POLITECNICO DI MILANO
Computer Science and Engineering Master Degree
Dipartimento di Elettronica Informazione e Bioingegneria

Teaching a Learner Driver using Reinforcement Learning and Planning Strategies

Artificial Intelligence and Robotic Laboratory
of Politecnico di Milano

Supervisor: Prof. Marcello Restelli

Co-supervisors: Alessandro Lavelli, Dott.

Master thesis of:
Angelica Sofia Valeriani 940377

Academic Year 2020-2021

Abstract

Sequential decision problems are modelled through Reinforcement Learning (RL), a framework in which an agent learns to improve its decisions' quality during the interaction with the environment. The learning phase requires a lot of interactions with the environment and it may need a lot of time for very complex tasks. The concept of teaching is helpful, since the teacher is able to provide informative notions, far more representative than the information obtained with trial and error.

There are several applications in which the presence of a teacher becomes essential to significantly improve the quality of the learning process, e.g. autonomous driving, general game playing, medical applications.

This thesis will deal with the application of Teacher-Student Reinforcement Learning, i.e. the integration of a Teaching supporter in the RL framework, in the field of autonomous driving. In the context of racing cars, considering a single-driver, the goal is to complete one lap on a closed circuit in the least possible time, i.e. with the higher performance. To this extent, an inexpert student may desire to rapidly improve its driving abilities, through specific and targeted advice. For teacher definition, also planning strategies will be studied to better identify and characterize the optimal teacher trajectory.

Our approach first focuses on the offline case, in which the student trajectory is analyzed afterwards and is subject to the teacher corrections and advice. A second approach is related to the online case, in which the student is subject to teacher advice directly during the driving phase on simulator.

Sommario

I problemi di decisione sequenziale vengono modellati attraverso il framework del Reinforcement Learning (RL), in cui un agente impara come migliorare la qualità delle sue decisioni durante l'interazione con l'ambiente. La fase di apprendimento richiede una quantità di interazioni con l'ambiente molto grande e può necessitare di diverso tempo in caso di tasks ad elevata complessità. Il concetto di teaching è utile, dal momento che il teacher ha la capacità di fornire nozioni informative, che si rivelano molto più rappresentative dell'informazione che si può ottenere attraverso il trial and error.

Esistono diverse applicazioni in cui la presenza di un teacher diventa essenziale per migliorare in modo significativo la qualità del processo di apprendimento, e.g. la guida autonoma, game playing in generale, applicazioni mediche.

Questo lavoro di tesi sarà dedicato all'applicazione del Teacher-Student Reinforcement Learning, i.e. l'integrazione di un supporto con ruolo di teacher nel framework RL, nell'ambito della guida autonoma. Nel contesto delle auto da corsa, considerando un pilota singolo, l'obiettivo è il completamento di un giro di pista, in un circuito chiuso, nel minor tempo possibile, i.e. con le migliori performance. A tal fine, uno studente che non abbia esperienza potrebbe desiderare un rapido miglioramento delle proprie abilità di guida, attraverso consigli mirati. In aggiunta, per la definizione del teacher, algoritmi di planning saranno studiati, per identificare e caratterizzare la traiettoria ottima del teacher.

Il nostro approccio si focalizzerà prima sul caso offline, in cui la traiettoria dello student è analizzata in seguito al giro in pista ed è soggetta alle correzioni e ai consigli del teacher. Un secondo approccio è invece incentrato sul caso online, in cui lo student è soggetto al consiglio del teacher direttamente durante la fase di guida in simulazione.

Acknowledgements

I would like to thank Professor Marcello Restelli, my supervisor, who has given me the opportunity to undertake this path. Its guidance and creativity through all the steps of my thesis work have been precious. He represents either the most precise professionality or the greatest accuracy and passion in his work and research.

I would like to thank Dott. Alessandro Lavelli, my co-supervisor, for its constant support during my thesis work. His guidance and feedback have driven me through this journey, making me grow not only with technical skills, but also with work autonomy, self-confidence and systematic work method.

I thank all my dearest friends. The old ones, outside university context, that have supported me and that are an indelible part of my personal growth path. A special thought goes also to all my friends at University, from whom the pandemic emergency has temporarily limited in-person contact.

I thank Bernardo, for being an irreplaceable part of my life and bringing out the best part of me. Thanks for always stimulating me to improve, for understanding and supporting. No better experience than having started and finished this journey through Master's Computer Science together was possible.

The most precious acknowledgement is dedicated to my Family, in particular to my parents. Thanks for supporting me in every decision, for believing in my potentialities and having always encouraged me to give my best. Thank you for contributing to make me the person I am today, for always remembering that following my passions is the main road to happiness and teaching me that the glory in life is to make a work of my passions.

Angelica Sofia

Milan, 7^{th} October 2021

Success is not final, failure is not fatal. It is the courage to continue that counts.

- Winston S. Churchill

To my beloved parents,

Contents

List of Figures

List of Tables

Acronyms

ANN Artificial Neural Networks. 32, 34

DDPG Deep Deterministic Policy Gradient. 31, 49, 50

DP Dynamic Programming. 15, 19, 22

DPG Deterministic Policy Gradient. 31

ES Evolution Strategies. 36, 38

GP Genetic Programming. 9

MARL Multi Agent Reinforcement Learning. 43, 44, 47

MC Monte Carlo. 23, 25

MCTS Monte Carlo Tree Search. XVII, 2, 3, 40, 42, 76, 80, 81, 83, 104, 111–113, 115, 117, 118

MDP Markov Decision Process. 14–16, 19, 23, 46, 53, 60, 62, 76, 78

ML Machine Learning. 44, 45

MPC Model Predictive Control. 51, 52

NN Neural Networks. 9, 33, 76, 104

PGPE Policy Gradients with Parameter based Exploration. 3, 31, 35–39, 58, 62, 63, 68, 69, 83, 86–91, 96–102, 117

PID Proportional Integral Derivative. XVI, 65, 66

REINFORCE REward Increment Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility. 30–32, 36, 38, 69

RL Reinforcement Learning. I, III, 13–15, 17, 22, 23, 28, 43–47, 49, 51, 53, 55, 56, 117

RMSE Root Mean Square Error. 103, 104

SL Supervised Learning. 45

SPSA Simultaneous Perturbation Stochastic Approximation. 36, 37

TD Temporal Difference. 26, 27

TL Transfer Learning. 43–46

Mathematical Notation

List of adopted mathematical symbols with the corresponding meaning.

$\mathbf{x}$	Vector.
$\mathbf{x}_i$	Element i of vector $\mathbf{x}$.
$\mathbf{X}$	Matrix.
$\mathbf{X}_{ij}$	Element (i,j) of matrix $\mathbf{X}$.
$\boldsymbol{\pi}$	Matrix representing the policy of an MDP.
$\nabla_{\mathbf{x}} f(\mathbf{x})$	Gradient of function f with respect to a vector of independent variables $\mathbf{x}$.
p_x	Probability density function of a random variable x.
$p_x(\cdot\|y)$	Probability density function of a random variable x conditioned by the value of the random variable y.
$\mathbb{E}_{x\sim p}[f(x,\dots)]$	Expected value of a function $f(x,\dots)$ with respect to the random variable x (or random vector if $\mathbf{x}$) distributed according to the probability density p.
$\mathbb{E}_x[f(x,\dots)]$	Abbreviation of $\mathbb{E}_{x\sim p}[f(x,\dots)]$ in case no ambiguity arises.
$\mathbb{E}_\pi[\cdot]$	Expected value of a random variable given that the agent follows policy π.

Probability density function of the reward is denoted both with R and r, not only the lowercase letter. The same holds for the transition distribution, denoted with P as well as p.

List of adopted conventions concerning Roman and Greek letters. Given a state s_t, at time t, next state is denoted indifferently with s_{t+1} and s'. The same holds for action a_t at time t. Next action is denoted both with a_{t+1} and a'.

T	Time horizon.
γ	Discount factor.
α	Learning Rate.
S	Set of states for MDP definition.
A	Set of actions for MDP definition.
$\boldsymbol{\theta}$	Policy parameter vector.
Ψ	Set of critical sectors of the tracks (i.e. sectors that are identified for boost steer).
$\tau(\mathbf{s}, \mathbf{a})$	Transition Model with input parameter vector of states $\mathbf{s}$ and vector of actions $\mathbf{a}$.
ω	Set of Transition Model Parameters.

Chapter 1

Introduction

1.1 Motivation and Goal

When driving with a racing car, having the best setting in terms of driving style, e.g. deciding the exact point in which to steer, the intensity of steering, when and how much pushing the accelerator, is fundamental. Such features are elements that determine significant time gains over the vehicle performance on the track. Car manufacturers devote a great effort and resources to improve the driving experience of their purchasers and affectionate.

For a sport car owner, it is often difficult to drive on a track getting the best possible time and performance. For this reason, the development of a virtual coach, i.e. teacher, that can teach to sport car enthusiasts how to drive in real world is very important, since it would provide better performance as well as major safety conditions and mastery of the vehicle. The need to improve the driving style arises from the wide spread of racing cars and high performance cars as private cars, not related only to the racing car competition context. The same reasoning can be applied to video games, i.e. to create learning features for inexperienced gamers.

The mere knowledge of the optimal trajectory is unfortunately not enough to teach correctly to an inexperienced student. Teaching procedure is complicated, since the ability of teaching requires not only techniques that learn how to perform the optimal lap, but also techniques that allow to teach it to others.

In this context, the goal of this thesis project is to develop an autonomous agent, called Student, able to learn with the support of an expert entity, called Teacher, able to suggest changes in the driving style to the driver. The policy student is defined as simple and is based on a trajectory to follow. Then, the approaches of Teacher-Student Reinforcement Learning have been

studied and applied in order to simulate the learning process between the learner and the teacher part. In particular, we focused on the development of a planning algorithm capable of finding the optimal trajectory. The goal is to optimize the student policy to guarantee the most effective learning and to refine the Teacher, for the selection of the appropriate learning method to allow the final Student driver to get the most appropriate advice to improve its driving style. The final part focuses on the teaching algorithm that, given a policy and an optimal trajectory, seeks to transfer knowledge to the student.

1.2 Original Contribution

We investigate the structure of the most proper student policy to develop a parametrized policy, able to efficiently learn. The teaching approach that we study is based on planning. To this extent, offline planning method with Monte Carlo Tree Search (MCTS) has been applied and adapted to our context. The transition model applied for the MCTS prediction adopts a multi-step loss function, to outperform the effects that can be obtained in planning when the standard one-step loss function is applied. The final goal of the teaching algorithm, given the transition model and a policy for action basic selection, is to update the student policy parameters to correct its driving style. We perform an effective study of the student optimization, transition model definition and learning process with TORCS simulator.

1.3 Thesis Outline

The content of this thesis is organized in the following chapters.

Chapter 2 introduces the context in which this thesis work is applied. In Section 2.1 a complete overview on TORCS simulator, the adopted strumentation to perform evaluation, is described and detailed. In Section 2.2 an overview of analogous use of TORCS simulator in the field of autonomous driving is presented.

Chapter 3 introduces exhaustively the preliminary concepts required to understand the content of this thesis work. In Section 3.1, we provide a complete introduction to Reinforcement Learning, including the fundamental basis of Markov Decision Process and the adopted methodologies to solve them. In Section 3.2, we provide a description of Supervised Learning methodologies and in particular of Artificial Neural Networks, that will be adopted in this thesis. In Section 3.3, we present the fundamental algorithms

that are commonly adopted in the Reinforcement Learning context, in comparison to Policy Gradients with Parameter based Exploration (PGPE), the algorithm that is adopted in this work. We present the fundamentals of PGPE and a taxonomy of the related methodologies. In Section 3.4, we provide an introduction to planning and we present MCTS, the adopted algorithm for the planning phase.

Chapter 4 presents the state-of-the-art related to the application context of this work. In Section 4.1, we introduce Transfer Learning, as a related topic to Teacher-Student Reinforcement Learning, providing a complete overview of this field. In this Section, variations and optimizations of Transfer Learning framework are described. In Section 4.2, the Teacher-Student Reinforcement Learning framework is introduced. In Section 4.3, we deepen the study related to the autonomous driving field in literature. In this context, algorithms and works that are related to the task of learning on track are presented. Then, an overview on planning algorithms adopted on track is presented.

Chapter 5 presents the formalization of the problem we face in this thesis work. In Section 5.1, the Environment of TORCS is formalized according to our context. In Section 5.2, the Student policy formalization is detailed, with a taxonomy on the adopted parametrization. In Section 5.3, we discuss the Teaching phase, presenting the planning structure and learning algorithm adopted to perform teaching.

Chapter 6 presents the methodology and approach that we adopted to tackle the problems defined in Chapter 5. In Section 6.1, we present the Environment according to the formalization model. In Section 6.2, we present the *student* policy structure. In Section 6.3, we present the teaching framework, defining both the planning phase and the teaching approach.

Chapter 7 presents the experimental results for the proposed student optimization policy, the planning phase and the learning process of the teacher framework.

Chapter 8 presents conclusions that can be drawn from our work, and proposes areas and themes that can be object of future research work.

Chapter 2

Context

This thesis work takes place in the scope of autonomous driving applied in virtual environments, i.e. video games and simulators. In particular, our goal is to design an autonomous driver able to achieve competitive lap-time compared to human drivers. This task is a key problem in real-world racing car as well as in the development of non-player characters for a commercial racing game. The optimal racing line is defined as the line to follow to achieve the best possible lap-time on a given track with a given car. Autonomous racing in video games is a common practice since the birth of the first video games consoles in the 90's. Sony, Namco, Electronic Arts are just some names of the most active and notorious commercial distributors of the most played racing video games. Along with the evolution of technology, virtual driving styles have seen constant improvement in human-like skills. Over the years, the so called "bots" have improved their driving capabilities reflecting the state-of-the art theoretical and technological innovations that the scientific community contributed to. The latest promising topic in this field is Machine Learning, and in particular Reinforcement Learning, which is based on an agent learning from its errors to improve its performance. We will discuss these techniques in detail in the next chapters. However, the state-of-the art technologies upon which autonomous driving is based refer more to traditional AI algorithms. These involve expert's rules encoded into the program, while Machine Learning solutions are still at research and experimentation stage. To address the challenge of autonomous racing, we choose to adopt a simulator. This is due to many reasons: a simulator is much simpler than the real world, and the task we are going to address is much easier to manage in a controlled environment such as a simulator. Safety is another aspect to take into consideration: training an autonomous vehicle involves a lot of trial-and-error, and an error could be very dangerous

- if not fatal - especially at the speed that racing cars typically go. Finally, training an autonomous vehicle in the real world is a long and costly operation: it involves a car that races a lot of times, and a computer elaborating data with energy-consuming GPUs. Using a simulator is a sensitive choice for training, in order to deploy afterwards the final algorithm in a real-world scenario. However, transferring the policy obtained by means of a simulator to the real world is a hard task, due to the intrinsic differences between them, which makes the real world far more complex than a simulator. For instance, the computed parameters of the equations ruling the world and the actions to fulfill a task may be different compared to the real ones, thus they may need some fine tuning when embodied into the machine. This is due to the approximations and all the aspects which are not taken into account by the model of the simulator. Today there exist several racing-oriented driving simulators, along with development environments. Some examples are AWS Deep Racer [5], TORCS, Racer [19].

2.1 TORCS

TORCS [89], The Open Racing Car Simulator, is a state-of-the-art open source racing car simulator. The TORCS project was originally created by Eric Espié and Christophe Guionneau and it is now continued by Bernhard Wymann, Christos Dimitrakakis and other collaborators. This simulator, written in C++, allows the user to drive and to develop automatic drivers, called robots or bots. A bot can control a car using gas and brake pedals, gear shift and steering wheel. TORCS features a detailed physics engine that presents realistic aerodynamics, wheel properties, damage and collision model, etc. The game-play allows different types of races and a wide range of different tracks and vehicles, and it provides a rather sophisticated 3D graphics for visualization. For these reasons, TORCS is not only used as a game and has also become quite widespread as research platform in the field of racing car. Figure 2.1 depicts a frame of our simulation context. TORCS comes as a standalone application in which robots are programmed in C++, so it restricts the choice of the programming language. The work of Loiacono et al. [50] for the 2009 Simulated Car Racing Championship proved to be very useful for the scientific community. Their software for the competition [81] extends the original TORCS architecture structuring it as a client-server application: the driving algorithm can run separately from TORCS and can communicate with its server through UDP connection. The server job is to send information of the TORCS environment to the client, to receive the action and to apply it to the environment. Figure

Figure 2.1: TORCS: Simulation Context [89]. There is the possibility to play with different types of races and a wide range of different tracks and vehicles. TORCS also provides a rather sophisticated 3D graphics for visualization.

2.2 shows the software architecture. Server bots manage the connections with external clients through UDP. This extension enables programmers to use different programming languages, different than C++, more suitable for their needs. For this thesis, we use TORCS version 1.3.7. For interfacing with the simulator we use SnakeOil. It is a Python library that manages the UDP connection with the server, enabling an agent (or driver) to act as client bot in the client-server architecture. It takes care of receiving the information coming from the server, pre-processing it in order to be ready for our agent and sending back the computed action to control the car. The competition architecture creates a separation between the simulator and the agent. What the driver perceives of the racing environment and the available actions are defined in the server bot. The server sends to the client, via the UDP, a list of virtual sensor-readings (e.g. the track, the speed) that measures the environment. The agent receives the sensor-reading as input and responds with a computed action to control the car. Afterwards, the server job itself actuates the action on the environment. The actions available to control the car are a rather typical set of actuators, i.e., the steering wheel, the gas and break pedals and the gearbox.

2.2 Autonomous Driving and Simulation approaches

One of the greatest achievements and most sophisticated applications of computerized kinematic and dynamic simulations is the realization of driv-

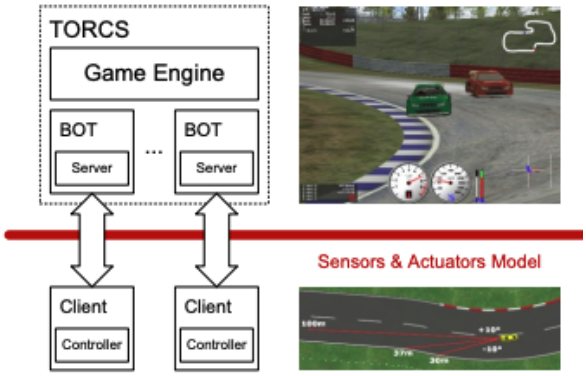


Figure 2.2: Client-Server architecture of TORCS [89]. Extension of the original TORCS architecture structured as a client-server application, in which the driving algorithm can run separately from TORCS and can communicate with its server through UDP connection.

ing simulators. Advanced driving simulators are used in research and industry, and may have many applications, e.g. vehicle design, human factors studies as driver behaviour. They provide a controlled environment for testing, but also a source of application involving levels of performance that are increasing in time. One of the first research work about driving simulation proposed the adaption and evolution of neural networks in a homemade racing game to drive a single car as fast as possible on a single track [82]. In this work Togelius J. et al. focus on the identification of the best suited controller architectures for optimization in a simple racing game. Furthermore, a variety of papers focused on different versions of the same considered experimental setup, e.g. learning of racing skills on multiple tracks, competitive controllers co-evolution, human driving styles imitation, and new racing tracks evolution to suit human driving styles.

In [83], computational intelligence is applied to racing games and a summary of all described developments is proposed. An interesting overview regarding various computational intelligence challenges that racing car games may face is presented. In particular, the first challenge regards optimization, i.e. to optimize a given setup or strategy. The second challenge is related to innovation, i.e. starting from a given input-output space, to develop something new or original. The last one is devoted to imitation, i.e. to optimally emulate player strategies. Another interesting point presented in [83] is related to the faced challenges in the development of individual racing controllers or in actual competition, as well as to the generation of

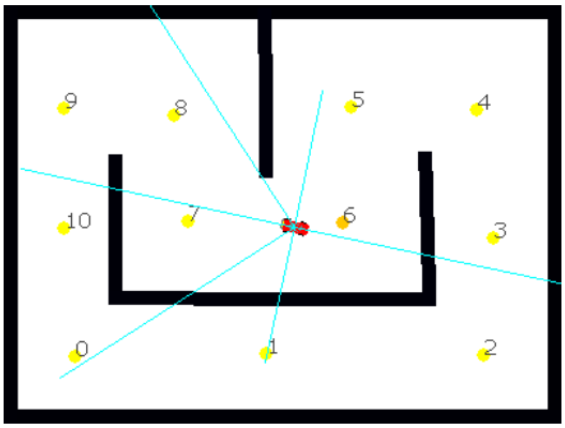


Figure 2.3: A single car and one of the easier tracks in the track-based racing game [80]. Circles are way points and the lines protruding from the car represent wall sensors.

new tracks and advanced modeling both for driving behaviours and actual players.

In [80] a similar overview is presented. Different methods for simulation are tested. The track-based racing game that is exploited to overview different methods is shown in Figure 2.3. Furthermore, the approach in [80] considers as an alternative to Neural Networks (NN) also Genetic Programming (GP), i.e. mostly *expression trees* (trees in which both terminal and non-terminal nodes have children that can be themselves terminal or non-terminal) [44]. The other adopted methods, MLP, refers to Multi-Layer Perceptron, a particular type of NN [53].

Specific approaches in imitation of human driving style have been proposed in [15, 86]. In [15], TORCS racing simulator has been used for training agent drivers through imitation learning adopting supervised models of human drivers. In Figure 2.4, the new sensory representations introduced in [15] are presented. The rangefinder projects some rays around the car, with a range of 100m, and returns the distance between the car and the edge of the track. The second new sensor model, i.e. look-ahead sensors, represents the state of the car using only the radius of curvature of the track segments ahead. The model consider next h track meters, dividing them into equal length segments, e.g. 10m [15]. For each segment, look-ahead sensor returns a bending radius. The applied codification is 0 for rectilinear traits, positive radius for a right turn, negative radius for left turn. In Figure 2.4, the 80 meters ahead are shown, while the car approaches a right turn. The authors in [15] also test on different tracks the same bot. In Figure 2.5 the different

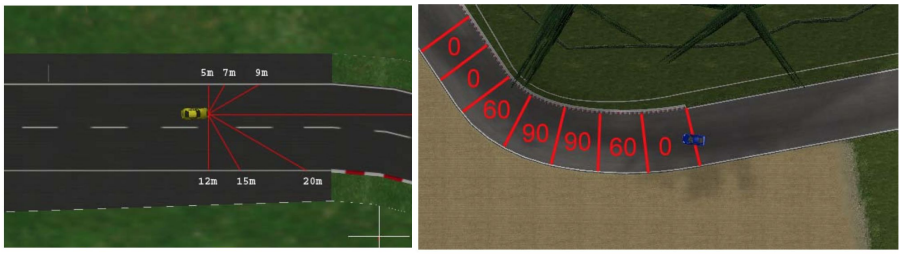
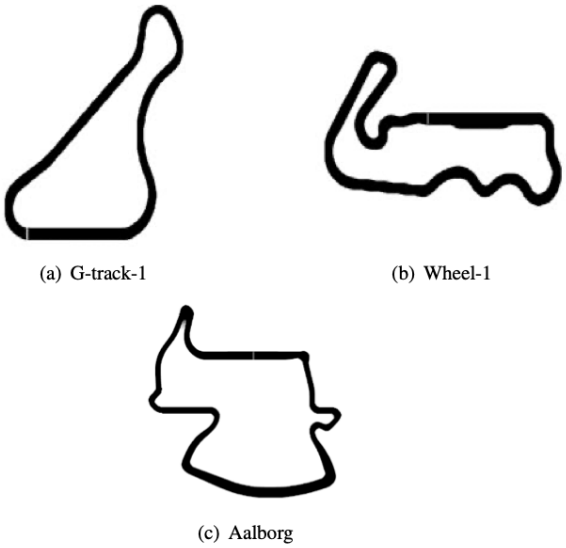


Figure 2.4: Sensors from [15]. Rangefinder Sensor (left). The rangefinder projects a series of rays, with a range of 100m, around the car, returning the distance from the car to the edge of the track. Look-ahead Sensor (right). The look-ahead sensors represent the state of the car using only the radius of curvature of the next track segments.

tracks are shown. *G-track-1* is a simple track with interesting trajectories in case of fast driving, *Wheel-1* is a more complicated track presenting a series of fast turns, *Aalborg* is a difficult track with many tight and slow curves. A similar study on human driving style is performed in [86], in which van Hoorn et al. adopt Multi-Objective Evolutionary Algorithm (MOEA) [20, 24] to obtain a final driving style that is a balance between driving effectively and human-style driving, i.e. an analogous effect to recorded human drivers play traces.

A different perspective, more related to the generation of interesting driving, is presented in [2]. In this work Agapitos et al. study an evolution of Non-Player Character (NPC) for procedural context generation in games [2]. The goal is to automatically produce populations of opponents and collaborators, to be diverse in behavioural space. Many researchers in this field, study also the application of MOEA algorithms to this task [84, 92, 62]. Some of the experiments carried out in [2] refer to optimization for wall collisions and car collisions with opponents driver. In Figure 2.6 experimental results concerning wall and car collisions are shown; the learner is identified with red, while the opponent in blue [2]. First goal is to evolve a controller that is able to learn to keep an enough safe distance from the wall. The goal is hence to minimize wall collisions. In (b) evolved controller fails this aim since it keeps an aggressive behaviour, high speeds and crashing in turns. On the contrary in (c), a smooth trajectory is generated by keeping a low speed, without generating any collision. Not enough progress in performance is reached because of too low speed.

The research work in [16] moves the direction of simulation for autonomous driving in the set of online learning. In this research work, the authors propose an online neuroevolution approach to face the trade-off of



(a) G-track-1 (b) Wheel-1

(c) Aalborg

Figure 2.5: Examples of used tracks for training in [15]. G-track-1 (a), Wheel-1 (b) and Aalborg (c). G-track-1 is a simple track with interesting trajectories in case of fast driving, Wheel-1 is a more complicated track presenting a series of fast turns, Aalborg is a difficult track with many tight and slow curves.

exploration, i.e. looking for better unknown solutions, and exploitation, i.e. exploitation of the most promising solution known so far. Cardamone et al. in [16] use the approach of Neuro-Evolution of Augmenting Topologies (NEAT) [74], that is developed to evolve neural networks without assuming prior knowledge, neither on connections nor on network topology. In this context, both the task of evolving a fast controller from scratch and the task of optimizing an already existing controller for a new track are considered according to online strategy. Different online evaluation strategies to maximize performances during learning have been studied, with main focus on action selection.

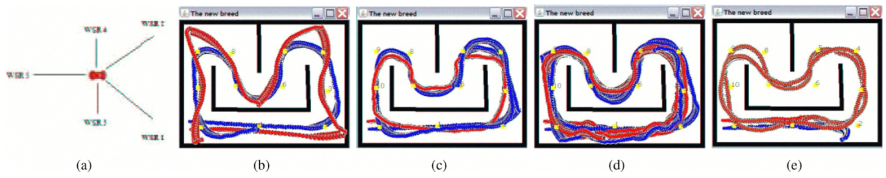


Figure 2.6: Experimental results from [2]. (a) Wall-sensors setup; Movement Traces (learner in red, opponent in blue): (b) max speed, wall-collisions; (c) low speed, no wall collisions; (d) car-collisions-provoking driver; (e) opponent weakness exploitation.

Chapter 3

Theoretical Background

3.1 Reinforcement Learning Background

Together with Supervised Learning and Unsupervised Learning [16], RL is a branch of Machine Learning, the discipline which studies the way a computer program can learn from experience and improve its performance at a specified task. While Supervised Learning algorithms are based on inductive inference (where the model is typically trained using labelled data to perform classification or regression) and Unsupervised Learning exploits techniques of density estimation and clustering applied to unlabelled data, in the RL paradigm an autonomous agent learns what to do (how to map situations to actions) in order to maximize a numerical reward signal by interacting with the environment. Typically, the RL agent is not explicitly taught how to act by an expert, but it must discover which action yields the largest reward in a trial-and-error procedure, the so called exploration phase. As the learner gathers experience, it becomes able to distinguish good actions from bad actions and therefore it can exploit its knowledge to reach the goal of maximizing the reward signal received at each step of the interaction. The overall interaction is illustrated in Figure 3.1. The challenge of RL is to design the best trade-off between exploration and exploitation, that is the capability of an agent to use its knowledge to obtain high rewards and, by contrast, being capable of exploring new possibilities in such a way not to remain stuck in a local optimum. Intuitively, the exploration process is high at the first iterations of the learning process, while it should decrease with accumulating knowledge.

