiG_XpjnPrSNw-1XQaM_gB&index=6

[20] Wikipedia, 2021, Video game bot,
https://en.wikipedia.org/wiki/Video_game_bot

[21] Wikipedia, 2021, Expert System,
https://en.wikipedia.org/wiki/Expert_system