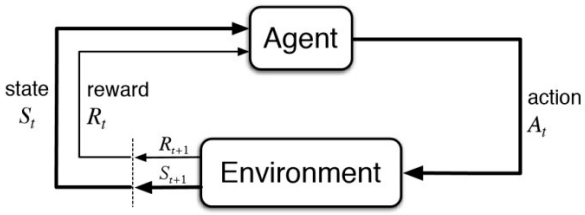


Figure 3.1: Agent-Environment interaction in a Reinforcement Learning task [75]. At each time step t, the agent receives a representation of the state $S_t \in S$ of the environment, based on which it selects an action $A_t \in A$ to perform. In the following time step $t+1$, the agent receives a numerical reward $R_{t+1} \in \mathbb{R}$ based on the goodness of its decision.

3.1.1 Markov Decision Process

An RL setting is composed by an agent, that is the learner and the decision maker, which interacts with an environment. The environment is the mathematical representation of the world in which the agent operates. It reflects the set of features of a machine learning problem, which could be numerical (coordinates, velocities, acceleration) or raw (such as images or signals). At each time step t, the agent receives a representation of the *state* $S_t \in S$ of the environment, based on which it selects an *action* $A_t \in A$ to perform. In the following time step $t+1$, the agent receives a numerical reward $R_{t+1} \in \mathbb{R}$ based on the goodness of its decision. Then, it gets in a new state S_{t+1} (see Figure 3.1). The sequence of states and actions is called trajectory. The RL environment can be formalized as a Markov Decision Process (MDP) [59]. An MDP is a tuple $\langle S, A, p, R \rangle$ composed by:

- Set of states S: $S = \{s \in \mathbb{R}^n\}$ is the set of possible states in the environment;
- Set of actions A: it is the set of actions the agent can perform. Generally, the set of possible actions can depend on the actual state of the environment, thus, we define it as function of the state, $A(s)$;
- Transition probability distribution can be stochastic or deterministic. It is defined as $p(S_{t+1} = s' | S_t = s, A_t = a)$, that indicates the probability of going into state s', from state s, by performing action a;
- Reward function $R(s, a) = \mathbb{E}[\, R_{t+1} | S_t = s, A_t = a]$, the reward received after taking action a in state s.

We consider the case in which the environment and the behavior of the agent satisfy the *Markov property*, i.e., the next state of the environment does not

depend on the past states and actions given the current state and action. Furthermore, the decision of the agent is determined exclusively as a function of the current environment state. A finite MDP (i.e. a MDP with finite set of states and finite set of actions), having a reasonable number of states, can be solved through Dynamic Programming (DP) [75]. However, in real-world scenarios, typically we do not have full knowledge of the environment, or it is too complex to be solved through such methods. In these cases, RL can be used. By contrast, when the dynamics is not known before, i.e. transition model is unknown, or it is too complex, i.e. there are too many state to consider, RL can be used. In fact, RL operates by receiving only samples of the process.

3.1.1.1 Return

Every time the agent performs an action on the environment it receives a reward signal $R_t \in \mathbb{R}$ that determines the goodness of taking that action on that specific state. The goal of the agent is to maximize the expected value of the cumulative sum of a received reward G_t, called *expected return*, that is computed starting from timestamp t. The task can be *episodic* or *continuous*. In the former case, the learning process is subdivided in *episodes*, all of which terminate with a *terminal state* at time T, where T is a stochastic variable. In the latter, by contrast, there is no terminal state, but the learning process can go on indefinitely. Thus, the expected return G_t is changed by adding a discount factor that avoids G_t to diverge to infinity. The return is a function of the reward sequence:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{3.1}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called *discount factor*. The discount factor influences how the agent considers future and present rewards. A small γ results in a higher consideration of present rewards than the future, leading to a *myopic* maximization of reward, while with a γ near to 1, the agent will consider equally the reward coming either from the present or from the future, becoming more *farsighted*.

3.1.1.2 Policies and Value Function

The goal of reinforcement learning is finding the optimal policy π, which maps states to a probability distribution over the actions, in order to maximize the return. A deterministic optimal policy always exists, and in general

an optimal policy can be deterministic or stochastic. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ given $S_t = s$. To allow policy improvement, we must first evaluate the given policy. Evaluating the goodness of a policy is essential to make optimization over it.

The *state-value function* of a state s under a policy π, denoted $v_\pi(s)$ is defined as the expected long-term return starting from the current state and choosing actions with policy π:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \tag{3.2}$$

for all $s \in S$, where $\mathbb{E}_\pi[\,\cdot\,]$ denotes the expected value of a random variable given that the agent follows policy π.

The *action-value function* for a policy π is defined as the value of taking action a in state s and following π:

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]. \tag{3.3}$$

The Q-value represents the possible reward received in the next time step for taking action a in state s, plus the discounted future reward received from the next state-action observation. A fundamental property of value functions is that they satisfy a recursive relationship that allows us to reformulate it as the so called *Bellman Equation*:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[\, G_t | S_t = s] \; = \mathbb{E}_\pi[\, R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[\, r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']\,], \end{aligned} \tag{3.4}$$

When the MDP is finite, the *Bellman Expectation equations* become matrix equations and can be solved to find the value functions. For $v^\pi(s)$ and $q^\pi(s,a)$ the *Bellman expectation equations* can be rewritten in matrix notation as:

$$\begin{aligned} \mathbf{v}^\pi &= \left(\mathbf{I} - \gamma \mathbf{P}^\pi\right)^{-1} \mathbf{r}^\pi, \\ \mathbf{q}^\pi &= \left(\mathbf{I} - \gamma \mathbf{P}^\pi\right)^{-1} \mathbf{r}, \end{aligned} \tag{3.5}$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{P}^\pi$ is a matrix having as rows the probabilities of transitioning. $\mathbf{P}^\pi = \boldsymbol{\pi}\mathbf{P}$ is the state transition matrix, whose dimensions are $|\mathcal{S}| \times |\mathcal{S}|$. It is a stochastic matrix and every row has terms that sum up to 1. $\boldsymbol{\pi}$ is the matrix denoting the policy of an MDP. $\mathbf{r}^\pi$ is a vector $\in \mathbb{R}^{|\mathcal{S}|}$ that

includes the rewards. It is defined as $\mathbf{r}^\pi = \boldsymbol{\pi}\mathbf{r}$, in which $\mathbf{r}$ is a vector $\in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$. $\mathbf{v}^\pi$ is the vector of *state-value function* of policy π, i.e. $\mathbf{v}^\pi \in \mathbb{R}^{|\mathcal{S}|}$. *Action-value* function is a vector $\mathbf{q}^\pi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$. Computing the state-value function requires a computational effort of $\mathcal{O}\left(|\mathcal{S}|^3\right)$, as imposed by the inversion of the matrix $(\mathbf{I} - \gamma\mathbf{P}^\pi)$. The Bellman equation expresses the relationship between the value of a state and the values of its successor states, and the value function v_π is the unique solution of its Bellman equation when $\gamma < 1$. The goal of RL is to find a policy π that maximizes the expected return on the long run. Any policy fulfilling this property is called *optimal policy* π^*. The value function of the optimal policy can be a proxy to order policies in terms of optimality. The state-value function corresponding to the optimal policy is called *optimal state-value function*, defined as:

$$v_*(s) = \max_\pi v_\pi(s), \text{ for all } s \in S. \tag{3.6}$$

Similar concept for the *optimal action-value function*, defined as:

$$q_*(s,a) = \max_\pi q_\pi(s,a), \text{for all } s \in S, \text{ for all } a \in A. \tag{3.7}$$

For the state-action pair (s,a), this function represents the expected return of taking action a in state s and, thereafter, following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$\begin{aligned} v_*(s) &= \max_{a\in\mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_{a\in\mathcal{A}(s)} \mathbb{E}_{\pi_*}\left[G_t|S_t = s, A_t = a\right] \\ &= \max_{a\in\mathcal{A}(s)} \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a\right] \\ &= \max_{a\in\mathcal{A}(s)} \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}|S_t = s, A_t = a\right] \\ &= \max_{a\in\mathcal{A}(s)} \sum_{s'} p\left(s'|s,a\right)\left[r + \gamma v_*(s')\right]. \end{aligned} \tag{3.8}$$

Intuitively, the *Bellman optimality equation* expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state. In an analogous way, the *Bellman optimality equation* for q_* is:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a\right] \\ &= \sum_{s'} p\left(s'|s,a\right)\left[r + \gamma \max_{a'\in\mathcal{A}(s')} q_*(s',a')\right]. \end{aligned} \tag{3.9}$$

The quantities derived with Bellman equations can be computed in a recursive way, by exploiting the *Bellman Expectation Operator* for V^π, defined as $T^\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$, that maps value functions into value functions:

$$(T^\pi V^\pi)(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s'). \right) \tag{3.10}$$

Using the Bellman Operator, the *Bellman Expectation equation* can be compactly rewritten as:

$$T^\pi V^\pi = V^\pi. \tag{3.11}$$

V^π is defined as a fixed point of the Bellman Operator T^π [59]. In the same way, also the *Bellman Expectation Operator* for Q^π can be defined. It is defined as $T^\pi : \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|} \to \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$, and maps action-value functions into action-value functions:

$$(T^\pi Q^\pi)(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s',a'). \tag{3.12}$$

Q^π is defined as a fixed point of the Bellman Operator T^π, as for the value function V^π. Using the Bellman Operator T^π, the *Bellman Expectation equation* can be compactly rewritten as:

$$T^\pi Q^\pi = Q^\pi. \tag{3.13}$$

Both operators, Bellman Operator for V^π and Q^π are linear. Besides, they satisfy the contraction property in $L_\infty-$norm, i.e. $\|T^\pi f_1 - T^\pi f_2\|_\infty \leq \gamma \|f_1 - f_2\|_\infty$, therefore, the iterative application of T^π allows any function to converge to the value function:

$$\begin{aligned} \lim_{k\to\infty} (T^\pi)^k f = V^\pi, \forall f \in \mathbb{R}^{\mathcal{S}}, \\ \lim_{k\to\infty} (T^\pi)^k f = Q^\pi, \forall f \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}. \end{aligned} \tag{3.14}$$

This property is exploited in Section 3.1.2.1 by Policy Iteration algorithm 1, in *iterative policy evaluation.*

Bellman Operators are defined also for the optimal state-value function and optimal action-value function. *Bellman Optimality Operator* for V^*, defined as $T^* : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$, that maps state-value functions into state-value functions:

$$(T^* V^*)(s) = \max_{a \in \mathcal{A}(s)} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right). \tag{3.15}$$

Bellman Optimality Operator for Q^*, defined as $T^* : \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|} \to \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$, that maps action-value functions into action-value functions:

$$(T^*Q^*)(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}(s')} Q^*(s',a'). \tag{3.16}$$

As for V^π and Q^π, also V^* and Q^* are the unique fixed point of respective *Bellman Optimality Operators*. Besides, it must be noticed that linearity property is lost for the *Bellman Optimality Operators*, because of the maximum function. This implies that no closed form solution exists. Both operators satisfy the contraction property in $L_\infty-$norm, and also in this case, the iterative application of the operators guarantees that any function converges to the optimal value function.

$$\begin{aligned} \lim_{k\to\infty} (T^*)^k f = V^*, \forall f \in \mathbb{R}^{\mathcal{S}}, \\ \lim_{k\to\infty} (T^*)^k f = Q^*, \forall f \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}. \end{aligned} \tag{3.17}$$

3.1.2 Dynamic Programming

Solving a MDP means to find the optimal policy. *Brute force*, i.e. the most *naïve* approach, implies the enumeration of all possible policies, evaluating their performances and then returning the best one. The issue of this method is its inefficiency for the case of finite MDPs since the number of policies is exponential $\left(|\mathcal{A}|^{|\mathcal{S}|}\right)$ and not applicability to the case of infinite MDPs. Here, we briefly outline the approach of DP, that is the most common method to solve a MDP when there is knowledge about the model of the environment. DP is a general approach that can be subdivided into sub-problems. The basis is the principle of optimality, i.e. the original problem is solved once the sub-problems are solved. The property of Bellman equations (3.4), (3.8), (3.9), of being recursive is suited to this extent. The common techniques of DP are Policy Iteration and Value Iteration. In Figure 3.2, it is shown that DP considers all possible actions and states, then uses values of the states reached in one step, averaging them to update the starting state value. Starting point is state s_t, white circles are the states, black dots are the actions and green squares represent the termination. The following update rule for the value function is used:

$$V^{(t+1)}(s_t) = \mathbb{E}_\pi \left[r_{t+1} + \gamma V^{(t)}(s_{t+1}) \right], \tag{3.18}$$

with r_{t+1} the immediate reward obtained when going from state s_t to state s_{t+1}. Dynamic Programming uses the full knowledge of the model and makes update accordingly. This approach is different from the ones that will be

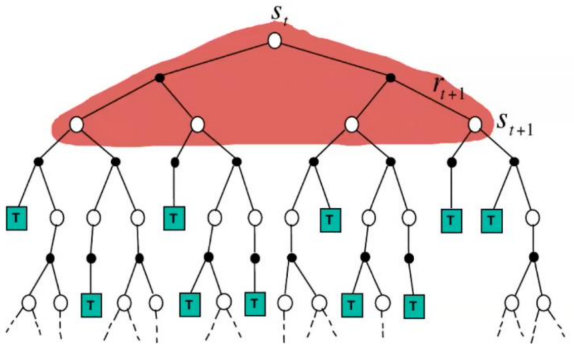


Figure 3.2: Dynamic Programming DP trajectory [71]. Starting point is state s_t, white circles are states, black dots are actions and green squares represent termination. r_{t+1} the immediate reward obtained when going from state s_t to state s_{t+1}.

seen with Monte Carlo in Section 3.1.3.2 and Temporal Difference in Section 3.1.3.3, that have not any knowledge of the model. DP performs *bootstrap*, since it uses only the values of the state that is reached after one step.

3.1.2.1 Policy Iteration

Policy Iteration is an algorithm that alternates two phases, *policy evaluation* and *policy improvement*, see Figure 3.3. The algorithm 1 begins with an arbitrary policy, e.g. random. The goal of *policy evaluation* phase is to retrieve the value function of the current policy V^π. This phase can be carried out into two different procedures. The first procedure is with a closed form solution, see Equation (3.4), that is computationally heavy. The second procedure is *iterative policy evaluation*, that implies the iterative application of the Bellman Expectation Operator, see Equation (3.10). The difference between the two approaches is substantially that the former returns an exact value function, even if it is computationally huge, while the latter retrieves only an approximation. With *iterative policy evaluation* we do not need to wait to reach convergence. An accurate approximation is produced also by *modified policy iteration* [60]. On the other side, *policy improvement* has the goal of building a new policy taking in each state the specific action that maximizes the state-value function, i.e. greedy policy improvement, $\pi^{t+1}(s) = \arg\max_{a \in \mathcal{A}} Q^{\pi^{(t)}}(s, a), \forall s \in \mathcal{S}$.

All the derived policies are deterministic by definition and the new policy is ensured to perform better than the previous one. This is guaranteed by the *Policy Improvement Theorem*, stating that, given two deterministic policies π and π', such that $Q^\pi(s, \pi'(s)) \geq V^\pi(s), \forall s \in \mathcal{S}$, then the policy π' must be

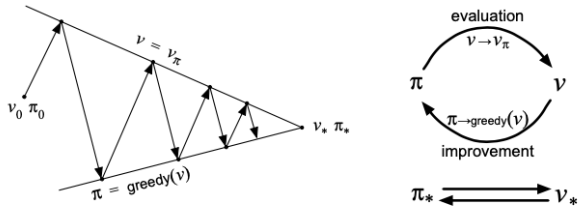


Figure 3.3: Policy Iteration Structure [75]. Policy Evaluation focuses on the estimation of V^π (e.g. Iterative Policy Evaluation), Policy Improvement generates $\pi' \geq \pi$ (e.g. Greedy Policy Improvement). v_0 denotes the initial value of initial policy π_0, then the estimation is updated till the optimal policy π_*, with value v_* is reached.

as good as, or even better, than π, i.e. $V^{\pi'}(s) \geq V^\pi(s), \forall s \in \mathcal{S}$ (see Section A.2).

Algorithm 1 Policy Iteration algorithm

Input: $\mathcal{S}, \mathcal{A}, P, R, T$
Output: π^T

1: *Initialize policy* $\pi^{(0)}$ *arbitrarily*
2: **for** $t = 0$ *to* $T - 1$ **do** ▷ T is maximum number of iterations
3: *Evaluate policy* $\pi^{(t)}$ *and get* $V^{\pi^{(t)}}, \forall s \in \mathcal{S}$
4: *Perform policy improvement*
$$\pi^{(t+1)}(s) = \arg\max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V^{\pi^{(t)}}(s') \right] \right\}, \forall s \in \mathcal{S}$$
5: **end for**
6: **return** *optimal policy* $\pi^{(T)}$

Fixing the maximum number of iterations to T, the contraction property holds [59]:

$$\|V^{\pi^{(T)}} - V^*\|_\infty \leq \gamma \|V^{\pi^{(T-1)}} - V^*\|_\infty \leq \gamma^T \|V^{\pi^{(0)}} - V^*\|_\infty. \tag{3.19}$$

The sequence of policies is ensured to converge in finite number of steps in case of finite MDPs [59] to the optimal policy.

3.1.2.2 Value Iteration

Value Iteration is an approach defined to solve the control problem of finding the optimal policy π^*. The solution is obtained through the iterative application of the Bellman optimality backup as illustrated in Equation (3.20)

$$V_1 \to V_2 \to \cdots \to V^*. \tag{3.20}$$

In Algorithm 2, Bellman optimality Equations (3.8), (3.9) can be exploited to find the optimal policy through DP. There is not any explicit policy as in Policy Iteration, see Section 3.1.2.1, and therefore intermediate value functions may not correspond to any feasible policy. Value Iteration computes optimal state-value function without representing intermediate policies. The approach is based on the iterative application of the *Bellman Optimality Operator*, see Equation (3.15). At the end, optimal policy is retrieved since the greedy policy derived from the obtained optimal state-value function.

Algorithm 2 Value Iteration algorithm

Input: $\mathcal{S}, \mathcal{A}, P, R, T$

Output: π^T

1: *Initialize policy* $\pi^{(0)}$ *arbitrarily*

2: **for** $t = 0$ *to* $T - 1$ **do** ▷ T is maximum number of iterations

3: *Apply Bellman Optimality Operator*

$$V^{(t+1)}(s) = T^* V^{(t)}(s), \forall s \in \mathcal{S}$$

4: **end for**

5: *Compute the optimal policy*

$$\pi^{(T)} = \arg\max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V^{(T)}(s') \right] \right\}, \forall s \in \mathcal{S}$$

6: **return** *optimal policy* $\pi^{(T)}$

Given two subsequent approximations of the optimal state-value function, the error can be bounded with respect to the optimal state-value function:

$$\|V^{(t+1)} - V^{(t)}\|_\infty < \epsilon \implies \|V^{(t+1)} - V^*\|_\infty < \frac{2\epsilon\gamma}{1-\gamma}. \tag{3.21}$$

Value Iteration is focused only on the state-value function, while Policy Iteration explicitly represents the obtained policy. Both algorithms are polynomial for MDPs characterized by a fixed discount factor [52]. Considering a single iteration only, Policy Iteration is more computationally heavy with respect to Value Iteration since it requires both evaluating the policy and performing the greedy improvement. On the other hand, Policy Iteration tends to converge in a smaller number of iterations with respect to Value Iteration.

3.1.3 Reinforcement Learning

3.1.3.1 RL Taxonomy

RL problems are divided into two categories: *Prediction* and *Control*. *Prediction* estimates the value function V^π of a given policy π, having as input

the experience generated by policy π, or another policy π' in a given MDP. *Control* problems, that are the ones we face in this thesis, are related to the optimization of the value function of a MDP. These problems are not related only to the estimation of the value of a policy, but also in deriving how to reach the optimal policy π^*.

An RL problem can exploit a *model*, that implies to learn a model of the environment by performing actions and observing results that include the next state and the immediate reward. Therefore, the model predicts outcomes of actions. Real-world examples can be the rules of games, in scenarios as chess and go, or simulators built according to laws of physics.

In the Table 3.1 the dimensions that characterize RL taxonomy are defined.

On-Policy	**Off-Policy**
Learn the policy used in the interaction with the environment	Learn a policy that is different from the policy used to collect data
Model-Free	**Model-Based**
Learn optimal policy without model	Learn model and use it to compute optimal solution
Online	**Offline**
Update estimates during learning, as soon as the agent gets new data	Data are picked up in an asynchronous way with respect to the learning phase, so data are collected online, and policy is subsequently estimated in an offline phase, with the collected data
Tabular	**Function Approximation**
Store value for each state and action	Learn parametric value function
Value-Based Methods	**Policy-Based Methods**
See Section 3.1.3.4	See Section 3.1.4

Table 3.1: Reinforcement Learning Taxonomy

3.1.3.2 Monte Carlo Methods

Monte Carlo (MC) for policy evaluation uses empirical mean return instead of expected return. It can be applied in two fashions: first visit, that averages returns only for the first time s is visited (unbiased estimator), or every visit, that averages returns for every time s is visited (biased but consistent estimator). With few samples, every-visit is better because it reduces variance even if bias increases. With many samples, first-visit is better be-

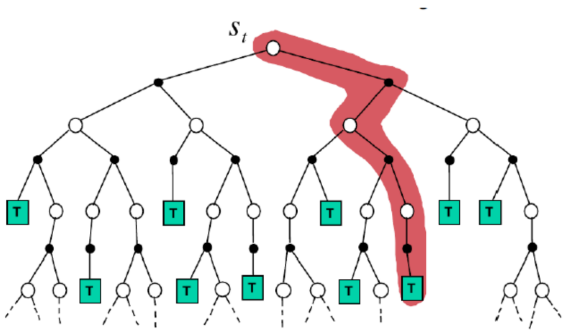


Figure 3.4: Monte Carlo MC trajectory [71]. Starting point is state s_t, white circles are states, black dots are actions and green squares represent termination.

cause it reduces bias and there is not any need to increase samples applying every-visit. MC updates $V(s)$ incrementally after episode $s_1, a_1, r_2, ..., s_T$. For each state s_t with return v_t:

$$\begin{aligned} N^{(t+1)}(s_t) &= N^{(t)}(s_t) + 1, \\ V^{(t+1)}(s_t) &= V^{(t)}(s_t) + \tfrac{1}{N^{(t)}(s_t)}\left(v_t - V^{(t)}(s_t)\right) \end{aligned} \tag{3.22}$$

$$\text{with } v_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots.$$

This approach learns only from complete sequences and works exclusively for episodic (terminating) environments. MC has good convergence properties. It works well with function approximation, is not very sensitive to initial values and is very simple to understand and use. In Figure 3.4 a possible trajectory considered by Monte Carlo is shown in red. Starting point is state s_t, white circles are the states, black dots are the actions and green squares represent the termination. MC chooses only one branch at a time, and goes in depth, since it simply takes one of the trajectories π, and when it reaches a terminal state, it takes the sum of rewards and updates the single state, using return v_t. The update rule in Equation (3.22) is used, with learning rate $\alpha = \frac{1}{N^{(t)}(s_t)}$.

Monte Carlo MC for control can be applied in two ways, *on-policy* and *off-policy*. *On-policy* Monte Carlo can be analogous to the generalized *Policy Iteration* with Monte Carlo evaluation of the policy. *Policy Evaluation* is performed through Monte Carlo update of the action-value function Q:

$$\begin{aligned} N^{(t+1)}(s_t, a_t) &= N^{(t)}(s_t, a_t) + 1, \\ Q^{(t+1)}(s_t, a_t) &= Q^{(t)}(s_t, a_t) + \tfrac{1}{N^{(t)}(s_t, a_t)}\left(v_t - Q^{(t)}(s_t, a_t)\right), \end{aligned} \tag{3.23}$$

$$\text{with } v_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots.$$

Policy Improvement is performed through the greedy policy improvement defined by the following system of equations, depending on the action-value function:

$$\pi(s,a) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & if\ a^* = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ \frac{\epsilon}{m} & otherwise \end{cases} \tag{3.24}$$

From *Policy Improvement Theorem*, see Section A.2, $V^{\pi'}(s) \geq V^{\pi}(s)$, with π' being a deterministic policy whose action-value function in state s is greater or equal with respect to the state-value function of s, given policy π. Policy Iteration converges to the optimal policy, see Section 3.1.2.1. GLIE Monte Carlo control (i.e. MC when the GLIE, see Section A.1.1, assumption holds) converges to the optimal action-value function [75].

The alternative approach is *off-policy* Monte Carlo for control, that applies the concept of importance sampling in reinforcement learning. A fundamental concept in the field of off-policy methods is the one of *Importance Sampling*. Importance sampling means to estimate the expectation of a different distribution with respect to the distribution used to draw samples. The algorithm uses returns generated from policy $\bar{\pi}$ to evaluate policy π. Considering generic distributions, importance sampling concept is modelled as:

$$\mathbb{E}_{x \sim p}\left[f(x)\right] = \mathbb{E}_{x \sim q}\left[\frac{p(x)}{q(x)} f(x)\right], \tag{3.25}$$

where $\frac{p(x)}{q(x)}$ is called *importance weight*, since it gives great importance to common samples in p, and rare in q, very small importance to samples that are not relevant for p. If p and q are very different, variance is large. If p and q are similar, then variance is small and importance sampling is efficient. The core idea of *off-policy* Monte Carlo is to weight return v_t according to similarity between policies. It multiplies importance sampling corrections along the whole episode, as in the following equation:

$$v_t^{\mu} = \frac{\pi\left(a_t|s_t\right)\pi\left(a_{t+1}|s_{t+1}\right)}{\bar{\pi}\left(a_t|s_t\right)\bar{\pi}\left(a_{t+1}|s_{t+1}\right)} \ldots \frac{\pi\left(a_T|s_T\right)}{\bar{\pi}\left(a_T|s_T\right)} v_t. \tag{3.26}$$

Then, update value towards corrected return $Q^{(t+1)}\left(s_t,a_t\right) \leftarrow Q^{(t)}\left(s_t,a_t\right) + \alpha\left(v_t - Q^{(t)}\left(s_t,a_t\right)\right)$. This method can be applied only if the two policies are not so different between them. *Off-policy* MC is derived from expected return:

$$Q^{\pi}\left(s,a\right) = \mathbb{E}_{\bar{\pi}}\left[\prod_{t=1}^{T} \frac{\pi\left(a_t|s_t\right)}{\bar{\pi}\left(a_t|s_t\right)} v_t | s_t = s, a_t = a\right]. \tag{3.27}$$

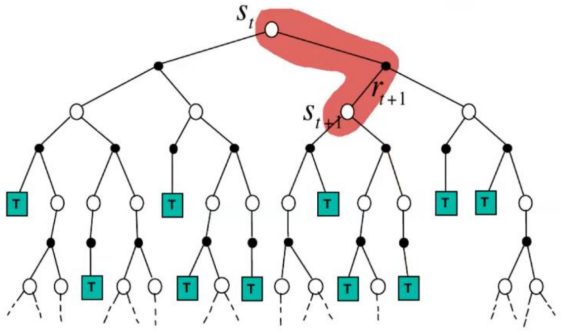


Figure 3.5: Temporal Difference TD trajectory [71]. Starting point is state s_t, white circles are the states, black dots are the actions and green squares represent the termination. r_{t+1} the immediate reward obtained when going from state s_t to state s_{t+1}.

3.1.3.3 Temporal Difference Methods

Temporal Difference (TD) method learns directly from episodes of experience. It can be seen as the RL version of the *Bellman Expectation equations*. The difference is that *Bellman Expectation equations* require knowledge of the transition model. TD learns from interactions and incomplete episodes through the bootstrapping technique. It is a model-free technique. The goal is to learn V^{π} online from experience under policy π. The update rule of the value function is defined as:

$$V^{(t+1)}(s_t) = V^{(t)}(s_t) + \alpha\left(r_{t+1} + \gamma V^{(t)}(s_{t+1}) - V^{(t)}(s_t)\right). \tag{3.28}$$

In practice, the real value v_t is substituted by the estimation $r_{t+1}+\gamma V(s_{t+1})$ with respect to Monte Carlo update, in Equations (3.22) (3.23). So, the update is made on the basis of another estimation. The process has a bias, but very small variance. $r_{t+1} + \gamma V(s_{t+1})$ is called the *TD target*, while $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is called the *TD error*. *TD target* represents the target towards which the algorithm wants to tend, *TD error* represents the difference between the new value to which TD tends and the previous estimate. In Figure 3.5, the possible trajectory considered by Temporal Difference is shown in red. Starting point is state s_t, white circles are the states, black dots are the actions and green squares represent the termination. Update of the value function is made after a single step, considering the specific reward r_{t+1}, that is collected when reaching the next state s_{t+1}. TD updates the current starting state s_t value according to the data collected in the single step, i.e. immediate return and expected value of the next state. The update rule in Equation (3.28) is used. Both Temporal Difference and Monte Carlo do sampling, since they pick only one action. The

difference is that TD stops after one iteration, while MC reaches the whole tree depth. TD can learn before knowing the final outcome, since it learns online at every step, while MC must wait until the end of episode before return is known, see Section 3.1.3.2. TD has also the advantage of being able to learn from incomplete sequences, with respect to Monte Carlo. TD works in continuing (non-terminating) environments and is usually more efficient than MC. TD(0), i.e. basic Temporal Difference, converges to $V^{\pi}(s)$. It may have problems with function approximation and it is more sensitive to initial values.

3.1.3.4 Value-Based Methods

Value-Based Methods, as the name suggests, are based on the preliminary learning of the action-value function and subsequently extraction of a policy derived by the value function. The policy π^* is derived as the one that maximizes the action-value function, so $\pi^* = \arg\max_{\pi} Q(s, \pi(s))$. According to the specific setting, we can constantly choose the greedy policy, that always selects the action that maximizes the action-value function. This approach is locally reasonable, but it does not guarantee exploration. The agent can be unable to learn new values and therefore can miss the optimal policy if exploration is not sufficient. To this extent, common selection criteria are ϵ-greedy, that implies the selection of the best action a^* with probability $1 - \epsilon$ or a random action with probability ϵ. Another strategy is *Boltzmann* that uses the concept of *temperature* τ, selecting the action according to weights that determine their probabilities, proportional to $e^{\frac{Q(s,a)}{\tau}}$. The most important algorithms in this category are *SARSA* and *Q-learning*.

SARSA algorithm simply requires to apply the TD algorithm, see Section 3.1.3.3, in *Policy Evaluation*, while keeping the greedy *Policy Improvement*. Name of *SARSA* comes from the update of action-value function Q, State-Action-Reward-State-Action. As soon as one step is performed, the greedy-policy is applied. The policy always changes at every step. Convergence is reached starting from every action-value function, because Policy Improvement and TD are applied. *SARSA* converges to the optimal action-value function under the GLIE assumption (see Section A.1.1), that requires that the greedy factor tends to zero and that learning rate satisfies Robbins-Monro conditions [75]:

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \qquad \sum_{t=1}^{\infty} \alpha_t^2 < \infty. \tag{3.29}$$

In on-policy *SARSA*, action-value function is updated according to the fol-

lowing rule:

$$Q^{(t+1)}(s_t, a_t) = Q^{(t)}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q^{(t)}(s_{t+1}, a_{t+1}) - Q^{(t)}(s_t, a_t) \right). \tag{3.30}$$

Policy Improvement is performed through the greedy policy improvement defined by the following system of equations, depending on the action-value function:

$$\pi(s,a) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & if\ a^* = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ \frac{\epsilon}{m} & otherwise. \end{cases} \tag{3.31}$$

SARSA algorithm is applied also in off-policy context. As for *off-policy* Monte Carlo, also in this case importance sampling is exploited and the algorithm uses returns generated from policy $\bar{\pi}$ to evaluate policy π. The off-policy *SARSA* algorithm makes bootstrap in next step, applying importance weight. Importance sampling is defined as $\frac{\pi(a_{t+1}|s_{t+1})}{\bar{\pi}(a_{t+1}|s_{t+1})}$. It uses the TD target $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ generated from policy $\bar{\pi}$ to evaluate policy π. The update rule of action-value function is hence defined as:

$$w = \frac{\pi(a_{t+1}|s_{t+1})}{\bar{\pi}(a_{t+1}|s_{t+1})},$$

$$Q^{(t+1)}(s_t, a_t) = Q^{(t)}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma w Q^{(t)}(s_{t+1}, a_{t+1}) - Q^{(t)}(s_t, a_t) \right), \tag{3.32}$$

with w as the importance weight for next state action-value function. This method can be applied only if the two policies are not so different. There is much lower variance than Monte Carlo importance sampling.

Q-learning is an off-policy RL algorithm for control, so it has the goal to estimate the optimal policy. It is the most popular and used algorithm. It is the RL equivalent of Value Iteration, while SARSA is the RL equivalent of Policy Iteration. The idea of *Q-learning* is to directly try to estimate optimal action-value function Q^*. It works with the following update rule of $Q-$value:

$$Q^{(t+1)}(s,a) = Q^{(t)}(s,a) + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q^{(t)}(s', a') - Q^{(t)}(s,a) \right). \tag{3.33}$$

Q-learning can learn the optimal policy from every policy (even random) that does not put zero probability to an action, as in case zero probability is put, then that action cannot be explored in the future. This is a great difference with *SARSA*, that can learn the optimal policy only with a similar policy. *Q-learning* converges even if exploration does not tend to zero, if the

learning rate respects Robbins-Monro conditions. *SARSA* converges only if the exploration term ϵ goes to zero (i.e. GLIE assumption, see Section A.1.1) and if learning rate respects Robbins-Monro conditions.

All the methods seen so far belong to the class of *action-based methods*, that learn values of actions and then select actions according to the estimated action values.

3.1.4 Policy Gradient Methods

These types of algorithms differ from the previously seen *action-based methods*, since they learn a *parametrized policy*, that is able to select actions without need of a value function. In fact, a value function is not required for action selection, even if it can still be useful to *learn* policy parameter. The policy formulation can be written as $\pi(a|s, \boldsymbol{\theta}) = Pr\{A_t = a, S_t = s, \boldsymbol{\theta}\}$, where $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ is the policy parameter vector, i.e. the probability that when environment is in state s at time t with parameter $\boldsymbol{\theta}$, action a is taken. The strategy of these methods implies to look at some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter itself to learn the policy parameter. The goal is to maximize performance, so the update of policy parameter $\boldsymbol{\theta}$ is made with approximate gradient *ascent* in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)}, \tag{3.34}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$ represents a stochastic estimation of the gradient. Its expectation approximates the performance measure J gradient with respect to its argument $\boldsymbol{\theta}_t$. In this case, gradient ascent is exploited, since the goal is the maximization of a function. If the objective is defined as a minimization (e.g. of a loss $\mathcal{L}$), then the technique takes the name of gradient descent and the minus sign in the weights update is applied, $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha\nabla\mathcal{L}(\boldsymbol{\theta}_t)$. All methods that can be schematized in this way are called *policy gradient methods*, independently of the learning of an approximate value function. There is a class of methods, that learns both the policy and the value functions, that is called *actor-critic methods*, see Section 3.1.4.1. Since $\pi(a|s, \boldsymbol{\theta})$ is differentiable with respect to its parameters, i.e. $\nabla\pi(a|s, \boldsymbol{\theta})$ exists and is finite $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s), \forall \boldsymbol{\theta} \in \mathbb{R}^{d'}$, the policy can be parametrized indifferently. The purpose in this approach is to optimize, i.e. maximize, the following performance measure: $J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0)$, where the true value function for $\pi_{\boldsymbol{\theta}}$ is $v_{\pi_{\boldsymbol{\theta}}}$. *Policy Gradient Theorem* allows for the estimation of the performance gradient $\nabla J(\boldsymbol{\theta})$ with respect to the policy parameter $\boldsymbol{\theta}$. Changes of policy on the state distribution have consequences that affect

the gradient. Policy gradient methods are found on the *Policy Gradient Theorem* that states:

$$\nabla J(\boldsymbol{\theta}) = \sum_s \mu(s) \sum_a q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla \pi_{\boldsymbol{\theta}}(a|s,\boldsymbol{\theta}), \tag{3.35}$$

in which μ represents the on-policy distribution over $\pi_{\boldsymbol{\theta}}$, whose definition is $\mu(s) = \lim_{t\to\infty} Pr\{S_t = s|A_{0:t} \sim \pi_{\boldsymbol{\theta}}\}$. $\pi_{\boldsymbol{\theta}}$ is the specific policy based on vector $\boldsymbol{\theta}$. Finally, the gradient represents a column vector of partial derivatives with respect to the parameter $\boldsymbol{\theta}$'s components. The first *policy-gradient methods* historically developed are *stochastic policy-gradient methods* [75], characterized by optimization over a stochastic policy $\pi_{\boldsymbol{\theta}}(a|s,\boldsymbol{\theta})$. The first example of *stochastic policy-gradient methods* is REward Increment Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility (REINFORCE) algorithm [88], that is based on a Monte Carlo estimation of the action value function from Equation (3.35), with the derivation in Equation (3.36). In these equations, S_t and A_t substitute the possible values s and a and represent the replacement of a sum over the possible values of the random variable by an expectation under policy π, then sampling the expectation at time t. The derivation exploits the introduction of the term $\pi(a|S_t,\boldsymbol{\theta})$ at both numerator and denominator, since we need to introduce a weight by $\pi(a|S_t,\boldsymbol{\theta})$, necessary for expectation under π. In the end, it exploits the equality involving the return G_t, $\mathbb{E}_\pi\left[G_t|S_t,A_t)\right] = q_\pi(S_t,A_t)$.

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\
&= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right] \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[q_\pi(S_t,A_t) \frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} \right].
\end{aligned} \tag{3.36}$$

The final expression in brackets represent exactly the quantity that may be sampled at each time step. Its expectation is equal to the gradient, so the definition of the update of weights for REINFORCE can be defined as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}, \tag{3.37}$$

where the stochastic estimation $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ in the update rule expression is substituted by the content of the last term in brackets in Equation (3.36). This specific equation refers to the undiscounted version of the update rule, i.e. γ is missing. The obtained result entails that each increment is proportional to the product of return G_t and a vector. This vector is the ratio between the probability of taking the action actually taken and the probability of taking that action. The direction is the one that maximizes the probability of repeating action A_t, when in state S_t, in the future. The algorithm pseudo code of REINFORCE 4 is detailed in Section 3.3.1.3, in which a comparison with respect to Policy Gradients with Parameter based Exploration (PGPE), the adopted algorithm in this thesis, is presented.

Recently, also *deterministic policy-gradient methods* have been developed. The main examples are Deterministic Policy Gradient (DPG) and Deep Deterministic Policy Gradient (DDPG). DPG exploits a deterministic policy to determine an off-policy *actor-critic* algorithm that exploits linear functions [72]. Such derivation uses a differentiable function approximator for estimating action-value function. Subsequently, it adjusts policy parameters in the direction of the estimated action-value gradient. The peculiarity of DPG is that it integrates over the state space only, while in stochastic policy gradient methods integration is performed also over actions. It results in fewer samples in contexts with large action spaces.

DDPG solves the issue of the curse of dimensionality of DPG and other actor-critic methods, derived from the use of linear functions [48]. As this feature can represent an obstacle for scalability, the use of a model-free, off-policy, actor-critic algorithm is a good solution. DDPG learns policies through high-dimensional and continuous action spaces, using an actor and critic both approximated by a neural network. For action selection, a Gaussian noise is present in order to guarantee exploration. Critic network parameters are optimized via Adam optimization, while actor network parameters are updated with a proportional step to the critic sampled gradient, considering the action that is taken.

3.1.4.1 Actor-Critic

Actor-Critic techniques are a subclass of Policy Search, in which the term *actor* refers to the learnt policy (i.e. for performing action selection), and *critic* to the learnt value function, e.g. typically a state-value function (i.e. for estimating value function). Such methods represent a combination between policy-based, see Section 3.1.4, and value-based, see Section 3.1.3.4, methods as they combine benefits of both classes. After picking each action,

evaluation of the new state is performed by the critic, to decide whether the outcome of the chosen action was worse or better than what was expected. According to this, the actor adjusts the policy distribution towards the direction indicated by the critic in one step, with the following formula:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(R_{t+1} + \gamma \hat{v} \left(S_{t+1}, \mathbf{w} \right) \right) \frac{\nabla \pi(A_t | S_t, \boldsymbol{\theta}_t)}{\pi(A_t | S_t, \boldsymbol{\theta}_t)}, \tag{3.38}$$

where $\hat{v}\left(S_{t+1}, \mathbf{w}\right)$ is an estimate of the state value, with $\mathbf{w} \in \mathbb{R}^m$. One-step actor–critic methods replace the full return of REINFORCE, see Equation (3.37) with the one-step return. *Actor-Critic* performs bootstrapping solving the issue slowness that characterize Monte Carlo methods and REINFORCE, being episodic algorithms.

3.2 Regression Models

Supervised (inductive) learning is the largest, most mature, most widely used sub-field of machine learning. Supervised Learning has the goal to estimate the unknown model that maps the known inputs to known outputs. It has a training set as input $\mathcal{D} = \{\langle x, t \rangle\} \implies t = f(x)$, that is the set of samples on which the model is trained. The goal is to learn f, the model that maps the input x into the output t [7]. Supervised Learning problems are subdivided into sub-categories on the basis on the type of the target t. *Regression* is a specific category of Supervised Learning problems, that is characterized by a target t that is a continuous variable, e.g. price of a house on the market. Other Supervised Learning problems are *Classification*, i.e. target is a categorical, discrete variable, and *Probability Estimation*, i.e. target t is the probability of x. Figure 3.6 shows the schema of Supervised Learning, that learns the model, finding a function that maps the input x into the output t, and the other models of Machine Learning, i.e. Unsupervised Learning, that has input data but not supervision (no output data) and aims at understanding if the input data have some structure, and Reinforcement Learning, that learns how to behave by looking at experience.

3.2.1 Artificial Neural Networks

Artificial Neural Networks (ANN) are networks, i.e. directed graphs, of *neurons*, i.e. simple and small computational units, structured in layers as

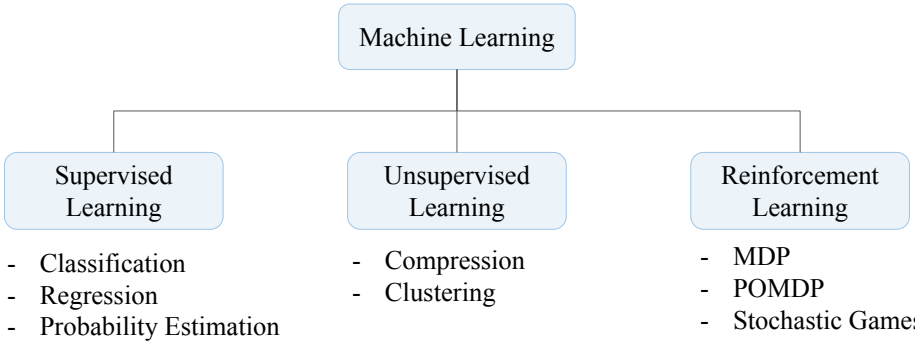


Figure 3.6: Machine Learning classification, with the problems belonging to each category. Supervised Learning, that learns the model, finding a function that maps the input into the output. Unsupervised Learning, that learns a new representation of input data, without supervision (i.e. looking for some structure in input data). Reinforcement Learning, that learns how to control, determining how to behave starting from experience.

shown in Figure 3.7. Prediction $y(\mathbf{x}, \mathbf{w})$ is computed as:

$$y(\mathbf{x}, \mathbf{w}) = h\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right), \tag{3.39}$$

where $h(\cdot)$ denotes the activation function. Sigmoid functions, e.g. logistic function and tanh, and ReLU (Rectified Linear Unit) are widely used activation functions. Identity is commonly used as activation function of the output neurons in case of regression. w_j represent the weights of the network, and $\phi(\mathbf{x})$ denote non linear basis functions of the input $\mathbf{x}$ [7]. There are different categories of NN. In this thesis, we adopt the class of *Feed Forward* NN, see Figure 3.7. As depicted in Figure 3.7, there are the input layer, that takes raw data, one or more hidden layer that extract high level features and an output layer that outputs the regression target. The neuron j is the elementary unit of a neural network. It takes a vector of inputs and applies a non linear function to the linear combination of the vector elements as shown in Equation (3.40):

$$z_j = h\left(\sum_{i \in in(j)} w_{ji} z_i\right), \tag{3.40}$$

where z_i is the output of the previous neuron and w_{ji} is network learnable parameter, i.e. weight. In these type of networks, neurons are placed on

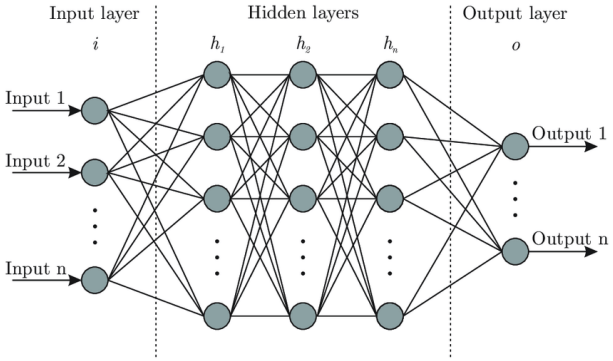


Figure 3.7: Artificial Neural Network structure [11]. The input layers is used for the initial data provided to the neural network, while the output layer produced the result for such inputs. There are also hidden layers, i.e. intermediate layers between input and output layers that are dedicated to computation.

directed acyclic graph, characterized by nodes without entries as input and neurons without outer edges as output.

Backpropagation is the algorithm used to train ANN. It is a gradient descent algorithm focused on the recursive computation of network's gradient over the chain rule. Given $\mathbf{z} : [\, z_j : j \in \text{output of neurons}]$, *activation function* of neuron j defined as $a_j = \sum_{i \in in(j)} w_{ji} z_i$, we can define a differentiable loss function $L(\mathbf{z})$, with gradient $\delta_j = \frac{\partial L}{\partial a_j}$. At this point, it is possible to compute gradient for output neurons. If supposing the *Identity activation function* for the neurons output:

$$\delta_j = \frac{\partial L}{\partial z_j}, \tag{3.41}$$

subsequently, perform *backpropagation* of the gradient across the neural network. To this extent, the following chain rule is used

$$\delta_j = \frac{\partial L}{\partial a_j} = \sum_{i \in out(j)} \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial a_j} = h'(a_j) \sum_{i \in out(j)} w_{ij} \delta_i. \tag{3.42}$$

Computing the value of δ_j for each neuron allows to compute the derivative with respect to the weights that can be learnt as $\frac{\partial L}{\partial w_{ji}} = z_i \delta_j$. Update of them is then performed though gradient methods.

3.3 Reinforcement Learning Algorithms

3.3.1 PGPE

Differently from classical Policy Gradient methods like REINFORCE, PGPE uses two policies: action-policy and hyper-policy. The former can be any function that maps state to a probability distribution over actions (or action directly if it is deterministic), while the latter is a parametric probability distribution over action-policy parameters. The advantage of such an approach is to speed up convergence, that is slower in case of policy estimation performed by policy gradient methods. By using a simple form of hyper-policy, like Gaussian distribution, it is possible to compute hyper-policy gradients in closed form resulting to be efficient. In PGPE a single parameter sample is effective for generating a complete action-state history, and there are lower variance estimates. PGPE faces the variance issue in classical policy gradient methods by using directly a probability distribution over parameters themselves, defined as:

$$p(a_t|s_t, \boldsymbol{\rho}) = \int_{\boldsymbol{\Theta}} p(\boldsymbol{\theta}|\boldsymbol{\rho}) \delta_{F_{\boldsymbol{\theta}}(s_t)}(a_t) d\boldsymbol{\theta}, \tag{3.43}$$

in which the hyperparameter ρ determines distribution over parameters θ, s_t is the state, and δ is Dirac. $F_\theta(s_t)$ is the deterministic action that the model chooses in state s_t. Variance estimates of gradient are reduced due to the determinism of actions. The goal, given hyperparameter ρ, is to maximize the expected reward:

$$J(\rho) = \int_{\Theta} \int_{H} p(h, \theta|\rho) r(h) dh d\theta, \tag{3.44}$$

in which h represents the history of agent, i.e. the sequence of state-action pairs (also called trajectories or roll-outs). $r(h)$ means that cumulative reward is associated with each history, by summing, every time step, over rewards $r(h) = \sum_{t=1}^{T} r_t$, with T as length of the sequence. At the beginning of each sequence, parameters are sampled from a separate Gaussian distribution, i.e. one for each parameter. Considering θ_i in $\boldsymbol{\theta}$, i.e. the single parameter in the parameters' vector; each one of them is characterized by an independent normal distribution with mean μ_i and standard deviation σ_i. Mean and variance of the distribution of each parameter are updated during training with the following update rules:

$$\delta\mu_i = \alpha(r - b)(\theta_i - \mu_i) \qquad \delta\sigma_i = \alpha(r - b)\frac{(\sigma_i - \mu_i)^2 - \sigma_i^2}{\sigma_i}, \tag{3.45}$$

in which step size of the gradient is defined as $\alpha_i = \alpha\sigma_i^2$, with α being a constant factor; b is a baseline and r is the reward, following [88]. Therefore, it speedups convergence, supplying issue of slow convergence typical of policy gradient methods that perform exploration over action spaces through probabilistic policies. Pseudo code of PGPE algorithm is shown in Algorithm 3, that sums up its main steps. There is very little noise in the gradient estimation, as reward for each sequence is dependent only on single samples. Besides PGPE [65], that directly applies the search over parameters space (in place of a search in policy space, considering classical policy gradient methods 3.1.4), there are many other algorithms that have been evaluated in the state-of-the-art, e.g. Simultaneous Perturbation Stochastic Approximation (SPSA) [73], Evolution Strategies (ES) [64] and the family of algorithms defined as REINFORCE algorithms [88].

Algorithm 3 PGPE algorithm

procedure PGPE(μ_0, σ_0) ▷ Initialize mean and standard deviation
  $\mu \leftarrow \mu_0$
  $\sigma \leftarrow \sigma_0$
  while TRUE **do**
    for n=1 to N **do** ▷ N is number of histories of PGPE
      draw $\theta^n \sim \mathcal{N}(\mu, I\sigma^2)$
      $r(h)$ *evaluation* ▷ history reward evaluation
    end for
    Compute $\delta\mu_i = \alpha(r - b)(\theta_i - \mu_i)$
    Compute $\delta\sigma_i = \alpha(r - b)\frac{(\sigma_i - \mu_i)^2 - \sigma_i^2}{\sigma_i}$
    Update μ
    Update σ
  end while
end procedure

A complete overview of the algorithms related to PGPE is presented in Figure 3.8. There are three main features that are considered for classifying the algorithms: following of the gradient method, control over exploration and perturbation in the parameter space. PGPE holds these three features simultaneously, while the other mainly used state-of-the-art-algorithm are characterized by only a binary combination of the three, e.g. classical REINFORCE follows the gradient, assuming control over exploration, while it doesn't perform perturbation over the parameter space. In Figure 3.9 we can see how PGPE outperforms with respect to the benchmark problem of Pole Balancing (i.e. task of balancing a pole upright in track center for a

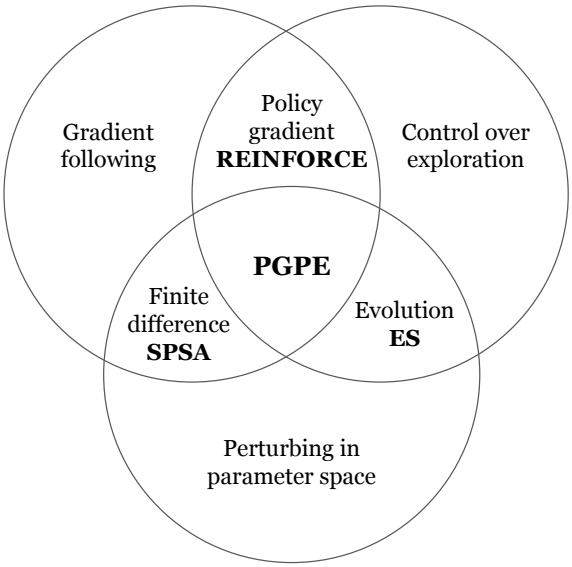


Figure 3.8: Policy Gradients with Parameter-based Exploration PGPE [65]. It is an algorithm that follows the gradient method, has control over exploration and perturbs the parameter space. These 3 features are not all simultaneously present in the other mainly used state-of-the-art-algorithm.

movable cart for the longest amount of time) [65] in relation to the other algorithms.

3.3.1.1 SPSA

SPSA differs from PGPE since it exploits uniform sampling in place of Gaussian sampling. As a consequence, it manages finite differences gradient in place of likelihood gradient of PGPE. Besides, variances of perturbations are kept fixed (while are trained in PGPE). Original update rule for SPSA is defined in Equation (3.46):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{r\left(\boldsymbol{\theta}_t + \nabla\boldsymbol{\theta}_t\right) r\left(\boldsymbol{\theta}_t - \nabla\boldsymbol{\theta}_t\right)}{2\epsilon}, \tag{3.46}$$

in which $r(\boldsymbol{\theta})$ is the evaluation function, ϵ is the step size, that decays as time t grows as usual, and α the learning rate. ∇ is drawn from a Bernoulli distribution, that is scaled by the step size, i.e. $\nabla = \epsilon_t \cdot rand[-1, 1]$. In SPSA the choice of initial parameters is critical for the algorithm's performances and convergence (i.e. an issue that PGPE aims to solve) [65, 73].

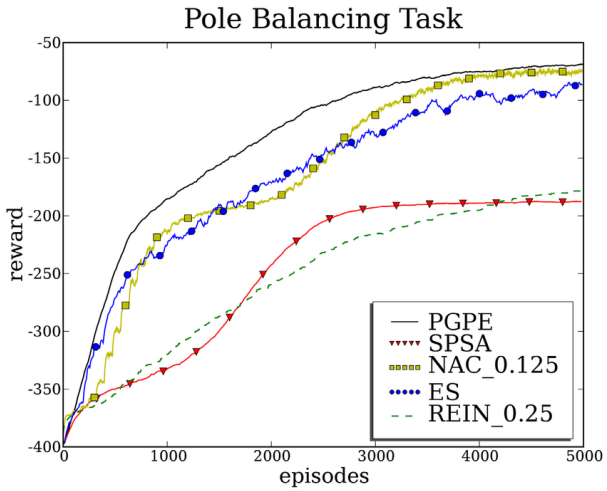


Figure 3.9: Pole Balancing Task [65, 63]: comparison PGPE, ES, SPSA, REINFORCE and NAC.

3.3.1.2 ES

ES main features are having both stochastic mutations and standard deviation updates. Deterministic ES versions have been developed [65, 64], but convergence still remains not comparable with respect to PGPE. If we apply the likelihood gradient to ES [33], the update rule Equation (3.47) is derived.

$$\nabla\theta_i = \alpha \sum_{m=1}^{M} (r_m - b)(y_{m,i} - \theta_i), \tag{3.47}$$

in which i denotes the parameter i of $\boldsymbol{\theta}$, M is the overall number of samples, $y_{m,i}$ denotes the parameter i of sample m [65].

3.3.1.3 REINFORCE

Considering REINFORCE, whose formulas have been derived in Equations (3.36), (3.37) in Section 3.1.4, the main difference with PGPE is that it requires one sample every time step (while PGPE requires one sample per history). In REINFORCE, decreasing the frequency of perturbations binds exploration range as well as reducing gradient variance at the same time. Once a saturation point is reached, there is no further improvement. This trade-off is not proper of PGPE, in which a single perturbation of parameters can bring a significant behavioural change [65, 88]. REINFORCE algorithm pseudo code is shown in 4. The pseudo code refers to Monte-Carlo Policy-Gradient Control (episodic) for optimal policy π^*. This class of algorithms

finds an unbiased estimate of the gradient, without the need of learning a value function; the main feature is using a probabilistic policy, in place of the deterministic policy applied by PGPE.

Algorithm 4 REINFORCE algorithm

Input: *Differentiable policy parametrization* $\pi(a|s, \boldsymbol{\theta})$

$\boldsymbol{\theta} \leftarrow \mathbf{0}$ ▷ *Initialize* $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ *, e.g. to* $\mathbf{0}$
while TRUE **do** ▷ *For each episode*
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, *following* $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  for t=0 to $T-1$ **do** ▷ *Loop for each step of the episode*
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
  end for
end while

Step size $\alpha > 0$ is a parameter of the algorithm, γ is, as usual, the discount factor. In the algorithm pseudo code, the gradient is updated with $\ln \pi(A_t|S_t, \boldsymbol{\theta})$, in place of $\frac{\nabla \pi(A_t|,S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}$, of Equation (3.37). This compact expression that is substituted in the pseudo code follows from the identity $\nabla \ln x = \frac{\nabla x}{x}$. The update rule is also generalized through the presence of the discount factor, with respect to (3.37).

Recently, other algorithms and approaches are a central topic of active research in this context, e.g. in [95, 31, 49].

3.4 Planning Algorithms

Planning task can be performed both with methods that require a model, i.e. *model-based* (see Table 3.1), and that do not require a model, i.e. *model-free* (see Table 3.1). *Model-free* principally rely on learning, then on planning. The main feature of planning methods, both *model-based* and *model-free*, is that they focus on computation of value-function, since they look at future events, compute backed-up values and use it for approximate value-function as an update target. There are two main strategies of performing planning. The former consists in using planning to progressively enhance a policy or value-function on the basis of simulated experience acquired from a model (e.g. a sample or a distribution model). This approach can be found in Dynamic Programming [75]. The latter consists in starting a new plan and completing it *after* each new state S_t has been encountered. In this case the computed plan has the only single action A_t as output. At next step, the plan

starts from state S_{t+1}, reached from state S_t taking action A_t, to compute next action A_{t+1}, and so on. The advantage of using planning in this way is that it is not necessary to stop one-step-ahead, since it is possible to look much deeper. This category of methods is called *decision-time planning methods*. MCTS belongs to this category [21] and is the algorithm used for planning in this thesis.

3.4.1 Monte Carlo Tree Search

MCTS is a planning algorithm based on stochastic simulations that applies best-first search. The basis of the algorithm is considering a model of the environment in which the agent randomly selects actions (and so its opponents) in a finite-length time horizon. The overall resulting strategy, as the algorithm output, is obtained with the simulation of a large number of random states. It is, asymptotically, an optimal technique. The goal of MCTS is to build a sparse tree, trying to sample mainly interesting parts of the tree. In practice, the algorithm is very effective and the most used technique in planning today. Its main steps are:

- Selection;
- Expansion;
- Simulation;
- Backpropagation;

MCTS builds a tree of all possible future states that are expanded, according to the steps illustrated in Figure 3.10. MCTS pseudo code is described in 5.

Selection is the step that balances exploration and exploitation. In fact, the goal is to choose the action that has performed better so far. On the other side, evaluation can suffer from uncertainty and therefore exploration of less promising actions must be performed to guarantee enough exploration. A possible strategy to select a node is to use an *Upper Confidence Bound* [12], so UCB1, that creates a bound with the following expression that balances between exploration and exploitation:

$$UCB1(n) = \frac{U(n)}{N(n)} + C\sqrt{\frac{\log\left(N\left(Parent(n)\right)\right)}{N(n)}}, \tag{3.48}$$

where n is the current considered node for computing the bound. $U(n)$ represents the number of positive results of the simulations that have traversed node n, while $N(n)$ is the number of times that node n has been

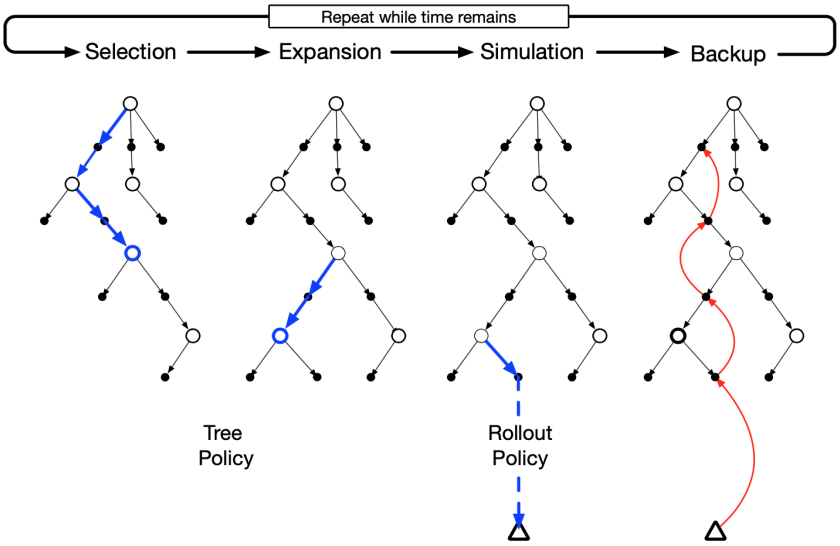


Figure 3.10: Monte Carlo Tree Search algorithm pipeline [75]. Selection is the first step that balances exploration and exploitation. Expansion is the algorithm tree building main step, that adds new nodes to the tree. Simulation is the step in which actions are selected across all the duration of the game. Backpropagation is the step in which the end of simulation is reached and update of stored values is performed.

explored, i.e. was present in the simulation. $N\left(Parent(n)\right)$ is a measure of the exploration of the parent node of n. C is a term that allows to weight between exploration and exploitation. We have exploration when $C \to \infty$ and exploitation when $C \to 0$. The other terms represent:

- $\frac{U(n)}{N(n)}$: exploitation, since this term is high if the node is promising.

- $\sqrt{\frac{\log(N(Parent(n)))}{N(n)}}$: exploration, since this term represent the fact that node n is a child not much explored of its parent.

Algorithm 5 Monte Carlo Tree Search

Input: s_0, H
Output: $best_{move}(s_0)$

```
1: Root ← s_0
2: for t=0 to H do                                  ▷ H is MCTS horizon
3:     n, s ← Root, Root.state
4:     if n is not a leaf then
5:         n ← n.child                               ▷ Selection
6:         Store move a
7:     end if
8:     expand(n.state)                               ▷ Expansion
9:     n ← n.child
10:    while s is not terminal do
11:        s ← simulation(s)                         ▷ Simulation
12:    end while
13:    res = evaluate(s)
14:    while n has a parent do
15:        update(n, res)                            ▷ Propagation
16:        n = n.parent
17:    end while
18: end for
19: return best_move(s_0)
```

Expansion is the algorithm tree building main step. For each simulation, the tree is added the first new node found in the selection that was not yet present.

Simulation is the step in which actions are selected across all the duration of the game. In this step, actions are chosen according to a *playout policy*, that is the default policy, i.e. a reasonable good policy to use in simulation that can also be learned from self-play. Selection probabilities of actions should be appropriately weighted to correctly weight the possible choices and guarantee a good play quality. Using a heuristic is often a good strategy to make MCTS stronger and give larger weights to more promising actions. Putting uniform probability to all valid actions could lead to a sub-optimal solution and not good performances.

Backpropagation is the final step in which the end of the simulated game is reached. In this step, each tree node that has been considered in the expansion is updated accordingly. Visit counts are increased and the node feature (win/loss ratio) is updated according to the simulation output.

Chapter 4

Related Works

Since this thesis aims at applying the concept of *Teacher-Student* RL in the field of autonomous driving, in this Chapter similar works, already present in the state-of-the-art that deal with the same topics are presented. In this chapter, there is also an overview on Transfer Learning, in Section 4.1, that refers a learning process between different tasks, i.e. environment changes. *Teacher-Student* Reinforcement Learning, with the related works is described in Section 4.2, and is different with to respect to pure Transfer Learning (TL) since the learning process takes place within the same environment, between an expert and a beginner. The chapter proceeds with an overview on the approaches of the state-of-the-art in the field of autonomous driving in Section 4.3. This section will detail approaches in the driving learning style and methods to perform planning activities on the track.

4.1 Transfer Learning

Sample inefficiency and slow training times are some of the weaknesses of modern RL methodologies that result in narrow scalability and applicability in many practical situations. To this extent, research area of TL is focused on the development of systems to perform knowledge transfer from a source to a target. In particular, TL focuses on transferring knowledge between different tasks. This approach has many advantages and one possible area of interest is the scenario of Multi Agent Reinforcement Learning (MARL), i.e. a domain that presents dimensionality curse features. Some proposals in this field can be found in [10, 22, 23]. In [10], Boutsioukis et al. concentrate on the evaluation of the applicability of the *transfer* to multi-agent RL. Furthermore, the authors examine the *transfer* from a single agent sys-

tem to a multi-agent system and from a multi-agent system to another, in offline mode. They propose a framework in which knowledge derived from a number of agents or information related to a single agent is generalized and exploited for mapping from source to target. They demonstrate that transfer is effective in multi-agent systems, but that a multi-agent source task has not advantage with respect to a single-agent source task. In [22], Da Silva et al. focus on the issue of considering both cooperation and competition in the multi-agent transfer process. The authors propose a framework in which the RL agent has to perform transfer learning from two perspectives. The former consists in the knowledge extraction directly from source tasks, the latter is the knowledge transfer coming from experienced RL agents. In [23], Da Silva et al. develop the MARL transfer problem, considering the complex problem of getting advice from n agents. In this case, having multiple teachers, the *student* must decide how to weight the advice from different sources. The authors propose in [23] a multi-agent advising framework in which advice selection is performed through majority voting.

An interesting topic of analysis, already extensively studied in many works, e.g. [45, 17, 78], is related to the determination of similarity between tasks. In [45], Lazaric proposes a complete survey with the definition of a TL taxonomy. The author focuses on the improving in the learning performance of the target task with the use of transferred knowledge, e.g. final convergence level increase, or number of needed samples for an optimal performance reduced. An important topic in the filed of transferring knowledge between different tasks is related to the similarity measure between them. In [17], Carroll et al. propose an experimental analysis to evaluate the improvement in learning on the basis of specific information, e.g. $Q-$values, reward structure and policy overlapping. The authors claim for the need of similarity to allow for robust transfer and propose different metrics. They also test the proposed metrics, evaluating their goodness. In [78], Taylor et al. focus on TL algorithms to initialize a learner in a target task. The authors pursue to transfer only a learned dynamics model, not the value-function, and provide a complete survey on TL and multi-task RL, mainly considering single-agent, fully-observable settings. In Figure 4.1 the comparison between traditional Machine Learning (ML) and TL. An open issue is the problem of performing knowledge transfer between very different agents [26, 27]. In [26], Ezra et al. propose also a specific definition for *reward* shaping [25] a supervised, iterative, process to support learning. This procedure allows to accelerate the learning process of the agent, since it is provided at every transition with localized advice. A powerful idea is the one of transferring knowledge across related tasks, even if different, in order

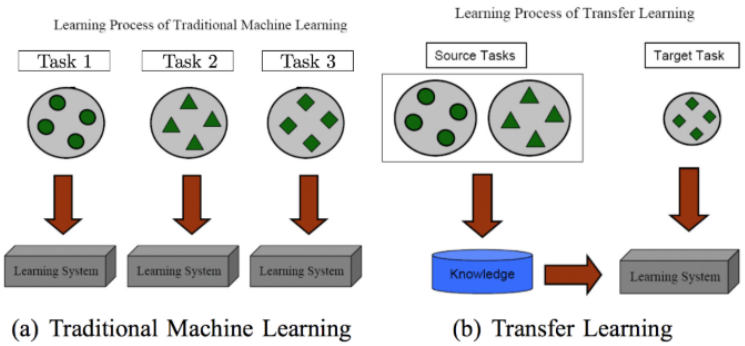


Figure 4.1: Traditional Machine Learning VS Transfer Learning [90]. In Transfer Learning, the system performs knowledge transfer from a source task to a target task. Many topics of research arise in this context, e.g. multiple-sources for the external knowledge, similarity study between source and target task.

to improve the performance of ML algorithms, that finds its root in cognitive science and psychology research area [27]. In [27], Fachantidis et al. focus on *Teacher-Student* RL and claim that the advisor value function may not be the optimal guide for the selection of the suggested action, since the *advice* will not follow the policy of the *Teacher*. Therefore, the validity of the horizon that is used for the policy computation may not be preserved. The authors demonstrate that the performance in solving the task alone is not sufficient to choose best advice action and propose a method to learn the way in which provide advice, to make learning as faster as possible. As with human relations, it was shown that humans learn better and faster if referring to prior knowledge on similar tasks. The goal in TL is to design algorithms that analyze prior collected knowledge from sources (e.g. demonstrations, solutions, samples) and transfer it towards the target learning process, influencing performances and moving towards good behaviours. TL has shown remarkable effects in improving performances already in case of Supervised Learning (SL), considering many applications, e.g. recommender systems, text classification, decision making, medical applications and general game playing [27]. Recently, also the application of TL to RL is a research area of growing interest. The power of TL in this field consists in solving the issues already mentioned, i.e. slow training and sample inefficiency, but also to the possibility of building prior knowledge that can significantly speedup the learning procedure of a policy (i.e. the target), starting from a related task (i.e. source), without the need of restarting from scratch. The learning process of the target is hence biased, and advantages are both a significant

reduction in the number of required samples for training and a remarkable improvement in the accuracy of the learnt policy.

4.1.1 Online Transfer learning

The first research work about TL was mainly made in an offline fashion, assuming static prior knowledge already given since the beginning of the student's learning process. This assumption is too strong and unsuitable for real-world applications, in which training samples may be available in an online, i.e. sequential, way. So, the main goal becomes combining the approach of online learning to TL. To this extent, different frameworks have been proposed [94, 93, 13]. In fact, a possible issue is that the learner can get stuck in a sub-optimal state. Expert's advice allows the learner to pass from the current state to a better one according to the teacher, i.e. expert, knowledge. In [94], online transfer learning topic is discussed and Zhao et al. focus on the issue of negative transfer when dealing with homogeneous domains. In the case of heterogeneous domains, they consider the issue of inconsistency of the feature space. The authors apply also online transfer learning to address complex real-world problems, e.g. spam detection and climate science and forecast. In [93], online transfer learning is applied with the focus on *Teacher-Student* RL by Zhan et al. They provide a formal understanding of the phenomenon of receiving advice by students, as well as implications on its learning process, e.g. convergence properties. The authors point out that receiving a finite amount of advice, the student cannot increase the asymptotic performance. This element is very important for the teaching process and the learnt policy. In [93], the authors propose a transfer method found on instance transfer. There is not a finite sample performance guarantee in their theoretical analysis of the asymptotic convergence. A different approach is proposed in [79], in which Tirinzoni et al. propose to transfer the Q-function from old to new tasks with a variational method under Bayesian setting. In [13], Cakmak et al. focus on the application of optimal teacher, with the goal of teaching the reward function of an MDP. In this context, the learner aims at identifying a reward function that is consistent with demonstrated actions of a user, identifying the corresponding optimal policy. The agent executes the trajectory that maximizes the reward, following the initial states chosen by the teacher. The authors focus on the teacher optimization problem, to generate the best set of demonstrations to support the student.

4.2 Teacher-Student Reinforcement Learning

Teacher-Student RL is a subfield of MARL aiming at improving the learning procedure, defining an experienced *Teacher* that helps the *Student* providing advice. In such case, the transmission of knowledge is within the same task. This thesis project is intended to apply Teacher-Student RL methodologies to the field of robotics for autonomous driving. Therefore, related approaches already treated in the state-of-the-art are presented in this section. So far, approaches that are related to action advice in the framework of Teacher-Student RL have been proposed [27, 93, 13] (see Section 4.1). An alternative strategy, as the one adopted in this project thesis, is the one of parameter optimization in the teaching framework. A similar concept can be found in [46], in which the application field is related to acoustic components in a far-field speaker system. In this case, the approach was to learn the optimal parameters in a small-size network, i.e. the student, obtained with model compression over the teacher, that could approximate with low error (e.g. divergence in output distribution) a deep network, i.e. the teacher [46].

4.2.1 Budget in Transfer Learning

An interesting feature is the adoption of enough budgets [85, 38]. Teacher's advice has the purpose of speeding up the student's learning, and the principle of budgets is to specify only a limited number of advice that can be transmitted from teacher to student [85, 38]. Different algorithms have been evaluated on specific benchmarks problem, e.g. Pac-Man and MountainCar, to establish how the specific moment in which advice is provided influences performances. This restriction (i.e. the number of times the teacher can support student with advice) is realistic to the extent that it represents the limited attention and patience of human students in the learning process. Furthermore, it also supports those domains in which communication between agents is limited [85, 4]. In [4], the approach is related to alternative tabular algorithms for exploration, i.e. with various degree of efficiency. Possible approaches in this context could be advice performed at early stage, i.e. advice given only when the student knows very little and could benefit more from prior knowledge, or advice performed at specific stages that are the most important, i.e. advice in these stages could lead to a far better performance. Other improved approaches could be advice in case of mistake, i.e. without wasting advice in stages in which the student perhaps already perform well, or advice related to the predictive student

policy, i.e. predicting student policy from its current behaviour can tell the teacher which states and actions would be encountered and so apply prior corrections [85]. In [85], Torrey et al. study how agents that interact in a new environment can benefit from the input of humans, or also experienced teachers, regarding which action to select next. It is perhaps the first proposal of *Teacher-Student* framework with limited amount of advice. The authors demonstrate significant gains in learning when using heuristics to guide the choices of the teacher for the advising strategies, e.g. concept of state importance [85]. They introduce the concept of setting budget with a limited advice, i.e. attention budget considering the student capacity to give attention, and specify constant monitoring from the teacher with respect to the student. In [38], Ilhan et al. propose a Deep Reinforcement Learning version of [23], in which each state has an estimation of the number of visits that determines the confidence to establish whether to require advice (i.e. from student to the teacher) or not. To this extent, the authors exploit a Deep Neural Network, keeping same features as [23]. In this context, the authors aim at preventing to spend budget on previously queried states, to avoid wasting the allowed budget for the teacher.

4.3 Autonomous driving

Autonomous driving is an active research field in control systems and computer vision. Real world human's life can really benefit from this applications. Hence, many companies are nowadays devoted to the development of advanced autonomous driving cars.

4.3.1 Learning Methods

The state-of-the-art saw a great effort in recent years regarding the task of learning on a specific track. In this Section, research works regarding the field of autonomous driving learning are presented. Many research works are also related to the field of computer vision for image detection. In [37], Huval et al. propose a framework in which Convolutional Neural Networks (CNN) [47] are applied for decomposing the autonomous driving problem into subproblems, i.e. car detection, task of lane detection and method evaluation in a real-world highway dataset scenario. The authors used Camera, Lidar, Radar, and GPS for building such a dataset, and the Overfeat CNN detector for the model [68, 37]. A different perspective is applied in [8], in which Bojarski et al. propose an end-to-end approach, based on CNN, to achieve autonomous driving. The authors present a model architecture that directly

maps raw pixels derived from a single camera to steer commands, avoiding problem decomposition phase. Tests have been performed both on simulator and real-world environment. In [70], Sharifzadeh et al. reach collision-avoidance in motion and human-like behaviour through the use of Inverse Reinforcement Learning (IRL) approach. IRL consists in the inference of the reward function that supports expert demonstrations, i.e. extracting reward function given optimal observed behaviour [57]. The authors exploit Deep Q-Networks [54], i.e. networks that are derived as a variant from Q-learning algorithm, using Deep Neural Networks (DNNs) as (non-linear) Q-function approximator through high-dimensional state spaces. The authors use DQN as IRL refinement step for reward extraction. Differently from previous state-of-the-art, in [69], Shalev-shwartz et al. present a model of autonomous driving that is based on a multi-agent control problem. The authors demonstrate the efficiency of a deep policy gradient method (see Section 3.1.4), via simulator experiments. Many research works [91, 18, 39], focus on autonomous driving based on deep reinforcement learning techniques. In [87], Wang et al. apply 3D object detection derived from a single image as an essential component of autonomous driving. The authors face the issue of the complex state space that the agent deals with in the real world scenario by adopting the DDPG algorithm (see Section 3.1.4). The core point in their work is that DDPG algorithm manages to handle both complex state and action spaces within a continuous domain. The tested environment in [87] is TORCS simulator (see Section 2.1). The single agent is trained with DDPG and the input given to the environment is derived from selected appropriate sensor information. The authors design their own network architecture for both the actor and the critic used in DDPG, i.e. see *actor-critic* in Section 3.1.4.1. Evaluation has been performed in different TORCS modes, that present various visual information. In Figure 4.2, an experimental result of [87] shows that the trained model with DDPG has learnt how to manage throttle pedal before corner, since drifting is one of the main reasons for driving wrongly. The fact that the model has still not learnt to avoid collisions is due to the absence of competitors in the training phase. Other works focus on Deep Reinforcement Learning techniques and development of hybrid controllers [35]. In [35], Huang et al. present a DDPG based end-to-end decision-making model. The mapping of state to action in a continuous way is an analogy to real-world driving style. The environment does not consider the influence of the traffic on vehicle behavior decisions. The authors propose an hybrid controller since they apply RL to the modelling network. In [96], Zou et al. propose a DDPG framework that exploits the concept of imitation learning, i.e. allow agent to imitate expert's

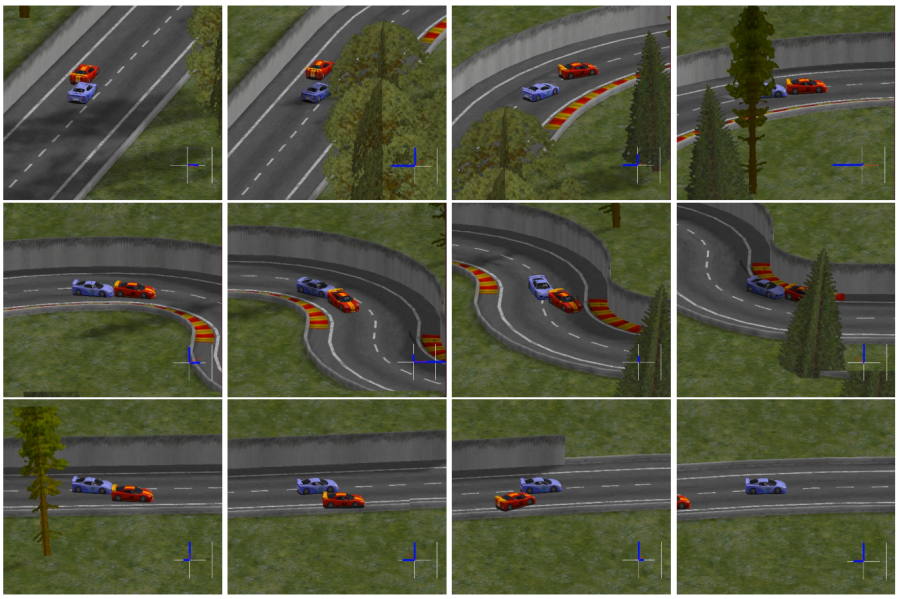


Figure 4.2: Compete Mode of TORCS [87]. The picture shows consecutive images that demonstrate that training, the agent (blue car) has learnt to slow down throttle before corner to avoid drifting, while it has not learnt how to avoid collisions with competitors (orange car).

policy [36]. This approach solves the DDPG issue of being strictly dependent on the initialization parameter settings and slow convergence time. In [32], Graves et al. apply [48] algorithm to autonomous driving with the introduction of General Value Functions (GVFs), i.e. architecture to learn several value functions through one experience stream [76]. GVFs learn a compact representation of the agent's state, i.e. road angle and future lane position, with out-of-policy predictions [32]. Finally, while previous work mainly relate to correct and safe driving style, in [14], Capo et al. focus also on the field of competitive driving for racing games. The authors use DDPG algorithm to develop artificial agents and use TORCS simulator. Different input designs, providing both numerical and visual information, are considered. The core of the agent learning process is to identify a target point on the track, that is subsequently mapped to commands that can be activated by the integration with low-level logic. This allows to push more performance and to design agents capable of learning high-level logic, that is directly mapped to low-level logic control, to reach optimality. This work concentrates on high-level actions, i.e. racing line of car is the action space, while previous work mainly focused on low-level logic, i.e. throttle, brake and steer.

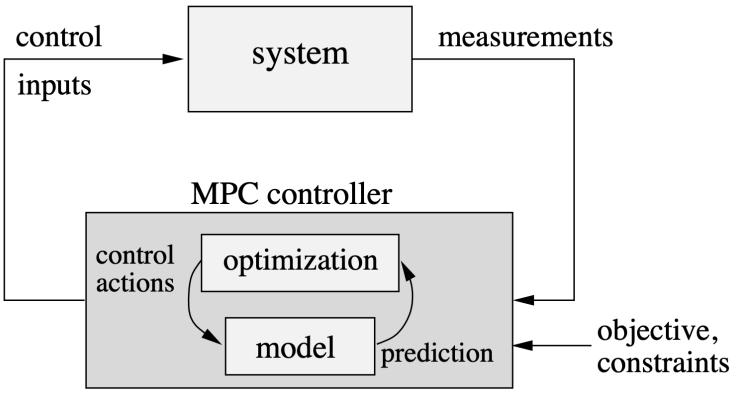


Figure 4.3: MPC schema illustration [6]. At each control step, the MPC controller measures the system current state, then determines which control input to provide, given a prediction horizon and the actions with the best predictive performance based on a given objective function.

4.3.2 Planning Methods

A typical approach in the field of RL for planning is related to Model Predictive Control (MPC). Literature is nowadays rich about MPC, e.g. [56, 30, 61, 1]. MPC consists of an advanced strategy for process control that implies the fulfillment of a set of constraints while a control process is going on. This method has been used in chemistry since 80's [61]. Its applications are wide and, nowadays, it is used in autonomous vehicles, power systems and power electronics. The strength of MPC consists in predicting variations (of the system that is modeled) in dependent variables that are due to variations of independent variables. There are several research work in the state-of-the-art that apply MPC in different application fields. MPC is well suited in applications related to building controls, as well as those applications that require the management of energy resources. There are also applications in real-time scenarios, where short-term accurate predictions are useful (e.g. weather forecast or future prices) [51, 29]. In Figure 4.3, the functioning schema of MPC is schematized. Every time a new control step passes, the MPC controller first measures the system current state, then determines, via optimization, which control input to provide, given a prediction horizon of a fixed number of steps. The input is determined considering which actions have the best predictive performance based on a given objective function. Contemporary widely used path planning algorithms can be partitioned into two stages, i.e. global planning and local planning [34]. In global planning stage, the whole routes and states of vehicles are determined

through a localization system and a digital map. In local planning stage, sensors, e.g. radars or cameras, give surrounding information that, together with a global route, leads to a local path [58]. In [40, 55], path planning methods are analyzed for robots and autonomous vehicles in order to avoid obstacles. In [40], Ji et al. consider vehicle dynamics and limits of actuators within the path tracking through a model predictive path tracking controller. To the extent of preventing collision while tracking the planned path, the authors design a MPC system to calculate the angle in front during steering in order to prevent the collision of the vehicle with a moving obstacle. In [67], MPC is applied to the field of sustainability. Serale et al. focus on Demand Side Management (DSM) methodologies, i.e. methods for district heating and electrical distribution networks. The authors consider the DSM of micro-grid, as it allows to optimize given criteria while explicitly managing to respect system dynamics and constraints. In this context, the final goal is to reach energy efficiency while performing the operation of Heating Ventilation and Air Conditioning (HVAC) systems [67]. In [28], Farshidian et al. apply non-linear predictive control on robotic systems. The authors generate a framework to produce a real-time whole-body non-linear MPC for creating trotting gaits. Planners for legged movement often generate dynamic motion plans as a solution to a complicated nonlinear program. Within only few milliseconds, they generate optimized trajectories for subsequent phases of the motion, demonstrating real-time MPC methods on the HyQ [66] quadruped robot. Another example of possible application is shown in [77], in which Tarragona et al. carry out a bibliometric analysis regarding thermal energy storage systems of smart control applications.

An alternative approach is the one of Artificial Potential Field (APF), that treats the robot as entity that combines repulsion from obstacles and attraction to the goal. This strategy is smooth and safe in planning production, but has slow reaction times, so for real-time applications is often unfeasible. In [55], Montiel et al. propose the use of Bacterial Potential Field (BPF), that ensures an optimal and safe path, as it makes use of the APF method with the addition of a bacterial evolutionary algorithm [43] in order to develop a more flexible path planner technique. This method is applied in the field of mobile robot navigation, to determine the shortest possible path in the least amount of time from whatever start position to the goal. The unknown environment is characterized by moving obstacles.

Chapter 5

Problem Formulation

In this chapter we will formalize the problems that are discussed in this thesis. Most of the terminology and concepts used in this chapter has been briefly presented in Chapter 1, and the details of the approaches and methods mentioned in the next sections are discussed in Chapter 6. The goal of this project can be divided into two main objectives:

1. learning a driving policy, capable of good performance on racing simulators, that we call student controller;

2. formulate a teacher policy, able to transfer information to the student policy, i.e. allow student to improve its driving performances according to the advice provided by the teacher;

The first sub-goal is the standard RL problem in which the policy learns how to behave within a particular environment in an optimal way. However, in the student settings, the policy does not need to be the optimal one but, indeed, it should be sub-optimal in order to allow the improvement provided by the teacher advice. The second topic is more related to a particular instance of RL that is the one of Teacher-Student RL.

5.1 Environment

The first point is to formalize the characteristics of the current environment. In RL scenario, environment is modelled through MDP (see section 3.1.1) $M : \langle S, A, p, R, \gamma \rangle$. In our settings, the agent is a driver that acts on a fixed track with a fixed car setups. We define as environment both the track and the car setups since they both characterize the transition distribution between states. In this case:

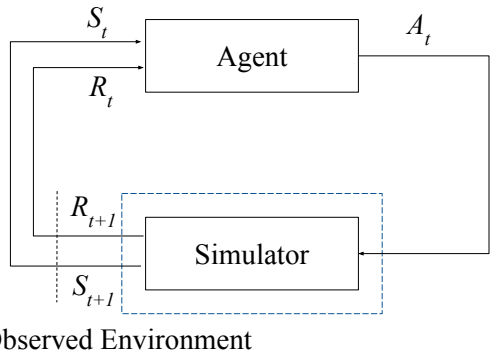


Figure 5.1: Agent-Simulator Environment. State S_{t+1} returned by the Simulator. During the training, the agent observes the Observed Environment, and picks a new action A_t accordingly.

- Set of states S: $s \in \mathbb{R}^n$, with n: number of features. It describes the possible location of the car in the space, but also its dynamics, e.g. velocity and acceleration.
- Set of actions A: $a \in \mathbb{R}^3$ it consists of a 3D space, that includes the inputs that control driving;
  - Throttle: $a_t \in [0, 1]$, i.e. throttle pedal
  - Brake: $a_b \in [0, 1]$, i.e. brake pedal
  - Steer: $a_s \in [-1, 1]$, i.e. steering wheel
- Transition Distribution $p(S_{t+1} = s' | S_t = s, A_t = a)$, that expresses the probability of going from state s to state s' by performing action a. This fact leads to reward r. In real world scenario, the transition probability is mainly related to the car kinematics, but it also strongly depends on the aerodynamics, tires temperatures and other related factors;
- Reward function R is suited to encode goodness and badness of a specific action. The reward function $R(s, a) = \mathbb{E}[\, R_{t+1} | S_t = s, A_t = a]$, expresses the reward obtained by transitioning from state s to state s' with action a.

In Figure 5.1 agent and environment interaction is shown, highlighting the process. State S_{t+1} is returned by the Simulator and reward r_{t+1} is computed. During the training, the agent observes the *Observed Environment*, and picks a new action A_t accordingly.

Environment is accessed via observations $\mathbf{x}_t \in X$, with $\mathbf{x}_t$ a features vector that will be deepened in Section 6. This set is processed in order to build the state s_t. There is a control frequency f_c, that regulates the fixed time step of the sequence of state-action pairs. This allows to determine the overall time length of a sequence. A sequence of state-action pairs is defined as $t : s_0, a_0, s_1, a_1, ..., s_{T-1}, a_{T-1}, s_T$, with s_T corresponding to the trajectory terminal state (i.e. finish line or out-of-track).

5.2 Student Learning

The *Student* is formalized as an agent that wants to maximize the overall expected return:

$$G_t = \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^k r_{k+1} \right], \tag{5.1}$$

having γ discount factor and T the final time step. T changes according to the performed lap, i.e. it is not fixed a priori, it depends on the execution of the lap. We have to specify that the long term goal of the student is to maximize its performance in terms of time. On the contrary, to correctly set up a learning scenario to apply *teacher-student RL*, the goal is to have a student that performs adequately, i.e. is able to complete the lap and reach adequate performances, but is still sub-optimal (and, therefore, it needs to learn and be guided by an expert *teacher*). The objective function of the *student* is defined as:

$$obj = \arg\max_{\pi_{\boldsymbol{\theta}}} \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^k r_{k+1} \right], \tag{5.2}$$

where $\pi_{\boldsymbol{\theta}}$ refers to the policy parameters $\boldsymbol{\theta}$. The *Student* is an agent defined by a deterministic policy $\pi(a|s) = f(s)$ that has the objective of maximizing the return, i.e. the policy is formalized as a parametric function. This problem can be solved by any RL algorithm. In this specific case, we need a margin of improvement to apply the teaching phase. This requirement implies the definition of a new objective function, that presents an additional term of deficiency, to reach the sub-optimality.

$$obj = G_t - D, \tag{5.3}$$

where D is the deficit factor that implies the sub-optimality of the policy. D is a gap term that will be learnt with the advice of the *Teacher*.

5.3 Teacher Learning

The objective of the teaching phase is to obtain the optimal policy, starting from a sub-optimal policy, in the least number of iterations. The problem has as input the sub-optimal policy and a policy that we name *oracle*, i.e. the expert, that knows everything. The optimization phase is an iterative procedure. At every step i, the objective is to minimize the divergence between π_{i-1} and $\tilde{\pi}_i$. $\tilde{\pi}_i$ represents an intermediate policy, i.e. a policy that is in between with respect to the base policy and the *oracle*, such that the *student* is able to learn it in one step. Hence, the choice of the teacher is to define $\tilde{\pi}_i$, such that:

$$V^{\pi_{base}} < V^{\tilde{\pi}_i} < V^{\pi_{oracle}}, \forall \text{ step } i, \tag{5.4}$$

(see Policy Improvement Theorem in Section A.2). The optimal policy is the policy that maximizes the expected total return:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{T} G\left(s_t, \pi_t(s_t)\right)\right], \tag{5.5}$$

where t denotes the time and T is the maximum number of steps. The Equation 5.5 refers to episodic RL with horizon T. This optimal policy represents an advice for the *Student* that has to improve to overcome the deficiency term D. To this extent, the *Teacher* must be defined as corresponding to a trajectory that could be performed by an expert driver.

When the *Student* receives a recommendation for action $a_{teacher}$ in state $s_{teacher}$ by the *Teacher*, then it picks that action as it was the selected one for exploration. Once advice is given, the *Student* doesn't know the value of $Q(s_{teacher}, a_{teacher})$, but it will know for sure that the following property holds:

$$Q(s_{teacher}, a_{teacher}) \geq Q(s_{teacher}, a), \forall a \in \mathcal{A}. \tag{5.6}$$

This means that the *Teacher* identifies $a_{teacher}$ as the most promising action in state $s_{teacher}$, suggesting this information to the *Student*, that will update its policy accordingly.

Chapter 6

Methods

In this chapter we describe the designed methodologies to face the problems we identified in Chapter 5. First, in Section 6.1, we explain the environment according to the described model, see Section 5.1. Then, in Section 6.2, we explain the *student* structure, that faces the problem described in Section 5.2. Finally, we describe the *teacher* implementation in Section 6.3, to overcome the problem defined in Section 5.3. In Figure 6.1, the thesis pipeline execution is shown. A set of human demonstrations conducted in the environment is used for the behavioural cloning phase and the fitting of the transition model through the loss multi-step. The Student is a parametrized rule-based policy that follows a reference trajectory. Planning prediction is derived through the action selection policy, i.e. behavioural cloning policy, and the transition model. Finally the teaching algorithm is applied and an improved policy is generated.

6.1 Environment

Considering the environment, we have the model of TORCS defined in Section 2.1. Besides, we model the MDP that defines the environment with a *Transition Distribution* p that is encoded in the simulator functioning and code. This is considered as a black box and not deepened in this thesis, as it would be too complex otherwise. Considering the *Reward Function*, the agent gets a reward of 0 when the vehicle reaches the finish line, so in the terminal state corresponding to a success. For the specific rewards that are collected in each state of the track, the specific formulation is strictly dependent on the faced problem. There is the specific function *reward* $r(\cdot)$ for this purpose. In case of a single lap, the intrinsic definition is a quantity that is inversely proportional to the final lap time, so this could be

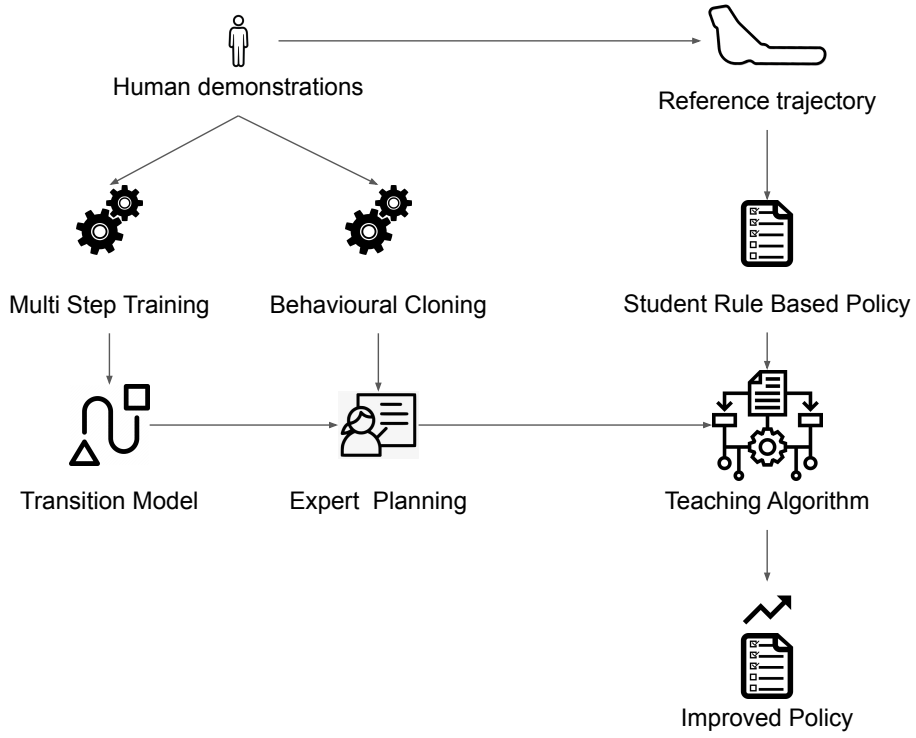


Figure 6.1: Thesis Pipeline Execution. Student is a parametrized rule-based policy that follows a reference trajectory. The set of human demonstrations is exploited to define the transition model and the policy adopted by the planning algorithm. Finally, the plan is exploited to perform teaching over the Student.

a reasonable definition. Nevertheless, this strategy is not suitable since it means that we would assign at the end of the episode a single signal. This approach introduces a wide spread between final *return* and agent's actions, so undermining algorithm's convergence. Other strategies will be adopted and described. In particular the *return* G_t assumes the following structure:

$$G_t = \begin{cases} 0 & \text{if final state is reached} \\ -N & \text{if a collision occurs} \\ -1 & \text{otherwise,} \end{cases} \tag{6.1}$$

in which N represents the number of steps reached at the time of collision. Since the reward function counts the overall the number of steps, an execution of PGPE algorithm presenting this type of reward function would have been subjected to divergence with respect to the target point. In fact, considering a non-terminating lap as starting point, the algorithm would

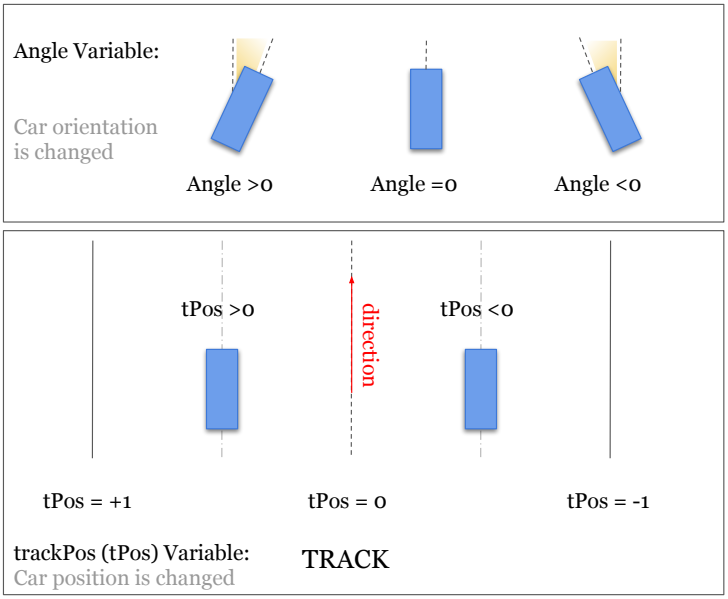


Figure 6.2: MDP - Example of state features, trackPos and angle. trackPos encodes the position of the vehicle with respect to the middle of the track. angle encodes the orientation with respect to the axis track and the direction of movement.

have been induced to terminate the lap sooner, with respect to the previous iteration, to get a higher reward, without being able to reach the goal of completing the lap in the least possible time. A reward function that depends on the number of steps is useful, since we are interested in improving time performances. Furthermore, to manage the case of non complete lap, a penalty of non-termination, equal to the worst possible lap execution reward function is applied. In Figure 6.2 some of the features that define the state and that are then treated in Section 6 are shown. *trackPos (tPos)* is a variable of state that encodes the position of the vehicle with respect to the middle of the track, while *angle* encodes the orientation with respect to the axis track and the direction of movement. Value of *trackPos* is zero when the vehicle is exactly in the center of the track, while it moves towards $+1$ if the car is in left half of the track. Symmetrically, it moves towards -1 in the right half. A positive value of *angle* means that the car is turned right, while a negative value means the opposite. There is one starting state S_0, while there are many terminal states. *Terminal state* is the episode final state and in this scenario two situations may arise. The first, and success, is when the car reaches the finish line of the of track. The second situation arises when termination is reached due to an off-track or head-tail. In Figure 6.3 steering action is illustrated according to the mathematical codification. A positive

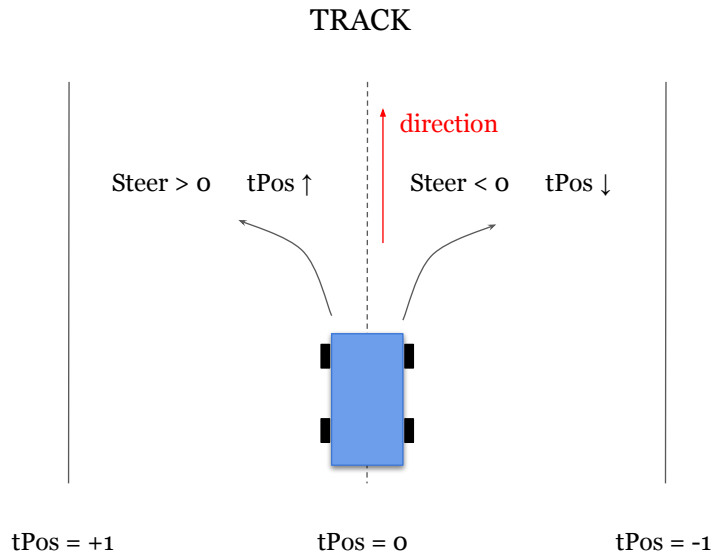


Figure 6.3: MDP - Steer action in relation to trackPos state feature. When steer function is positive, trackPos increases and the car drives towards left. When steer function is negative, trackPos decreases and the car drives towards right.

value for steer means to steer left and increase the value of *trackPos*, while a negative value implies steering right and decrease *trackPos*.

6.2 Student

Student Policy is designed to have as main characteristic the ability to generalize, following the Teacher's instructions and advice. To this extent, a deep theoretical study of the track topology and car features has been performed, and some differentiable equations have been realized. The goal of these equations is to develop a Controller able to correctly perform a lap with acceptable performances, having the task of learning its parameters to follow the Teacher Policy. The actions of the MDP we describe above are brake, throttle and steer. The *student* policy was designed as a rule-based policy composed of three equation, one for each action. These equations must be differential with respect to their parameters to allow gradient based optimization techniques that, as we will see, form the basis of the teaching step. The student is structured to follow a given reference trajectory. This is done in continuity with the teaching approach where the teacher will provide the right trajectory of the student.

Since the agent drives in racing context, the track contains different driv-

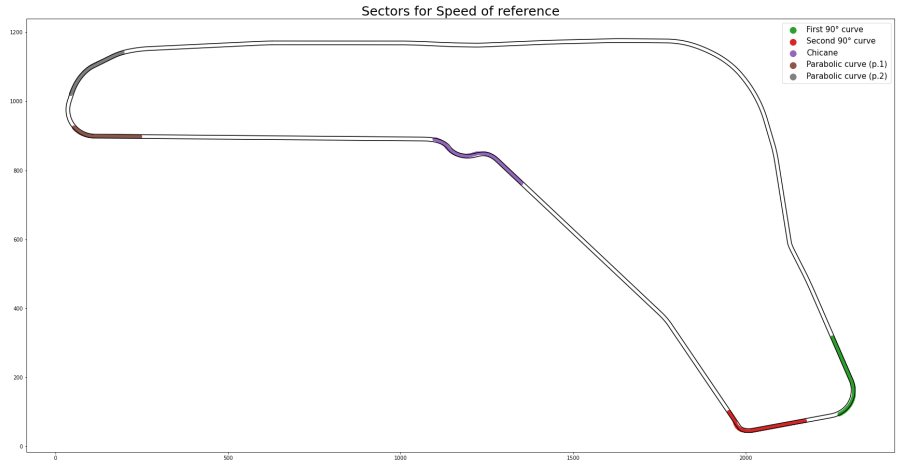


Figure 6.4: Sectors classification that will be detailed in Experiments Section, see 7. The white traits represent parts in which the standard rules of driving are applied. The coloured traits represent specific track features in which rules need adaptation.

ing conditions that needs to be tackled in the right manner. Therefore, we divided the track in sectors with the goal to put emphasis and characterize the most critical track sectors. In particular, in Figure 6.4 the white parts of the track are those in which no specific feature is needed, because the track is straight or easy to handle. The coloured parts represent the specific sectors' divisions and are sectors in which the driving style must be modified because they are critical. For instance, the green sector is a hard turn and the agent must use a very high steer. They are needed in particular for the reference speed, that cannot be kept as maximum in curves or chicane sectors, as well as *Angle* and *Tpos*, that are the reference parameters for this basic policy. The legend is further detailed in the experiments Chapter, see Chapter 7, Section 7.1. In which there is sectors' classification for experiments regarding the reference speed and the analysis of *Tpos*. Variables used in the Student Controller are detailed in Table 6.1.

6.2.1 Model Definition

The *student* policy is designed to follow a reference trajectory that, given the track position, returns the values of speed and direction to follow. The reference that is followed is based on the sectors' division previously mentioned. Criteria of sectors' division are themselves parameters of the policy that must be learnt. Equations will follow the reference, meaning that they will be based on the relative error between the current value of a variable

Table 6.1: Variables for Student Policy

Variable	**Definition**
v	Actual Speed
v^*	Reference Speed
$Tpos$	Actual *trackPos*
$Tpos^*$	Reference *trackPos*
a	Actual *angle*
a^*	Reference *angle*
ε_v	$: v^* - v$
ε_{Tpos}	$: Tpos^* - Tpos$
ε_{angle}	$: a^* - a$
$slope_b$	Slope of brake sigmoid equation
$slope_t$	Slope of throttle sigmoid equation
K_P	Proportional coefficient of PID
K_I	Integral coefficient of PID
K_D	Derivative coefficient of PID

and the reference value. In Table 6.1, this is represented by ε_v, i.e. the relative error between the reference speed v^* and the actual speed v, ε_{Tpos}, i.e. the relative error between the reference $Tpos$ and the actual value of $Tpos^*$, and ε_{angle}, i.e. the relative error between the reference angle a^* and the actual value a. The references v^*, $Tpos^*$ and a^* are specific of each sector, see Figure 6.4. This specialization is due to the peculiarity of each track trait that needs to be considered in the proper manner. The bounds of the sectors are themselves variables that will be optimized with PGPE. The other parameters in the Table, that characterize the *student* model are explained below. As said before, the student policy is composed of one equation for each action $a = f(\cdot)$. Every equation has its own set of parameters $\boldsymbol{\theta}_i$ that form the policy parameter vector $\boldsymbol{\theta}$, parameters that will be learned. The student model is defined by a set of equations that define the possible actions allowed for an agent in the MDP described in Section 5.1. So, for every action we will have a closed-form expression of the following type:

$$a_t = throttle\left(\boldsymbol{\theta}_0^t, ..., \boldsymbol{\theta}_i^t\right), \tag{6.2}$$

$$a_b = brake\left(\boldsymbol{\theta}_0^b, ..., \boldsymbol{\theta}_i^b\right), \tag{6.3}$$

$$a_s = steer\left(\boldsymbol{\theta}_0^s, ..., \boldsymbol{\theta}_i^s\right). \tag{6.4}$$

Every closed-form function is differentiable, in order to allow the correct calculation of the gradient. This means that the first derivative of the action

exists and is continuous (i.e. $f(x)$ approaches $f(c)$ if x tends to c). These conditions are expressed by the following system

$$\begin{cases} \exists a'_t \wedge \lim_{x \to \theta^t} a_t(x) = a_t(\theta^t) \\ \exists a'_b \wedge \lim_{x \to \theta^b} a_b(x) = a_b(\theta^b) \\ \exists a'_s \wedge \lim_{x \to \theta^s} a_s(x) = a_s(\theta^s). \end{cases} \tag{6.5}$$

Furthermore, there are parameters and hyper-parameters, i.e. parameters that will be considered fixed since they reflect intrinsic properties of the action structure or their change does not influence significantly the overall performance. To optimize the performance of the student controller, the following step is defining a set of trainable parameters $\boldsymbol{\theta}_i^a$, with i the parameter index and $a \in \{t, b, s\}$ the specific action. PGPE approach determines the best parameters $\bar{\theta} = \{\bar{\theta}^t, \bar{\theta}^b, \bar{\theta}^s\}$ to reach the best performance. Let's denote with α a generic policy, able to complete one lap in an acceptable time length. α cannot be a random policy as the problem is too specific and complex to guarantee a sufficient quality for the guiding style able to reach the finish line. At this point, we can formulate the student policy $\bar{\pi}$ in the following way:

$$\begin{gathered} G_t^{\alpha} \ll G_t^{\bar{\pi}} < G_t^{*} \\ \mathbb{E}_{\alpha}\left[\sum_{k=0}^{T} \gamma^k r_{k+1}^{\alpha}\right] \ll \mathbb{E}_{\bar{\pi}}\left[\sum_{k=0}^{T} \gamma^k r_{k+1}^{\bar{\pi}}\right] < \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{T} \gamma^k r_{k+1}^{\pi^*}\right]. \end{gathered} \tag{6.6}$$

In Equation (6.6), policy π^* is the optimal policy, so with the optimal parameters $\boldsymbol{\theta}^*$, while $\bar{\pi}$ is the sub optimal policy that the student controller follows, getting considerable performance results. In particular, $\bar{\pi}$ policy aims at satisfying the objective function obj, in Equation (5.2). Reward of the policy exploited by the student is much greater than a basic policy that is able to correctly complete one lap, meaning that its result is valuable even if still not optimal.

$$\pi^* = \begin{cases} a_t = throttle\left(\theta_*^t\right) \\ a_b = brake\left(\theta_*^b\right) \\ a_s = steer\left(\theta_*^s\right) \end{cases} \qquad \bar{\pi} = \begin{cases} a_t = throttle\left(\bar{\theta}^t\right) \\ a_b = brake\left(\bar{\theta}^b\right) \\ a_s = steer\left(\bar{\theta}^s\right). \end{cases}$$

In this case $\boldsymbol{\theta}_*^a$ is the vector of parameters $\boldsymbol{\theta}_i^*$ for the single action a. The meaning of the above formulation is that the reward G_t^*, obtained with policy π^*, that exploits parameters $\boldsymbol{\theta}^*$ is always larger than the reward of all the other sub optimal policies $\bar{\pi}$.

The student policy formal definition is therefore $\bar{\pi} = \pi_{\bar{\theta}}(a_t|s_t)$. The probability distribution over the parameters is defined as $p(a_t|s_t, \rho)$ as described

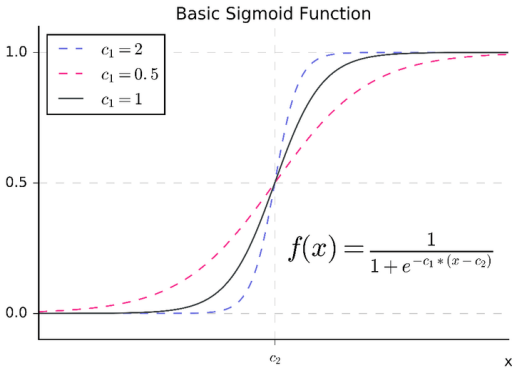


Figure 6.5: Sigmoid Function: Example of widely used smooth function.

in Section 3.3.1, with ρ hyper-parameters that determine distribution over the parameters $\boldsymbol{\theta}$.

A driving style quality requirement is that the change in action values should not be too immediate, meaning that the function that describes the action must not be nervous. This is particularly true for the value of *steer* action. Abrupt changes are not valuable and therefore some form of regularization must be applied. The concept of *smoothness* recalls the conditions about differentiable functions, see Equation (6.5). For all the equations that define the policy π we have to impose that every function defining actions, a_k with $k \in t, b, s$, is smooth. A function is defined *infinitely differentiable, smooth, or of class* C^∞ if it admits derivatives of all orders. For our problem, a function is smooth enough if it admits first order derivatives and this requirement is already presented in Equations (6.5). An example of smooth function can be seen in Figure 6.5, the basic sigmoid function, that is defined for every possible value of input variable x and has a range of outputs $y \in (0, 1)$.

6.2.1.1 Brake Definition

Brake has been defined as a function depending on the error between the reference speed and the actual speed, ε_v, see Table 6.1, with $\varepsilon_v = v^* - v$.

$$a_b(\varepsilon_v) = brake(\varepsilon_v) = \frac{1}{1 + \exp\left(slope_b \times (-\varepsilon_v - 10)\right)}. \tag{6.7}$$

The need for the specific Student policy was to have a brake such that its effect was as quick as possible when speed has to be reduced. To this extent, the most effective and at the same time simple function was the sigmoid. The sigmoid has been defined with a parameter that multiplies

the argument of the exponential, in order to allow to increase the slope of the function, making the grow faster and, therefore, braking faster. Value 10 is subtracted as argument, in order to start braking when the difference between the speeds is greater than this threshold. As the slope of brake is a parameter that can influence the driving style of the driver, it has been one of the first parameters learned in training.

6.2.1.2 Throttle Definition

As for Brake feature, the same reasoning has been performed for Throttle. Also this action has been defined as a function depending on the error between the reference speed and the actual speed, ε_v, see Table 6.1, with $\varepsilon_v = v^* - v$.

$$a_t(\varepsilon_v) = throttle(\varepsilon_v) = \frac{1}{1 + \exp\left(slope_t \times (-\varepsilon_v)\right)}. \tag{6.8}$$

In this case, threshold is directly 0, no term is subtracted as argument of the exponential because the goal is always to drive as fast as possible, so the need is to push throttle at maximum as soon as needed, while brake is activated only when it is really necessary. Also in this case, the equation is modelled as a sigmoid function (i.e. both brake and throttle), because their value must belong to the interval $(0, 1)$.

$$\begin{aligned} a_b(\varepsilon_v) &\in (0, 1) \\ a_t(\varepsilon_v) &\in (0, 1) \end{aligned} \tag{6.9}$$

As the slope of throttle is a parameter that can influence the driving style of the driver, it has been one of the first parameters learned in training.

6.2.1.3 Steer Definition

Steer is the most important action, since the driving style is very sensitive even to tiny changes in these settings. In order to make the action smoother, the choice was to apply a PID control over the action [3, 41]. The purpose of this control strategy is to consider error in relation to the present moment, with the Proportional term, but also to the past, with the Integral that considers the accumulated error, and to the future, with the Derivative term that functions as a prediction of the next error, see Figure 6.6. The proportional component of the steer equation depends on the error of two quantities. The orientation of the car on the track is determined by *trackPos*, that, as said in the previous sections, measures how much the car is far from the center of the track, and *angle*, which represents the orientation of the car

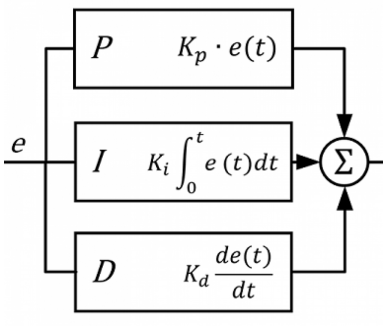


Figure 6.6: PID regulator schema [9]. K_p is the proportional component, that considers the error in relation to the present moment. K_i is the integral term, that considers the error related to the past, i.e. the accumulated error. K_d is the derivative component, that considers the error in relation to the future, i.e. with the Derivative term that functions as a prediction of the next error.

with respect to the curvature of the track. The reference quantities that the *student* must follow are expressed in these two terms and consequently the error term is expressed as the weighted sum between the position of the car and the reference, i.e. ε_t and ε_a. Since *trackPos* and *Angle* have different scales, the weights are chosen to let them have the same importance in the equation. Sign -1 for ε_a is due to the sign of angle and the delta w.r.t. the steer sign.

$$\varepsilon = \frac{\varepsilon_t}{100} + \frac{-\varepsilon_a}{10} \tag{6.10}$$

where the error ε is defined on the basis of the errors on *trackPos* and *angle*, see Table 6.1,with $\varepsilon_t = Tpos^* - Tpos$ and $\varepsilon_a = a^* - a$. Steer equation is based on a PID controller and is defined as the sum of three components. $steer = P + I + D$. Components are defined in the following way:

$$a_s(\varepsilon, d) = steer(\varepsilon, d) = P^t(\varepsilon^t) + I^t(\varepsilon^t) + D^t(\varepsilon^t). \tag{6.11}$$

PID controller for the steer with learnable parameters K_P, K_I and K_D:

$$\begin{cases} P^t(\varepsilon^t) = K_P \times \varepsilon^t \\ I^t(\varepsilon^t) = I^{t-1} + K_I \times \varepsilon^t \times \Delta T \\ D^t(\varepsilon^t) = K_D \times \frac{\varepsilon^t - \varepsilon^{t-1}}{\Delta T}, \end{cases} \tag{6.12}$$

with ε^{t-1} denoting the error measured at the previous iteration. Since the track has very tight curves, the Equation (6.12) is not enough to handle them. Therefore, we introduced some boost in case of curves. The final

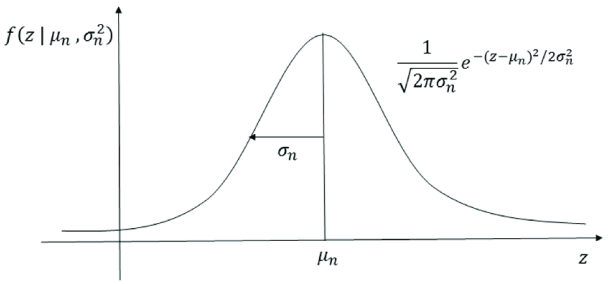


Figure 6.7: Gaussian Function with mean μ and variance σ^2

equation can be written as:

$$a_s(\varepsilon, d) = steer(\varepsilon, d) = PID(K_P(\varepsilon), K_I(\varepsilon), K_D(\varepsilon)) \cdot (1 + BOOST(d)). \tag{6.13}$$

The introduction of this element is necessary since the the equations that model the steer are based on *trackPos* and *angle*; however, they do not represent the curvature of the track that becomes really important for heavy turns in the track. Since the speed required by the vehicle is considerably high, such equations alone, together with PID controller, do not allow reactivity fast enough in curves. This is also due to the Environment TORCS and the kinematics of the car. In fact, for the steering action the values that are required in curves are superior of at least an order of magnitude with respect to a normal swing in the track. The $BOOST$ is modeled with a set of Gaussian functions that are based on the state feature $distFromStart$, i.e. the distance from the start, that codes the degree of progress on the track and, consequently, the specific point of the sector. The number of used functions is an hyper-parameter, while mean and standard deviation are parameters that are learnt in the optimization phase. Hence, the function of the steer can be rewritten as:

$$a_s(\varepsilon, d) = steer(\varepsilon, d) = PID(K_P(\varepsilon), K_I(\varepsilon), K_D(\varepsilon)) \cdot (1 + k \sum_{i \in \Psi} g(d, \mu_i, \sigma_i)), \tag{6.14}$$

with g representing a Gaussian function with mean μ and standard deviation σ.

$$g(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{6.15}$$

The choice of modelling the boost as a Gaussian is due to the necessity of having a smooth driving style. After the climax, k, i.e the point in which we have to apply the maximum boost in the Section, it is good to go back to a standard steer value gradually. Therefore, Gaussian function form is a

suitable and nice choice. The parameters that define the boost are related to the physical conformation of the track, not directly to the driving style, e.g. *throttle*, *brake* and *basic steer (i.e. only PID)*, that depend on errors defined in relation to the driving style variables (i.e. *angle*, *trackPos* and v_{ref}). Even though, also the parameters that defined the mean of the boost, that will be called *breakpoints* from this point on, can be learnt. Their values and properties in performance will be learned in Section 7 with PGPE as well as other reference parameters. Defining Ψ the set of Sections in which the boost must be applied, i.e. see first column in Table 7.1. We have $\Psi = \{$First curve, Mini chicane, First 90° curve, Second 90° curve, Chicane, Parabolic curve$\}$. The $BOOST$ expression is formulated as:

$$BOOST = k \sum_{i \in \Psi} g(d, \mu_i, \sigma_i)$$

$$\begin{aligned} BOOST &= k \cdot g(d, \mu_1, \sigma_1) \\ &= k \cdot g(d, \mu_2, \sigma_2) \\ &= k \cdot g(d, \mu_3, \sigma_3) \\ &= k \cdot g(d, \mu_4, \sigma_4) \\ &= k \cdot g(d, \mu_5, \sigma_5) \\ &= k \cdot g(d, \mu_6, \sigma_6). \end{aligned} \tag{6.16}$$

The value of k is a coefficient to regulate the intensity of the steer boost. The track has 6 critical points in which the boost needs to be applied. They are shown in Figure 6.8. In Figure 6.8, the $BOOST$ functioning in shown. The darkest point in the Section is the mean μ in which the boost reaches the highest value, k. The nuance means that the boost intensity is smaller and decreases as we move further from the mean μ. So, in this case we will have a boost value β, such that $\beta < k$ and $\beta \to 0$ as we approximate the end of the Sector. This is the effect of the standard deviation σ, due to the form of the Gaussian function, see Figure 6.7. The white colour in the track means that the boost is not applied, so the *steer* expression depends only on the PID controller action. An important and curious note is that for the curves Sections $i \in \{$First curve, First 90° curve, Second 90° curve$\}$, the boost must be applied a bit before the curve physical realization. This is a mathematical formulation to start steering before the curve, without going out of track and is a trick to model the reactivity of the driver.

6.2.2 Student optimization with PGPE

In Table 6.2, a complete resume of the variables that define the *Student* policy together with the related action is presented, considering brake and

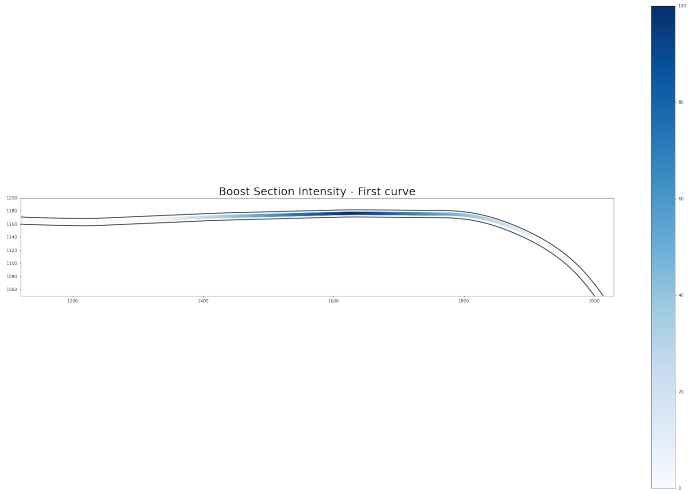


Figure 6.8: First curve. Graphical representation of the Sections in which the boost is applied. There is an increasing intensity in color as we move in the middle (i.e. mean μ), and the nuance in blue represents the variance of the Gaussian. The boost is zero in the white parts, meaning that the steer is composed only of the PID component.

throttle action. A complete overview of the steer model is detailed in the dedicated Table 6.3. These Tables contain all the students parameters. PGPE is the chosen algorithm to optimize them.

PGPE algorithm, see Section 3.3.1, requires two policies, an action-policy and an hyper-policy. The hyper-policy is implemented as a multivariate Gaussian with diagonal covariance, where each component represents the mean and variance of a certain parameter of the action policy. At each iteration of PGPE, the hyper-policy samples a number k of policy parameters $\boldsymbol{\theta}$. For each of them, the action policy is executed over TORCS to make one episode saving the return. All the policies running with such parameters (i.e. number of batch) generate a certain reward and the overall batch result is used to update the hyper policy through the gradient that maximizes the expected return. The number of iterations (i.e. number of gradient steps) is a hyper-parameter of the optimization procedure and given as input. There is the possibility of using a *baseline*, as in REINFORCE (see Section 3.3.1.3), that implies the correction of the gradient of a certain quantity (i.e. the baseline quantity), to speed up and facilitate convergence. In our case, we used the *baseline* as it proved to have better results. The optimizer we used

Table 6.2: Resume for Student Policy: Variables and Actions for Brake and Throttle. It is evident from this schema that the actions of brake and throttle depends in principle only on the reference speed. The slope defines the reaction rate, but the core element of definition is the relative error between actual and reference speed.

Action	**Variable**	**Definition**
Brake	v	Actual Speed
	v^*	Reference Speed
	ε_v	: $v^* - v$
	$slope_b$	Slope of brake sigmoid equation
Throttle	v	Actual Speed
	v^*	Reference Speed
	ε_v	: $v^* - v$
	$slope_t$	Slope of throttle sigmoid equation

Table 6.3: Resume for Student Policy: Variables that compose Steer. Consider that the boundary variable is needed to identify a generic point of the track (in terms of distance from the start), that is needed to identify a sector. An example of sectors can be seen in Figure 6.4.

Component	**Variable**	**Definition**
Reference values	$Tpos^*$	Reference *trackPos*
	a^*	Reference *angle*
Actual values	$Tpos$	Actual *trackPos*
	a	Actual *angle*
Relative errors	ε_{Tpos}	: $Tpos^* - Tpos$
	ε_{angle}	: $a^* - a$
PID component	K_P	Proportional coefficient of PID
	K_I	Integral coefficient of PID
	K_D	Derivative coefficient of PID
BOOST	k	Coefficient of the Gaussian that defines boost intensity
	μ_i	Mean of the Gaussian for the boost in sector i
	σ_i	Standard Deviation of the Gaussian for the boost in sector i
Boundary	l	Point to identify the boundary of a sector *A sector identifies a critical part of the track, e.g. a strong curve, see Figure 6.4.*

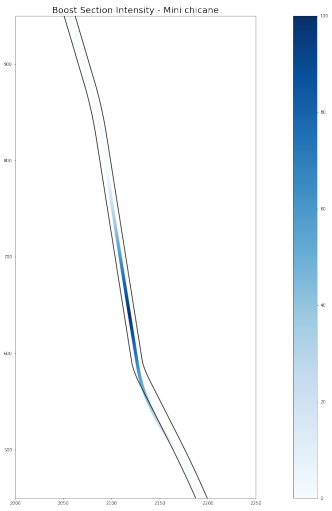


Figure 6.9: Mini chicane. Graphical representation of the Sections in which the boost is applied. There is an increasing intensity in color as we move in the middle (i.e. mean μ), and the nuance in blue represents the variance of the Gaussian.

is *Adam*. Given one parameter vector $\boldsymbol{\theta}$, only one episode is generated since we face the deterministic case (i.e. running more episodes for one $\boldsymbol{\theta}$ would give same result). To perform evaluation, the policy that corresponds to the mean of the hyper-policy distribution and an evaluation lap is run. This allows for the study of the evolution of returns of parameters $\boldsymbol{\theta}$ and the selection of the most reasonable checkpoint.

The action-policy of reference is the *student* controller. The hyper-policy is implemented apart and exploits the basic policy, having as input values the initial values of the distributions (i.e. the hyper-parameters). Having K parameters (i.e. parameters of the *student* controller), the overall number of hyper-parameters is $2K$, since it considers both the mean and the log standard deviation of parameter $\boldsymbol{\theta}_k$, i.e. μ_{θ_k} and $\log \sigma_{\theta_k}$, with $k = 1, ..., K$. The logarithm of the standard deviation is used in order to have a positive value. The distribution predicts the logarithm of the standard deviation, that is always a positive value. This ensures parameters to be positive and allows to use the standard Gaussian distribution, with no need to create a dedicated distribution. For every parameter $\boldsymbol{\theta}_k$ the hyper-policy samples from a distribution that has ρ_k mean, with $\rho_k = \mu_{\theta_k}$ and standard deviation $\exp^{\log \sigma_{\theta_k}}$. The prediction is computed through the action-policy of reference.

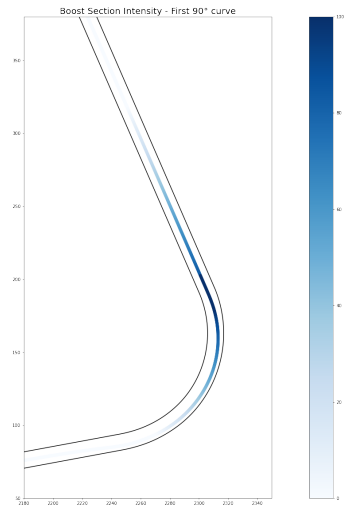


Figure 6.10: First $90°$ curve. Graphical representation of the Sections in which the boost is applied. There is an increasing intensity in color as we move in the middle (i.e. mean μ), and the nuance in blue represents the variance of the Gaussian.

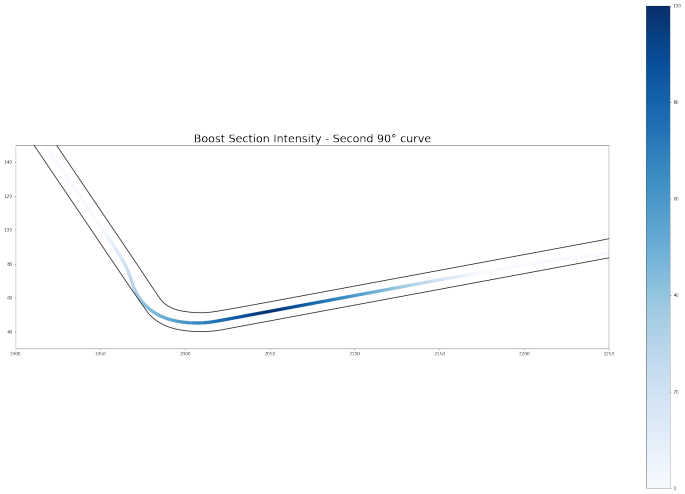


Figure 6.11: Second $90°$ curve. Graphical representation of the Sections in which the boost is applied. There is an increasing intensity in color as we move in the middle (i.e. mean μ), and the nuance in blue represents the variance of the Gaussian.

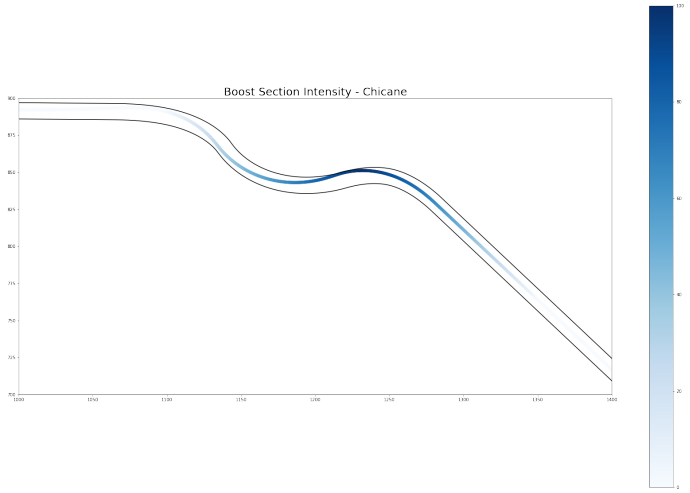


Figure 6.12: Chicane. Graphical representation of the Sections in which the boost is applied. There is an increasing intensity in color as we move in the middle (i.e. mean μ), and the nuance in blue represents the variance of the Gaussian.

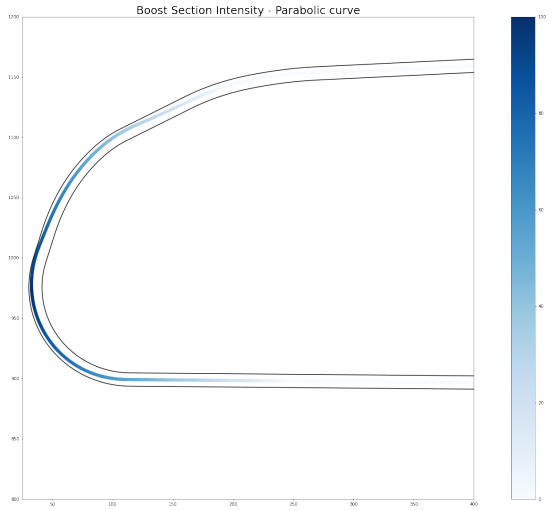


Figure 6.13: Parabolic curve. Graphical representation of the Sections in which the boost is applied. There is an increasing intensity in color as we move in the middle (i.e. mean μ), and the nuance in blue represents the variance of the Gaussian.

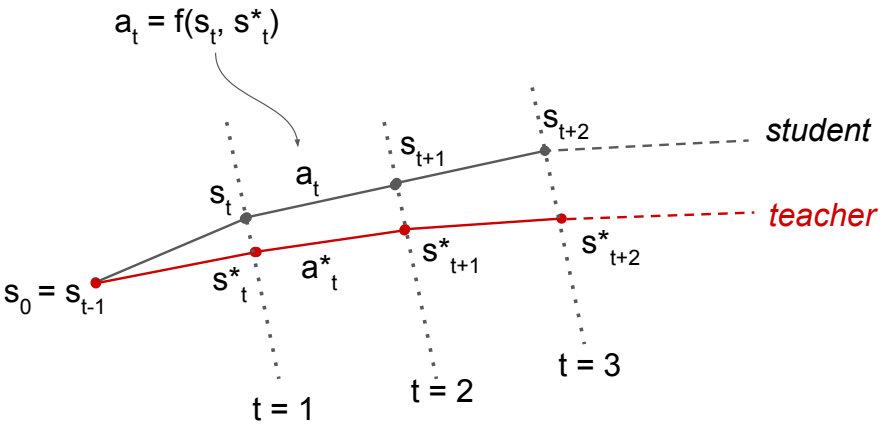


Figure 6.14: Teaching Schema. Black trajectory refers to the student, while the red trajectory refers to the teacher. s_{t-1} is the common starting state, i.e. the point in which the teacher starts giving advice. s_t identifies student trajectory state. From state s_t the student can reach state s_{t+1} picking action a_t. Analogously, s_t^* identifies teacher trajectory state. From state s_t^* the student can reach state s_{t+1}^* picking action a_t^*.

In the end, after the optimization is performed, convergence of the hyper-parameters is reached, i.e. the mean moves towards the optimal value for the given parameter and the standard deviation reduces. The final results of μ_{θ_k} are kept as input value of the *student* policy. This procedure allows to find the best parameters for the *student* policy.

6.3 Teacher

Teacher assumes the role of an oracle that provides the optimal trajectory. In a specific point of the track, the *teacher* provides the optimal trajectory, from that point up to horizon H. *Student* must learn the advice provided by the *teacher* and behave accordingly. In Figure 6.14, the black trajectory refers to the *student*, while the red trajectory is the optimal one provided by the *teacher*. Reference state for *student* Equations (6.7), (6.8), (6.13) is the current state. The functions define the action on the basis of the actual state s_t and reference s_t^* currently considered. The single action, i.e. (6.7), (6.8), (6.13), is defined as $a_t = f(s_t, s_t^*)$, where s_t is the current state and s_t^* is the current reference, on the basis on which the error is defined. Since the *student* is sub-optimal, when in state s_t, it is not directly able to perform action $a*_{t+1}$ and reach $s*_{t+1}$. In its basic trajectory, the *student* performs

action a_{t+1}, reaching s_{t+1}. The loss in Equation (6.17) is calculated on the basis of the best action a^*_{t+1}, that is the best action provided by the *teacher*, and of the action that the *student* is able to perform given the optimal reference provided by the teacher. To this extent, a fundamental assumption that holds at iteration 0 is that $a_t = f\left(s_t, s^*_t\right) \neq a^*_t$. The learning process has the goal to minimize this gap. This ensures that the *student* is not directly able to follow teacher's action in the current state, but has to minimize a loss function, defined as:

$$\mathcal{L}_t = \|a^*_t - f(s_t, s^*_t)\|^2, \forall t \in \{0, H\}, \tag{6.17}$$

that is the difference between the actual optimal action and the action the *student* can derive given the teacher reference. Since the loss is computed over the whole trajectory, the overall loss is defined in batch with the following expression:

$$\mathcal{L} = \sum_{t=1}^{H} \mathcal{L}_t, \tag{6.18}$$

in which H represents the horizon of the prediction. The parameter $\boldsymbol{\theta}$ is hence updated as:

$$\theta' = \theta - \alpha \nabla \mathcal{L}, \tag{6.19}$$

with α that represents the learning rate and the update that is made through *gradient descent*. The teaching phase controls only one step of the *student*, i.e. only one action is performed with the input and correction of the student. Afterwards, the *student* turns back to be free to follow its original trajectory. The pseudo code of the procedure is shown in 6.

Algorithm 6 Teaching algorithm pseudo code

```
procedure TEACHING(traj, traj*)
    for t=1 to H do                          ▷ H is the time horizon
        Compute L_t = ||a*_t − f(s_t, s*_t)||^2
    end for
    Compute L = Σ_{t=1}^{H} L_t
    Update θ' = θ − α∇L
end procedure
```

The input $traj$ refers to the *student* trajectory, i.e. $s_0, a_0, s_1, a_1, \ldots,$ while $traj^*$ refers to the *teacher* trajectory, i.e. $s^*_0, a^*_0, s^*_1, a^*_1, \ldots, a^*_{H-1}, s^*_H$, that stops at horizon H.

6.3.1 Model Definition

Teacher is composed of a planning policy that is based on MCTS. This policy requires a transition model and a policy for basic action selection.

6.3.1.1 Transition Model

As the real dynamics of the function is not known, we use an approximated version inferred from a set of data. There are many options to choose the function approximator, and a reasonable option is the use of neural networks, see Section 3.2.1, since they have great capabilities of approximating well complex functions. NNs represent a base building block for the transition model we propose. According to the nature of the MDP, considering the transition dynamics, the transition model can be deterministic or stochastic. In this case, the vehicle dynamics are assumed to be deterministic. The proposed model predicts the delta δ from s_t and obtains the subsequent state s_{t+1} through it, starting from state s_t and performing action a_t, by summing up the delta, i.e. $s_{t+1} = s_t + \delta$:

$$\tau_\omega : \mathbb{R}^{\dim(\mathcal{S})} \times \mathbb{R}^{\dim(\mathcal{A})} \rightarrow \mathbb{R}^{\dim(\mathcal{S})}, \tag{6.20}$$

in which τ indicates the model, ω represents the set of model parameters.

The model is used to predict the trajectories at a certain horizon. If the model is ordinarily trained, it does not behaves in a satisfactory manner with repeated predictions. Therefore, a dedicated loss to multi-step optimization has been dedicated. In Section 6.3.1.2, one-step loss and multi-step loss are detailed.

6.3.1.2 Block Model

The state representation presents many features dependent to each other, i.e. the track relative state features (e.g. angle, track position) are all defined on the basis of the x, y coordinates of the vehicle. Since neural network models assume independence between outputs, the estimated quantities can lead to unrealistic state vector. Estimating all the features with a neural network would be a waste of computational resources, leading to consistency issues among estimated quantities. In Figure 6.15, a block schema describing the transition model is shown. A block structure ensures more consistency in the estimations. The state vector is partitioned into two sets: dynamic state and track state. In Figure 6.15, there is the input *state* to the network that refers to the dynamic partition of the state. The dynamic set comprises speed, acceleration and position change. The track set regards absolute position

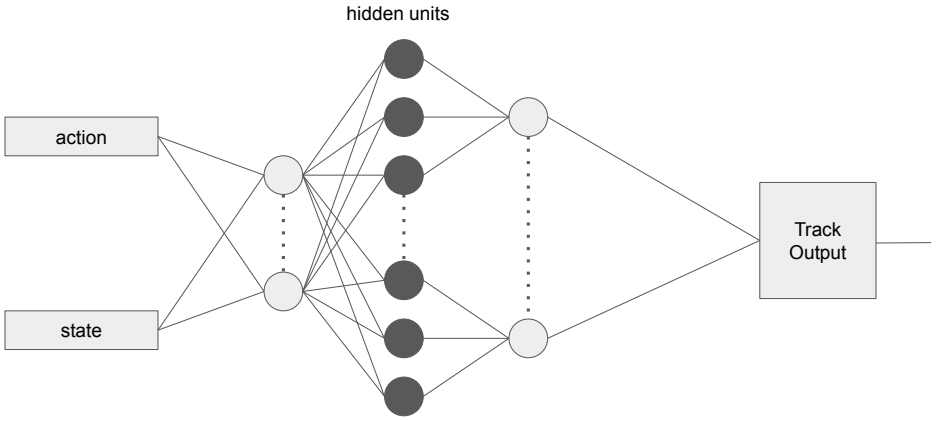


Figure 6.15: Block transition model. Input is composed of action and state. State refers to the dynamic partition of the state, to be computed by the neural network.

and all the features related to the track. The state is divided into two sub-blocks. The dynamic block estimates dynamic features starting from the dynamic set, i.e. actions. It is learnable as it is composed by neural networks. The track block analytically computes track features from the estimated dynamic features and is composed exclusively by fixed equations. According to this methodology, the vehicle dynamics depend exclusively on the dynamic state partition and the actions. They do not need other state partition, i.e. track set, to be estimated. In particular, the dynamic block estimates:

- forward pass;
- direction sin / cos;
- orientation sin / cos;
- speed x / y
- acceleration x / y

Absolute cartesian coordinates x, y are retrieved with the computation considering forward pass and direction sin / cos, that define the magnitude and the orientation of the delta to apply on the input x, y positions. The estimated x, y are then used to compute all the other track features. The transition model produces results that are consistent within state dimensions because the computation is systemic and starts from the same estimated values. The model is also data efficient, since neural networks consider only

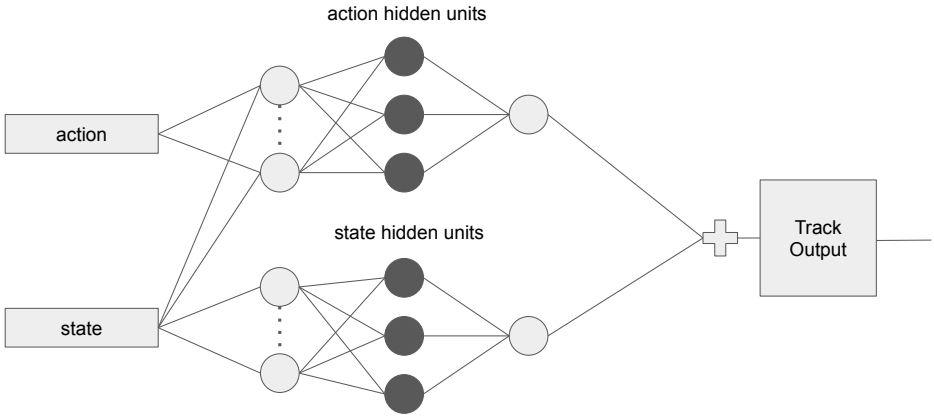


Figure 6.16: Inertial network. The dynamic output is the sum of the inertial network that takes the only the dynamic state as input and of the action network that has both the state and the action.

dynamic partition and not the whole state. This is useful to learn the dynamics independently of the position of the car on the track. Besides, the model is modular and interchangeable, since the dynamic block is related to the car setup, while the track setup is mostly related to the considered track the car is running on. The same dynamic block can be exploited in different tracks changing only the track block and the same vice versa, a different car, i.e. dynamic setup, can be used on the same track setup, changing only the dynamic block.

An extra option regards the use of an Inertial Model. The dynamic block of the block model can be improved considering that during race the car presents some physics properties. In particular, the car is moving with inertia that influences state even if no action is applied. In a classical MDP, when the *null action* is performed, i.e. no action is performed, the state does not change and the agent does not move if the environment allows for such a behavior. In this environment the car changes state and moves even if the *null* action is performed, i.e. 0 steer, 0 throttle, 0 brake. To this extent, the Inertial Model provides two networks, the inertial network, which has exclusively the dynamic state as input, and the action network, which takes as input both dynamic state and the actions performed by the agent. In Figure 6.16 the structure of Inertial Network integrated with the block model is presented.

6.3.1.3 Loss Functions

The loss function characterizes the learning type, since it allows to underline several aspects of the estimation problem according to its design. For the development of this transition model, two loss functions were designed. The first one is devoted to one-step prediction, the second one to multi-step prediction. The specific loss function we use in this work was the second type, see Section 6.3.2.

The one-step loss is the standard loss used in regression, i.e. the mean squared error between the real target and the predicted one. Considering $\tau(\mathbf{s}_n, \mathbf{a}_n)$ as the output of the dynamic block, we define $\mathbf{d}_n$ the delta sample, with a dataset of n from 0 to $N-1$. The loss is defined as:

$$\mathcal{L} = \frac{1}{N} \sum_{n=0}^{N} \|\mathbf{d}_n - \tau(\mathbf{s}_n, \mathbf{a}_n)\|^2. \tag{6.21}$$

The one-step loss function provides the prediction of the next state. The neural network of the dynamic block gives as output the delta of subsequent state with respect to the previous one.

6.3.2 Multi-Step Loss Function approach

The multi-step loss function is defined to increase the model performance in case of open loop estimation of H steps given a starting state and a sequence of actions. The procedure consists in applying iteratively the learnt one-step model, with the $h-1$ prediction as input for the h state. If the model is not accurate, this procedure can bring to divergence of the model during steps. In this context, it can happen that a fake input is produced by the model and therefore an input that is not possible to occur in such a domain is given back as input to the one-step model. The reasons for failure can be the inaccuracy of the model or the fact that the model obtains an inaccurate input during steps (except for the first). Considering higher horizons, the multi-step loss increases the robustness of the model. It allows to get more precise estimates for multiple calls of the model. Given a set $\mathcal{D} = \{\mathbf{s}_n^0, \mathbf{a}_n^0, \ldots, \mathbf{a}_n^{H-1}, \mathbf{s}_n^H\}_{n=0}^{N-1}$ of N trajectories of length H, the multi-step loss is defined as:

$$\mathcal{L} = \frac{1}{N} \sum_{n=0}^{N} \frac{1}{H} \sum_{h=1}^{H} \|\mathbf{s}_n^h - \hat{\mathbf{s}}_n^h\|^2, \tag{6.22}$$

where $\hat{\mathbf{s}}_n^h = \tau\left(\hat{\mathbf{s}}_n^{h-1}, \hat{\mathbf{a}}_n^{h-1}\right) + \hat{\mathbf{s}}_n^{h-1}$ is the predicted state in the sequence of the model recursive predictions. $\mathbf{s}_n^h$ represents the state of the $n-th$ trajectory at step h.

6.3.2.1 Monte Carlo Tree Search

For the *Teacher* trajectory generation, starting from a generic point of the track, decided as input, we propose to use MCTS, i.e. a planning-based method widely used in artificial intelligence and reinforcement learning, see Section 3.4.1. MCTS has been applied entirely offline, i.e. training of the transition model and of the policy is performed offline, because its objective is to have the trajectory. We use the behavioural cloning policy, see Section 6.3.2.2, as a variant of the policy. At each step, MCTS builds a search tree starting from the current starting state, taking several actions and computing the new states through the transition model. In the original MCTS procedure, at the end of the search, the best action is chosen, i.e. the one that from the root of the tree to the generated leaf produced the sequence with the highest performance. In this specific case, we propose to extend MCTS to return the whole best action sequence, i.e. starting from s_0, to return $a_0^*, s_1^*, a_1^*, ..., s_{H-1}^*, a_{H-1}^*, s_H^*$. When the search is terminated, MCTS returns the whole sequence that from the root to the generated leaf was the one with the higher performance.

The chosen action is then used in Algorithm 6, to update the *Student* $\boldsymbol{\theta}$ parameters, according to the loss defined over the horizon of the prediction of MCTS, in Equations (6.18), (6.19). The building procedure of MCTS consists in four different steps, as defined in Section 3.4.1 and in literature: selection, expansion, simulation, back-propagation. The description of the specific application of the algorithm to our context is now described.

Selection is the node selection step, that consists in the choice of the subsequent node to expand. To make MCTS traverse the tree until a leaf is reached, the definition method of which branch to select, i.e. which action, at each node, i.e. each state, is required. To this extent, we adopt an upper confidence bound approach for action selection, as in the UCT algorithm [42]. In addiction, there is an extra forced exploration. The chosen action is defined with the following formula:

$$a = \begin{cases} \arg\max_{a \in \mathcal{A}(s)} \left\{ Q(s,a) + \alpha \sqrt{\frac{\log N(s)}{N(s,a)}} \right\} & \text{with probability } 1-\epsilon, \\ \arg\min_{a \in \mathcal{A}(s)} N(s,a) & \text{with probability } \epsilon, \end{cases} \tag{6.23}$$

where $\epsilon \in [0,1]$ is the exploration term, and $\alpha \geq 0$ represents the learning rate. a is the action selected in state (i.e. node) s, with associated value $Q(s,a)$, where $N(s,a)$ counts how many times action a has been chosen in state s so far, $N(s)$ counts the whole number of visits to state s, and $\mathcal{A}(s)$ is the set of actions that is available for exploration in state s.

Expansion regards the node expansion as well as the choice of the candidate actions. MCTS selects one of the actions available in node s (i.e. when this node is selected), that was never chosen before. Then it computes the next state through the transition model. The resulting node is added to the tree as a child of the selected state (i.e. node s). Given that s is the selected node, MCTS selects an action $a \in \mathcal{A}(s)$, such that the action was never previously chosen and $N(s,a) = 0$, and computes the subsequent state as $s' = \tau(s,a)$, through the transition model τ. The new edge (s,a,s') is added to the tree. In this case, the creation of the new node s' requires the definition of the set $\mathcal{A}(s')$, i.e. the set of actions available in node s' for exploration. They are defined as *candidate actions*. Original MCTS predicts from scratch, while in this context the aim is to improve the base policy. Hence, our solution is that $\mathcal{A}(s')$ is built through the perturbation of the base policy in node s'. This is performed through the enumeration of different perturbations on the three action variables (i.e. steer, throttle and brake). The actions that are not reasonable in human driving style (i.e. brake and throttle simultaneously) are discarded.

Simulation starts from the new expanded node s' and performs H steps with the base policy and the trained transition model. The cumulative reward of the generated path is computed and is used to initialize the new node value. The procedure for the prediction performs a $h-$ step simulation with the base policy and the transition model according to horizons $h = 0, 1, ..., H$. The cumulative reward v of the resulting path is then added to a new dataset comprising tuples (s,h,v). A neural network is then fitted to predict v from (s,h) on the new dataset.

Back-Propagation is the final algorithm step, in which the output value of the simulation in the new node s' is propagated upwards across the tree. All values of nodes present in the path that produced that value (i.e. corresponding to leaf node s') are accordingly updated. This step is performed with Bellman backups. The value $Q(s,a)$ of node s, taking action $a \in \mathcal{A}(s)$ is updated as:

$$Q(s,a) = R(s,a) + V(s'), \tag{6.24}$$

where the value $V(s')$ represents the value of the next state, computed as:

$$V(s') = \max_{a' \in \mathcal{A}(s')} Q(s',a'). \tag{6.25}$$

All nodes at depth H are labelled as terminal, since the transition model does not allow to build an accurate tree with arbitrary depth. An out of track position of the car is considered terminal as well and the corresponding node is never expanded. After MCTS terminates, i.e. running after a given

number of iterations is performed, the algorithm returns directly the branch that lead to the highest value, not just the best action to perform at the next step. This approach reflects the need to return a complete trajectory.

6.3.2.2 Behavioural Cloning

The policy used by Monte Carlo Tree Search is initially trained with the *behavioral cloning* loss, defined as:

$$\mathcal{L}^{BC} = \frac{1}{N} \sum_{n=0}^{N} \|\mathbf{a}_n - \pi_{\boldsymbol{\theta}^\pi(\mathbf{s}_n)}\|^2, \tag{6.26}$$

in which $\mathbf{a}_n$ is the target action and $\mathbf{s}_n$ represents the input state. The dataset to generate the behavioral cloning step is a collection of human expert demonstrations, i.e. reference trajectory $\mathcal{D}^T$.

Chapter 7

Experiments

In this Section, the experimental results regarding the application of the approach described in this thesis are proposed. In Section 7.1, the Student is optimized through PGPE algorithm. In Section 7.2, the focus is on the teaching development through MCTS planning. In particular, the Section proceeds by defining the most appropriate transition model, in sub section 7.2.1, and the behavioural cloning policy on the basis of MCTS approach, in sub section 7.2.2. Finally, the planning approach of MCTS is integrated with the teaching algorithm, in sub section 7.2.3, both in the offline and online case.

7.1 Student Optimization with PGPE

The first experimental approach refers to the student optimization phase. The *student* has been defined in Section 5.2 as an agent that tends to a sub-optimal policy. In this Section, the optimization of the parameters defined in Section 6.2, that characterize the *student* policy, are optimized with PGPE algorithm. The optimization approach is divided in several steps, since this procedure enlighten the direct impact of each adopted optimization in the driving performance, i.e. the relevance. Criteria for the steps definition have been selected on the basis of the student driving analysis, i.e. on the identification of the most relevant features in one lap (i.e. speed). Then, we focused on the track, since the boost intensity and the position with respect to the center of the track are core technicalities of the driving style of the student. Initialization values have not been randomly chosen, they have been chosen from a human, as a consequence of circuit and performance analysis. First category of experiments, in Section 7.1.1 refers to the optimization of the speed of reference v^*, see Table 6.1. Then, in Section 7.1.2, the

Parameter θ : v^*	**Mean μ_0**	**Mean μ_f**	**SD σ_0**	**SD σ_f**
First curve	-	-	-	-
Mini chicane	-	-	-	-
First 90° curve	100	127	0.15	0.01
Second 90° curve	80	83	0.15	0.01
Chicane	80	120	0.15	0.01
Parabolic curve (p.1)	100	130	0.15	0.01
Parabolic curve (p.2)	80	300	0.15	0.09

Table 7.2: Experiment Section 7.1.1. Initialization and Final values for θ parameter. Values μ_0 and σ_0, are the initial mean and standard deviation that define the Gaussian distribution exploited by PGPE algorithm. Both the variables, for each parameter, are learnt by the algorithm. The final values are denoted with μ_f and σ_f.

boundaries of the parameters are learnt and optimized. Finally, in the last category of experiments regarding the *student*, in Section 7.1.3, the boost position is optimized, i.e. μ of the Gaussian distribution that defines the boost. In Table 7.1 the values of mean μ_i and σ_i, for each section i are shown. These values are used for the first two experiments. They are learnt in Section 7.1.3 Both values are in Meters.

Section	**Mean μ**	**Standard Deviation σ**
First curve	1000	100000
Mini chicane	1850	3000
First 90° curve	2400	3000
Second 90° curve	2700	6000
Chicane	3800	50000
Parabolic curve	5100	50000

Table 7.1: Values of Gaussian boost in curves

7.1.1 Speed Optimization

The first experimental test performed on *Student Controller* consists in a punctual evaluation of the values of the reference speed for the critical sectors of the track. In particular, according to the Formulation provided in Table 6.1, the component that is learnt in this phase is v^*, i.e. the *Reference Speed*, that is used to define $\varepsilon_v = v^* - v$, that models brake equation 6.7 and throttle equation 6.8. In Table 7.2 initialization values and final ones, obtained by the execution of the algorithm are shown. Sectors considered in this test are $\Psi = \{$First 90° curve, Second 90° curve, Chicane, Parabolic

curve (p.1), Parabolic curve (p.2)}. The sectors {First curve, Mini chicane} have not been considered in this training because physiognomy of the track doesn not require a reduction in speed. In this experiment 5 sectors have been identified that correspond to the main curves in the track.

- First curve of 90°
- Second curve of 90°
- Chicane
- Parabolic curve (p.1)
- Parabolic curve (p.2)

The basic value for reference speed that has been defined in linear traits of the track is suited also for these sectors. This test has been executed with *step size*=0.05. The number of tested parameters is 5, i.e. 5 reference speeds $v*$. Overall number of tested parameters is 10, considering also the standard deviation of the learnt parameters. An important note is that the reference speed for the last sector, *Parabolic curve*, has been divided into two parts to model the fact that the driver must go into the curve with a reduced speed, while it can push accelerator and increase considerably its speed while going outside the curve. Sectors division with legend is shown in Figure 7.1. The white colour on the track means the area is not contained in any sector. Coloured traits represent a specific sector, according to the legend.

Results regarding performance indices are shown in Figure 7.3. Results are expressed in terms of reward and discounted reward. We can notice that there is a great improvement as average return goes from -14034.16 to -11946.40. As we are working on $100Hz$ frequency, this means that the overall gain in terms of time equivalence is around 30 *seconds* of improvement. In Figure 7.4 other statistics and relevant information about experiment data are provided. In particular, the following performance measures are shown:

- Average episode length
- Variance of return J
- 2-norm of the gradient
- ∞-norm of the gradient
- Distance from the start

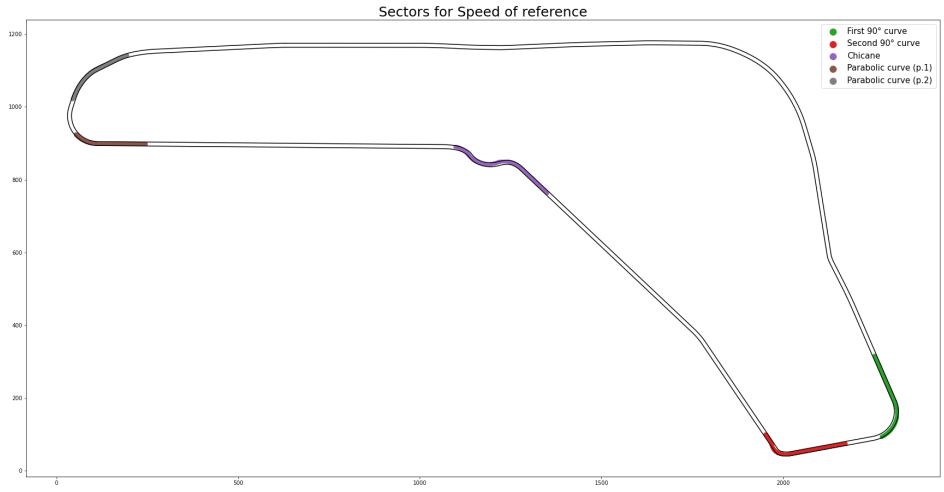


Figure 7.1: Experiment Section 7.1.1. Sectors division. Coloured traits represent specific sectors (see legend), while white areas correspond to areas not in sectors.

- Average episode quantile 25
- Average episode quantile 50
- Average episode quantile 75

7.1.2 Track Position Optimization

The second experimental test performed on *Student Controller* consists in a punctual evaluation of the boundaries of the sectors in which *trackPos* variable value is defined. This second approach considers a sectors' division with a finer granularity, since the driving style is very sensitive to changes in position along the track and needs a careful analysis. In this case PGPE works with the limits of the sectors in which *trackPos* is changing. Considering the Formulation provided in Table 6.1, the component that is corollary learnt in this test is $Tpos*$, i.e. the reference *trackPos* that is used to define $\varepsilon_{Tpos} = Tpos^* - Tpos$, that models steer equation 6.13. *Angle* component is not considered since its variation is negligible with respect to the influence of *trackPos*, and tuning also this parameter would increase convergence times. Considering only *trackPos*, convergence times speed up, and we apply a sort of regularization. In Table 7.4 initialization values and final ones, obtained by the execution of the algorithm are shown. In Table 7.3, sectors are renamed in order to allow a clearer explanation of the test results (i.e. only for this experimental Section). Sector 1, as shown in Table 7.3, starts when

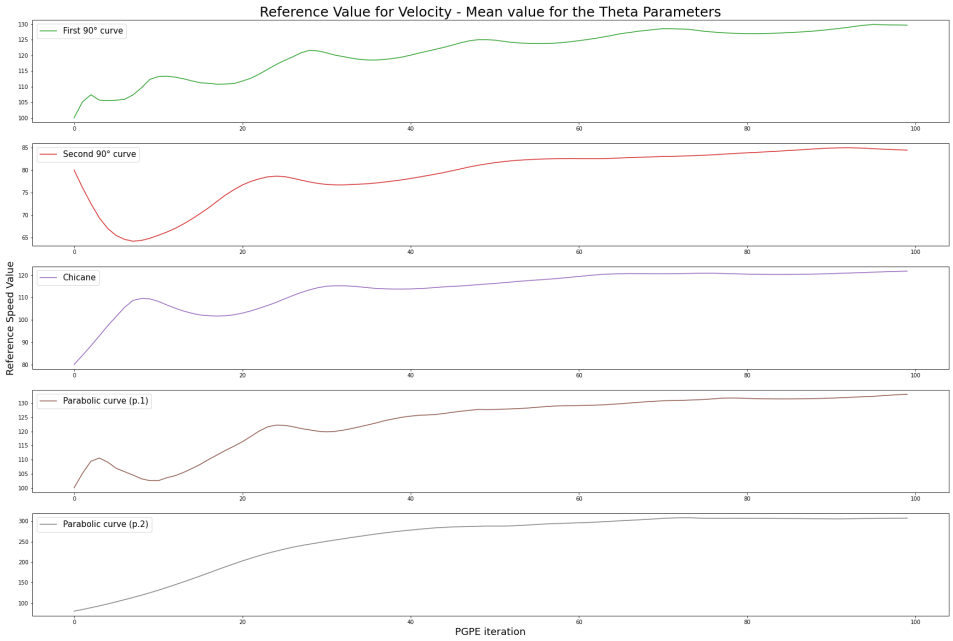


Figure 7.2: Experiment Section 7.1.1. Mean of the 5 trained variables, see Table 7.2, μ_0 and μ_f. The plots shows how the values go from the initial value to the convergence value. On the x-axis, the number of iterations of PGPE.

distance from the start is equal to 2200 and terminates when the distance is equal to 2450. The example of functioning of *trackPos* reference is shown in Figure 7.5. This picture refers to Sector 1, i.e. First 90° curve, and shows that the sector is partitioned into 3 parts. The points that fragment the sector are the parameters that PGPE learns during this experiment. The same holds for the other sectors. The picture shows that there are two points that divide the reference line of sector 1, i.e. 2325 and 2400. This expression for *trackPos* has been defined on the basis of the behaviour of this variable in the human expert demonstrations. It is evident that *trackPos* effective value has a slow reaction time with respect to its reference value. Green line describes how the variable changes in real execution when following the blue line reference. This may happen because of the closed form equations that describe the policy and the dynamics of the vehicle. In Table 7.4 the initial values of the points that determine the sectors are shown. In the table, both the extremes and the internal points for fragmentation are shown for completeness. As already stated, only the internal points are learnt, while the extremes, that identify a critical area of the track are kept constant and fixed. Hence, the overall number of θ parameters is 9. Two variables for sector 1, two variables for sector 2, four variables for sector 3, one variable

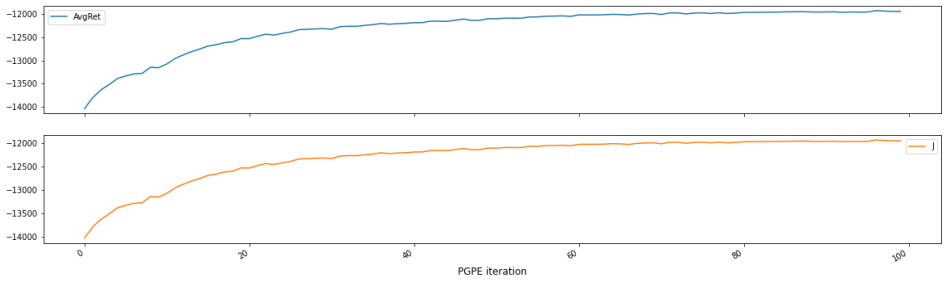


Figure 7.3: Section 7.1.1. Experiment 1. Reward and discounted reward as the number of iterations of PGPE increases. On the y-axis, the variable value. On the x-axis, PGPE iteration.

New name	Original name	Fixed sector borders
Sector 1	First 90° curve	2200, 2450
Sector 2	Second 90° curve	2550, 2800
Sector 3	Chicane	3700, 4000
Sector 4	Parabolic curve (p.1)	4850, 5050
Sector 5	Parabolic curve (p.2)	5150, 5350

Table 7.3: Experiment Section 7.1.2. Sectors renomination to allow a simpler description in the following chapter. In the last column there are the borders of the sectors (the two extremes) that are kept fixed. PGPE focuses on the points that refer to the internal segmentation of sectors. The borders are defined on the basis of the distance from the start of the car.

for sector 4. The parameter is named P.x.y, where x defines the sector, i.e. $x \in 1, 2, 3, 4, 5$, and y is the index within the same sector. P.1.1 refers to the first point that fragments sector 1, P.1.2 refers to the second point that fragments the sector, as shown in Figure 6.4. This test has been executed with *step size*=0.001. In Table 7.5, the θ parameters used by PGPE algorithm are detailed. The standard deviation is not shown in the table since its variation is minimal and negligible. The important result is that this value tends to converge, so to reduce. The means of the points that fragment the sectors change substantially in some cases. Notice that even if the value of standard deviation is not shown in Table 7.5, the overall number of learnt θ parameters is 18, since PGPE learns both the mean and the standard deviation of the 9 parameters. In Section 7.1.2 the progress of the learnt parameters during the execution of the algorithm is shown.

The overall results are shown in Figures 7.6, 7.7, 7.8, 7.9 and 7.10. Each Figure represents a sector, with the legend described in Table 7.3. For example, Figure 7.6 refers to Sector 1, i.e. First 90° curve, and is composed

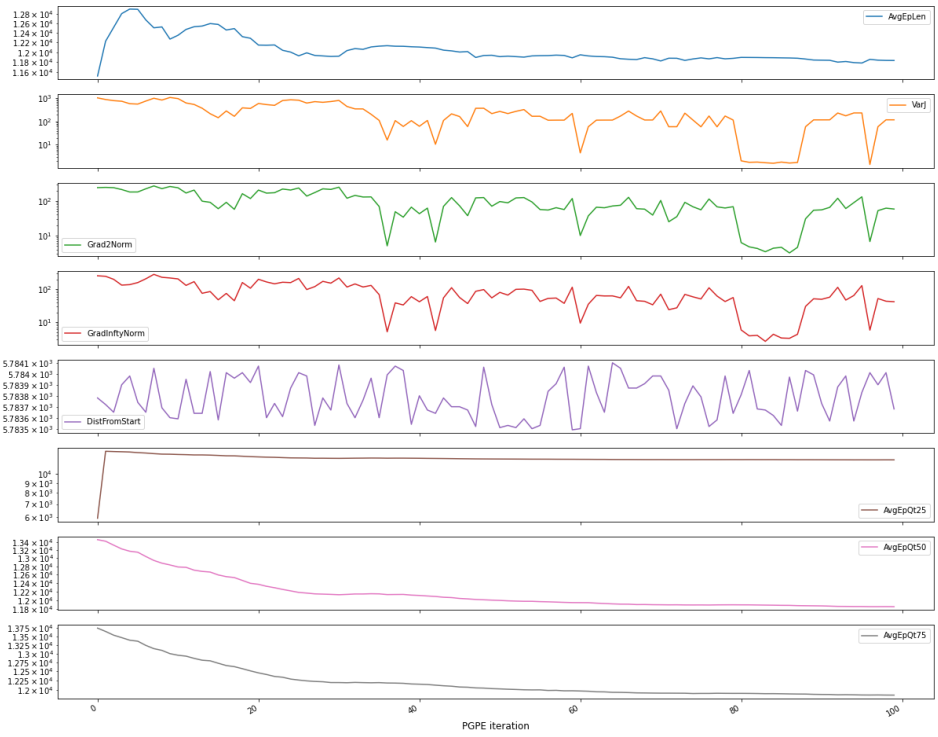


Figure 7.4: Section 7.1.1. Experiment 1. Stats and values for experiment analysis. On the y-axis, the variable value. On the x-axis, PGPE iteration. In the order, average episode length, variance of return J, 2-norm of the gradient, ∞-norm of the gradient, distance from the start, average episode quantile 25, average episode quantile 50, average episode quantile 75.

of two plots. The upper plot refers to the settings before the running of PGPE algorithm. The blue line represents the reference expression for the *trackPos* variable, while the green line represents the effective value of the variable when following its reference value. The second plot, i.e. the one below, refers to the obtained results after PGPE algorithm execution. The light blue line refers to the updated function of *trackPos* variable. Here we can see that the points that delimit the sectors inter-division have moved. The light green line refers to the effective value of the variable *trackPos* while following the reference after the execution of PGPE. The same concept can be seen in Figures 7.7, 7.8, 7.9 and 7.10.

Figure 7.10 is shown for completeness, even if no training of PGPE on variables is performed in sector 5, i.e. Parabolic curve (p.2). Sectors 3 and 4, i.e. Chicane and Parabolic curve (p.1), are very interesting in this PGPE experimental execution, since the algorithm does not limit to adjust the

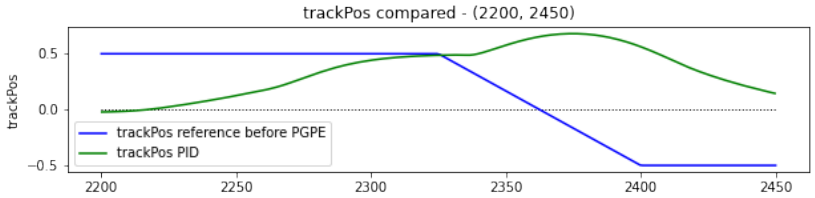


Figure 7.5: Experiment Section 7.1.2. Comparison between the reference value of trackPos (blue line) and effective value of the student policy when following that reference (green line). The reference equations for trackPos are defined in the following way. The first sub-sector, from the init value, till P.1.1 is a constant trait. The second sub-sector, from P.1.1 to P.1.2 is a monotonically decreasing line. The last sub-sector, from P.1.2 to the end, is a constant trait. On the x-axis, the distance from the start.

trackPos equation shape, as in Figures 7.12 and 7.13, in which the internal points are simply moved of a controlled quantity (with reference to sectors 1, i.e. First curve of 90°, and 2, i.e. Second curve of 90°). In the case of Figures 7.14 and 7.15 it is evident that the shape of the expression that defines the *trackPos* variable of reference is dramatically changed, since some sectors are even removed. For example, in case of sector 3, i.e. chicane, we pass from 5 sectors to 2 sectors (see Figures 7.8 and 7.14). In case of sector 4, i.e. Parabolic curve (p.1), we pass from 2 sectors two 1 single sector (see Figures 7.9 and 7.15).

The same test is repeated with the new obtained values for sectors limits, so with updated expressions that determine the *trackPos* of reference during car driving.

The analysis of the reference for *trackPos* brought to a reduction in the number of steps required to complete one lap of 0.20 seconds. In Figure 7.11, the improvement that is obtained through PGPE and the iterative application of the algorithm is shown. On the x-axis, there is the sequence of experiments that is carried out on the *student policy*. This sequence includes the original policy, the improved policy with the optimization of speed described in Section 7.1.1, the optimization of reference sectors for *trackPos* variable described in this section, and the optimization following directly the expert trajectory described in Section 7.1.3. On the y-axis, the number of steps that are required to complete one lap. Since we work on $100Hz$ frequency, 11770 steps are equivalent to 117.7 s. In the plot, there is the starting value identified with the first experiment. The asymptotic value refers to the best value that has been obtained by the *student* policy, that is not the global optimum. The obtained performance is the best with respect to its capabilities, and the delta of reward for improving the performance is

	Borders	**Mean μ_0**	**Is learnable?**
Sector 1	Init	2200	No, fixed value
	P.1.1	2325	Yes
	P.1.2	2400	Yes
	End	2450	No, fixed value
Sector 2	Init	2550	No, fixed value
	P.2.1	2700	Yes
	P.2.2	2725	Yes
	End	2800	No, fixed value
Sector 3	Init	3700	No, fixed value
	P.3.1	3775	Yes
	P.3.2	3825	Yes
	P.3.3	3900	Yes
	P.3.4	3950	Yes
	End	4000	No, fixed value
Sector 4	Init	4850	No, fixed value
	P.4.1	4975	Yes
	End	5050	No, fixed value
Sector 5	Init	5150	No, fixed value
	End	5350	No, fixed value

Table 7.4: Experiment Section 7.1.2. Initialization values for the sectors' definition variables. The extremes of the sectors are not conisdered θ parameters. They are kept fixed since they determine a critical area of the track (i.e. curve, or chicane). The points that define the internal fragmentation will be considered θ parameters for the execution of PGPE.

the quantity that is learnt through the teacher.

The evolution of the mean of the internal borders that define the sectors fragmentation during PGPE execution, synthesized in Table 7.5, is graphically shown in Figures 7.12, 7.13, 7.14 and 7.15. In these figures, the dashed gray borders represent the fixed extremes of a single Sector. The coloured line represents the sequence of values that PGPE learns for the internal fragmentation points during the algorithm execution. It is possible to see that the internal fragmentation points' values, i.e. the mean, asymptotically converge. The legend for sectors definition is provided in Table 7.3.

In particular in Figures 7.14 and 7.15, it is evident that some sectors are practically removed by PGPE algorithm.

Parameter θ : $P.\boldsymbol{x}.\boldsymbol{y}$	**Mean μ_0**	**Mean μ_f**
P.1.1	2325	2282.5
P.1.2	2400	2402
P.2.1	2700	2593.3
P.2.2	2725	2648
P.3.1	3775	3628
P.3.2	3825	3752.4
P.3.3	3900	4028
P.3.4	3950	3815.3
P.4.1	4975	4790.8

Table 7.5: Experiment Section 7.1.2. Initialization and Final values for θ parameter. Values μ_0 is the initial mean that defines the Gaussian distribution exploited by PGPE algorithm. The standard deviation is not shown. It is equal to 0.018 for all the parameters and is learnt (it negligibly decreases as expected). Both the variables, for each parameter, are learnt by the algorithm. The final value of the mean is denoted with μ_f.

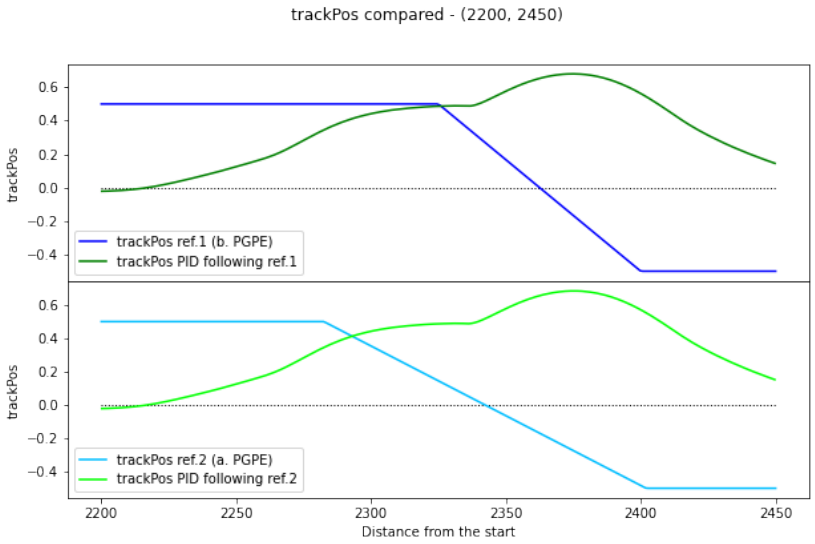


Figure 7.6: Experiment Section 7.1.2. Sector 1. Comparison between the reference value of trackPos before and after PGPE execution. Blue line and light blue line refer to the reference before and after the algorithm execution. Green line and light green line refer to the effective behaviour of trackPos when following the proper reference.

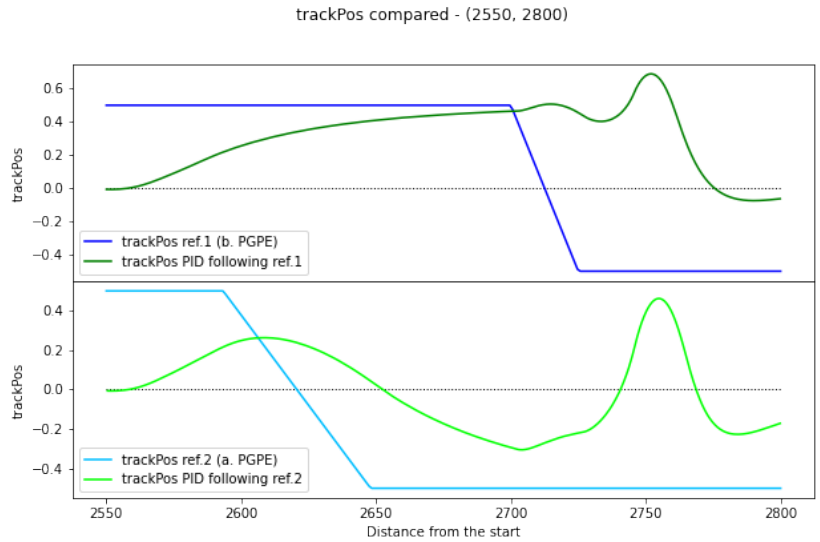


Figure 7.7: Experiment Section 7.1.2. Sector 2. Comparison between the reference value of trackPos before and after PGPE execution. Blue line and light blue line refer to the reference before and after the algorithm execution. Green line and light green line refer to the effective behaviour of trackPos when following the proper reference.

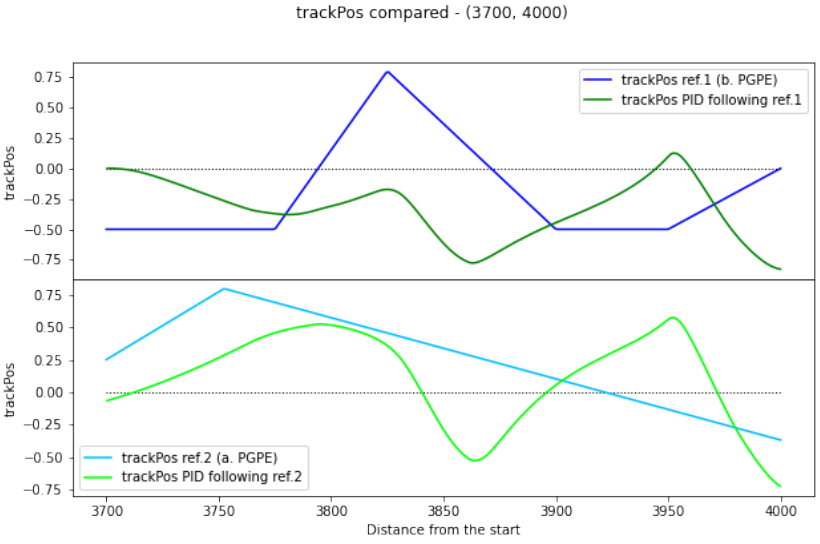


Figure 7.8: Experiment Section 7.1.2. Sector 3. Comparison between the reference value of trackPos before and after PGPE execution. Blue line and light blue line refer to the reference before and after the algorithm execution. Green line and light green line refer to the effective behaviour of trackPos when following the proper reference.

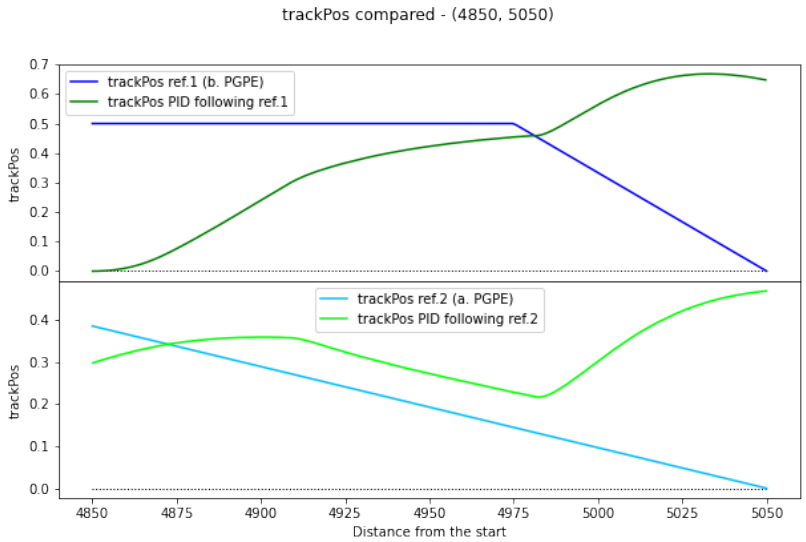


Figure 7.9: Experiment Section 7.1.2. Sector 4. Comparison between the reference value of trackPos before and after PGPE execution. Blue line and light blue line refer to the reference before and after the algorithm execution. Green line and light green line refer to the effective behaviour of trackPos when following the proper reference.

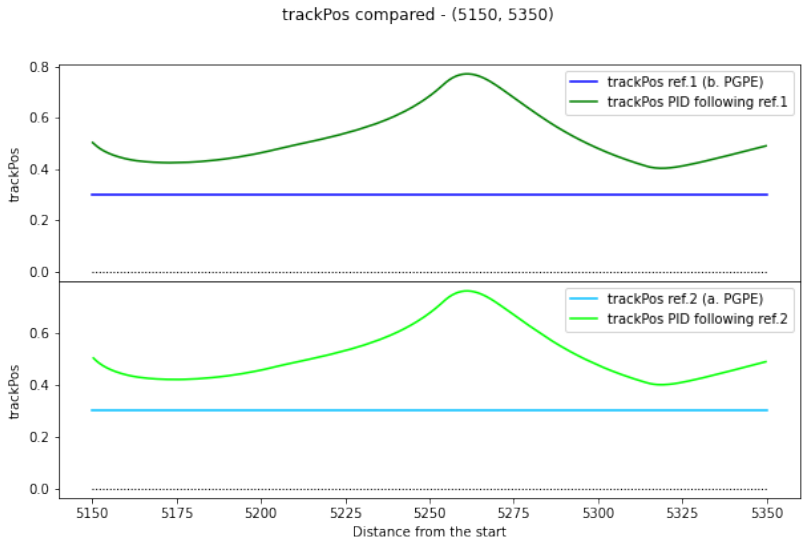


Figure 7.10: Experiment Section 7.1.2. Sector 5. Comparison between the reference value of trackPos before and after PGPE execution. Blue line and light blue line refer to the reference before and after the algorithm execution. Green line and light green line refer to the effective behaviour of trackPos when following the proper reference.

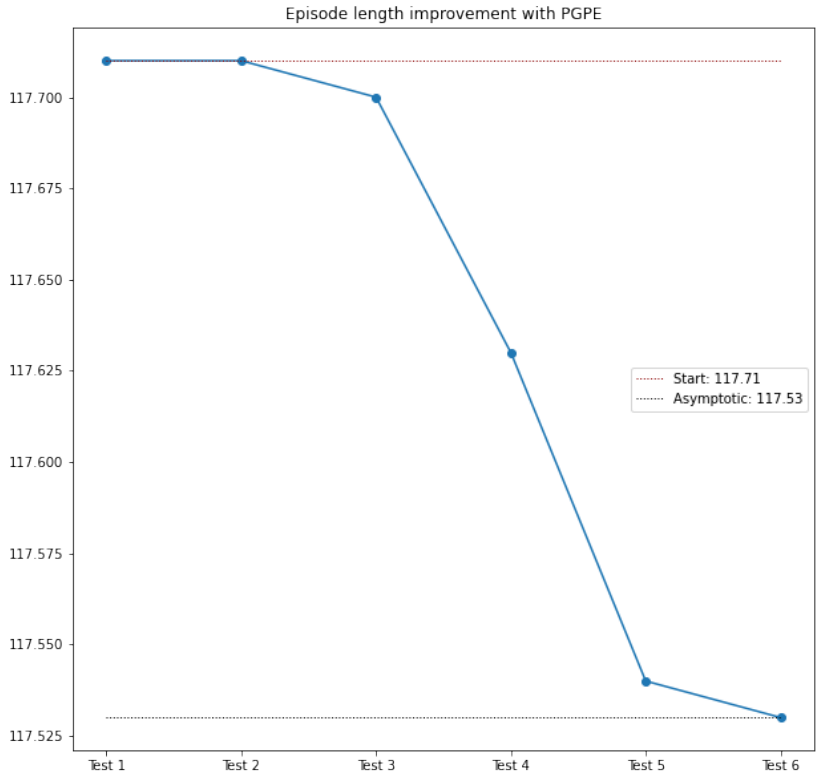


Figure 7.11: Improvement considering $N.steps/100$, since we work on $100Hz$ frequency. The x-axis denotes the index of experiments carried out on the student policy, that refers to more applications of the experiments described in Sections 7.1.1, 7.1.2, 7.1.3.

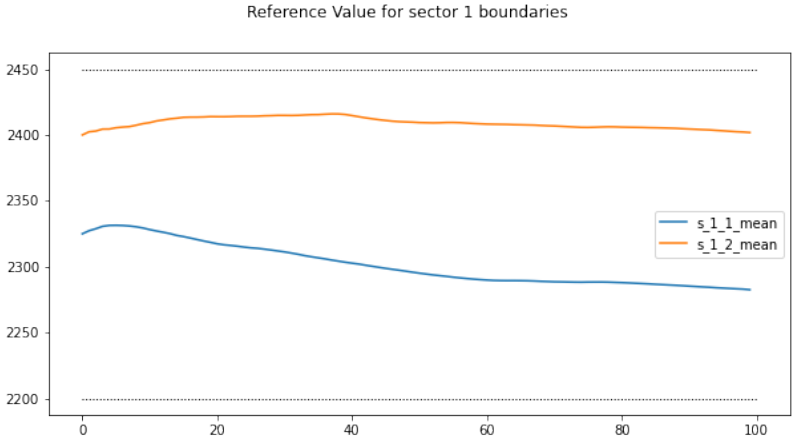


Figure 7.12: Section 7.1.2. Experiment 2. Sector 1. Dashed lines refer to the fixed initial point and termination point of the sector. The plotted lines refer to the trend of the internal fragmentation points, S.1.1 and S.1.2 during PGPE execution. On the x-axis, PGPE execution time, on the y-axis the distance from the start of the point. The blue line refers to the first point that fragments sector 1, i.e. P.1.1. The orange line refers to the second fragmentation point, i.e. P.1.2.

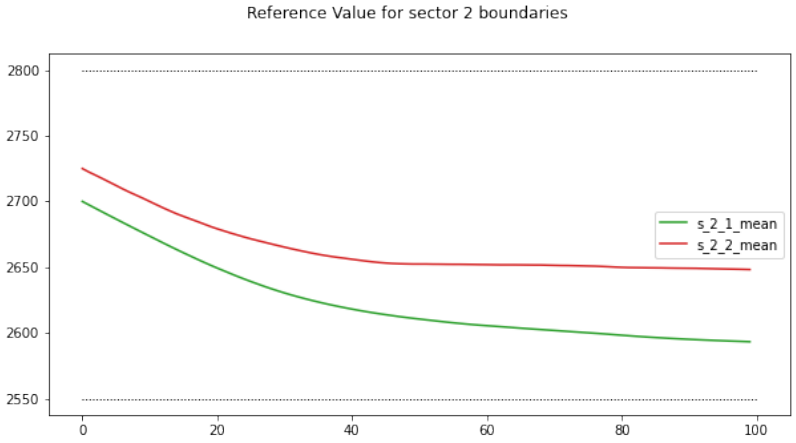


Figure 7.13: Section 7.1.2. Experiment 2. Sector 2. Dashed lines refer to the fixed initial point and termination point of the sector. The plotted lines refer to the trend of the internal fragmentation points, S.2.1 and S.2.2 during PGPE execution. On the x-axis, PGPE execution time, on the y-axis the distance from the start of the point.

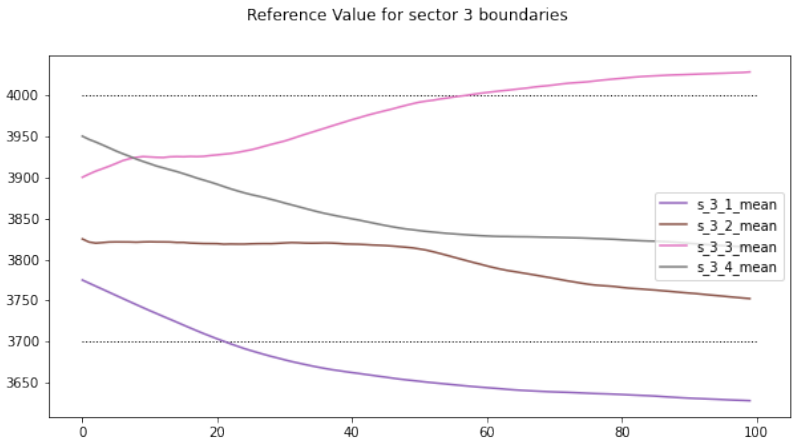


Figure 7.14: Section 7.1.2. Experiment 2. Sector 3. Dashed lines refer to the fixed initial point and termination point of the sector. The plotted lines refer to the trend of the internal fragmentation points, S.3.1, S.3.2, S.3.3 and S.3.4 during PGPE execution. On the x-axis, PGPE execution time, on the y-axis the distance from the start of the point.

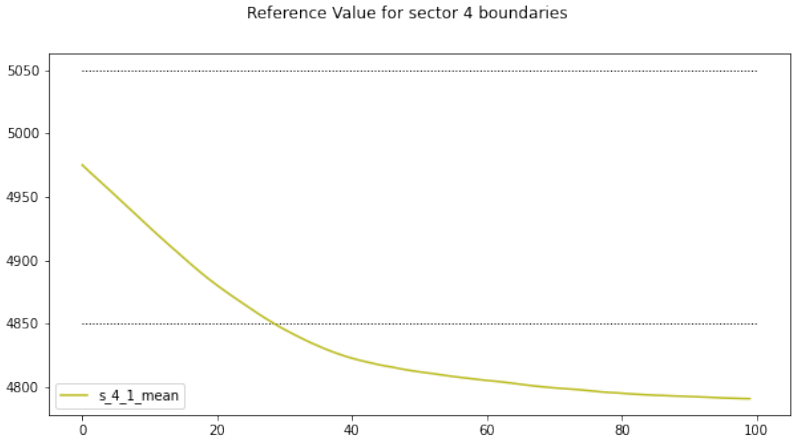


Figure 7.15: Section 7.1.2. Experiment 2. Sector 4. Dashed lines refer to the fixed initial point and termination point of the sector. The plotted lines refer to the trend of the internal fragmentation points, S.4.1 during PGPE execution. On the x-axis, PGPE execution time, on the y-axis the distance from the start of the point.

Parameter θ : d	**Mean μ_0**	**Mean μ_f**
First curve	1000	1006.28
Mini chicane	1850	1908.88
First 90° curve	2400	2430.42
Second 90° curve	2600	2622.41
Chicane	3800	3803.75
Parabolic curve	5100	5062.7

Table 7.6: Experiment Section 7.1.3. Initialization and Final values for θ parameter. Values μ_0 is the initial mean that defines the Gaussian distribution exploited by PGPE algorithm. The standard deviation is not shown. It is equal to 1 for all the parameters and is learnt (it negligibly decreases as expected). Both the variables, for each parameter, are learnt by the algorithm. The final value of the mean is denoted with μ_f.

7.1.3 Reference sectors Optimization

The third experimental test performed on *Student Controller* consists in a punctual evaluation of the values in which the boost for the steer is applied, see Equation (6.16). In particular, according to the Formulation provided in Table 7.1, the component that is learnt in this test is the column of the mean μ, that refers to the center of application of the boost before a curve or a critical track trait occurs. This test learns also the k coefficient of the boost intensity, see Equation (6.16), to analyze the intensity of the boost, as well as the position. In Table 7.6 initialization values and final ones, obtained by the execution of the algorithm are shown.

This test has been executed with *step size*=0.001. Sectors considered in this test are $\Psi = \{$First curve, Mini chicane, First 90° curve, Second 90° curve, Chicane, Parabolic curve$\}$. Overall number of parameters tested is 7, it includes both the boost intensity k and the mean of the boost μ_j with $j \in \Psi$. Parabolic curve is here considered as a single sector, since the curve is smooth and one boost is enough to correctly drive. PGPE algorithm has been executed in this context with the expert *teacher* policy as reference. This implies that next action is provided by the set of expert human demonstrations, not directly by the *student* policy. This test shows that the parameters of the *student* are modified and tend to the optimal result, even if they keep to be sub-optimal. They are not able to follow exactly the reference trajectory, there is a delta in the performance as expected. In Table 7.6, it is shown that the parameters θ values move significantly. This allows to change the application point of the boost and results in a more effective driving.

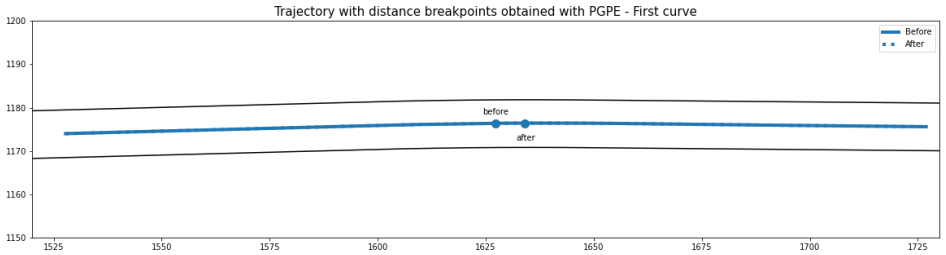


Figure 7.16: Experiment Section 7.1.3. First curve. Trajectory of the student policy before the execution of PGPE and after the algorithm execution. The boost application point has been learnt.

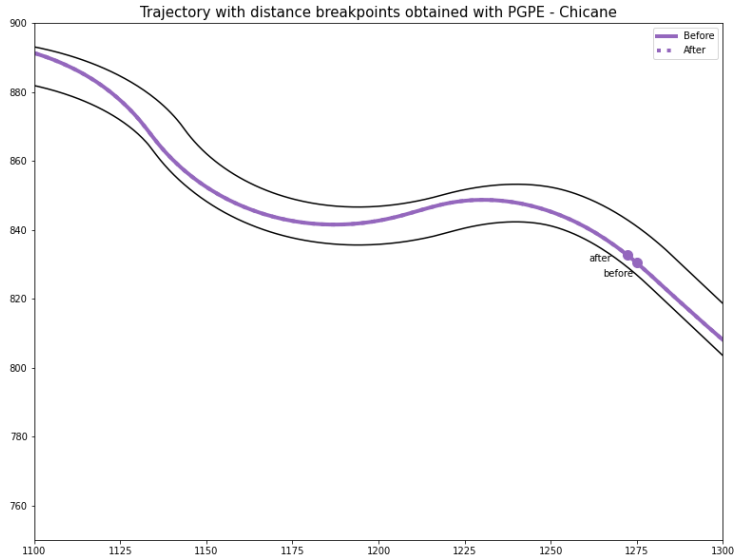


Figure 7.20: Experiment Section 7.1.3. Chicane. Trajectory of the student policy before the execution of PGPE and after the algorithm execution. The boost application point has been learnt.

In Figures 7.16, 7.17, 7.18, 7.19, 7.20 and 7.21 there is the representation of the specific sectors in Ψ of the parameters θ that have been learnt by the algorithm. The dashed line represents the behaviour of the *student* following the learnt parameters with PGPE.

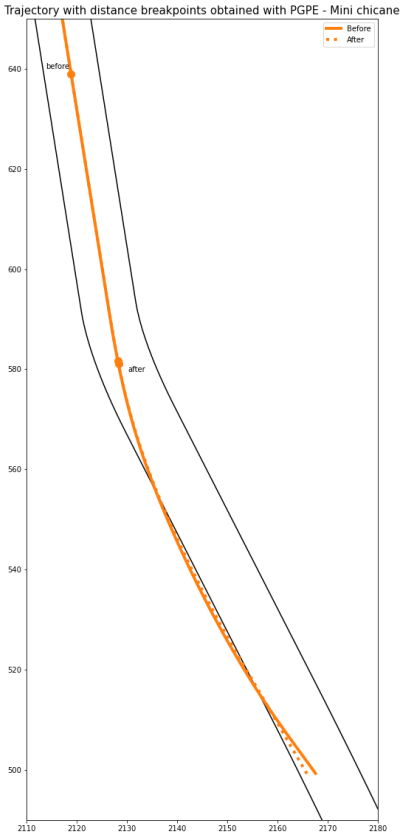


Figure 7.17: Experiment Section 7.1.3. Mini chicane. Trajectory of the student policy before the execution of PGPE and after the algorithm execution. The boost application point has been learnt.

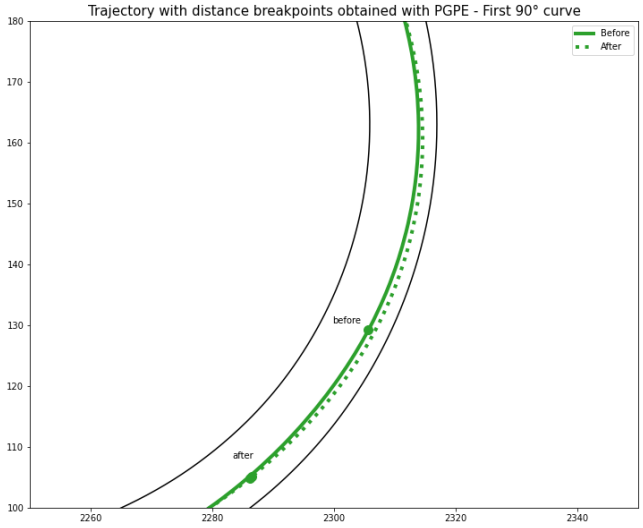


Figure 7.18: Experiment Section 7.1.3. First $90°$ curve. Trajectory of the student policy before the execution of PGPE and after the algorithm execution. The boost application point has been learnt.

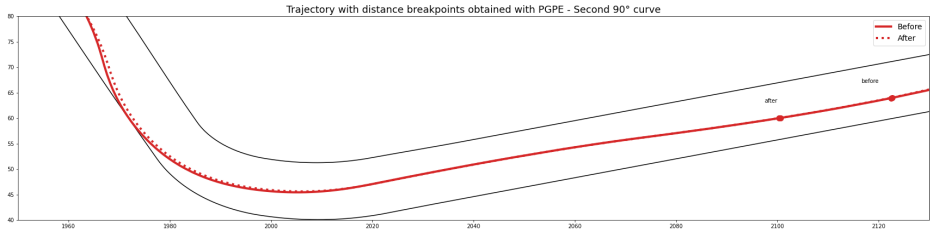


Figure 7.19: Experiment Section 7.1.3. Second $90°$ curve. Trajectory of the student policy before the execution of PGPE and after the algorithm execution. The boost application point has been learnt.

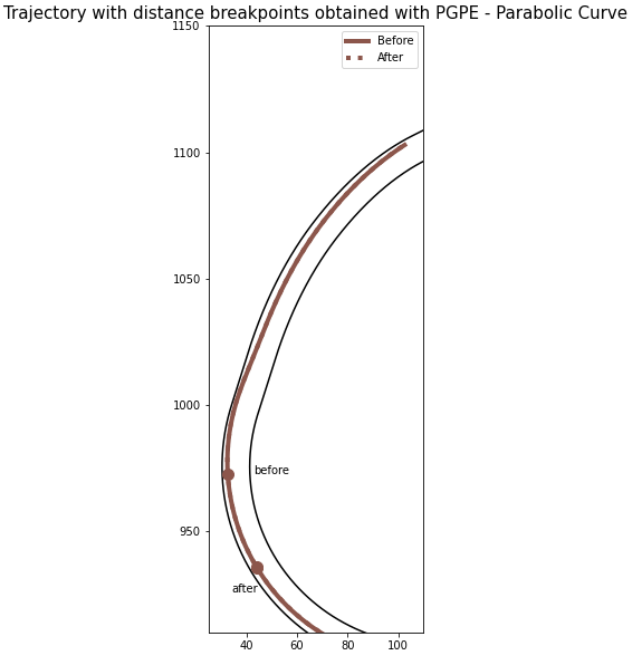


Figure 7.21: Experiment Section 7.1.3. Parabolic curve. Trajectory of the student policy before the execution of PGPE and after the algorithm execution. The boost application point has been learnt.

7.2 Teaching

In this section the core *teaching* process is applied. The teacher phase has been defined in Section 5.3 as the process to obtain an optimal policy, given a sub-optimal policy. To this extent, the first component that is required is a trained transition model, in order to predict the trajectory over a given horizon.

7.2.1 Loss Multi Step for Transition Model

As described in Section 6.3.1.2, the transition model is a neural network. It has many hyperparameters. The most important are network architecture, activation function, learning rate, batch dimension, regularization and horizon of the multi-step loss. Several networks' topologies have been tested, in order to find the best possible configuration for our task. In the end, the better results have been obtained with a transition model built with a neural network with two hidden layers, each of 60 neurons. The training horizon of the transition model is 20. The inertial network is used, since the obtained performances in terms of Root Mean Square Error (RMSE) were better. RMSE is used as performance index, i.e. the standard deviation of residuals (prediction errors). After grid search tuning the optimal values are the following:

- direction_sin
- direction_cos
- speed_x
- speed_y
- acceleration_x
- acceleration_y
- yaw_sin
- yaw_cos
- x
- y
- trackPos
- angle

- distFromStart

with *yaw* that represents the absolute angle of the car in the 2D space. In Figure 7.22 the value of RMSE for every performance index of the chosen transition model for prediction is shown. In Figure 7.23, the trend of the RMSE of the model is shown. In Figure 7.24, the prediction over horizon 20 of the transition model used in this thesis is shown. In Figure 7.25, a validation lap performed through the fitted transition model is shown.

The predicted trajectory is tested both with an horizon 20 and with slightly higher horizons, to estimate the model quality for larger horizon.

Details of the trained transition model along the track are shown in Figures 7.26, 7.27, 7.28, 7.29 and 7.30.

7.2.2 Behavioural Cloning Policy

As described in Section 6.3.2.2, the Behavioural Cloning policy adopted by MCTS is trained with the loss identified in Equation (6.26). The dataset that has been used to generate the behavioral cloning step is a collection of 20 human expert demonstrations, i.e. reference trajectories. The adopted frequency for the sampling is $100Hz$, as for the other components of this work. The Behavioural Cloning policy has been generated through the use of a NN. Different networks' topologies have been tested to make a complete and extensive analysis. Configuration of networks adopted for the policy differ for several features, e.g. the number of hidden layers, i.e. 2 hidden layers of 300 neurons each or 3 hidden layers of 50 neurons each, presence of regularizer (for actions steer, brake and throttle), activation of the dropout flag. We kept unchanged the ReLU activation function, the number of epochs, i.e. 1000, validation split percentage, i.e. 0.3. *Adam* optimizer has been adopted. Early-stopping flag is sometimes activated as well. In Figure 7.31, the trajectory derived from the tested network topologies is shown. 9 network topologies has been tested, with indices from 0 to 8. As we can see from the Figure, the mere Behavioural Cloning policy is not enough to perform a lap of adequate quality, i.e. not even complete. This result is due to the absence of a complete knowledge. By imitating a limited set of human demonstrations, the resulting policy may lead to unexplored regions of the space, in which the control of the vehicle and of the adequate action to perform is lost. In our case, the set of human demonstrations is tiny. Nevertheless, even with a larger demonstration set, the resulting policy would have been more robust, but still not sufficiently well performing for our goal.

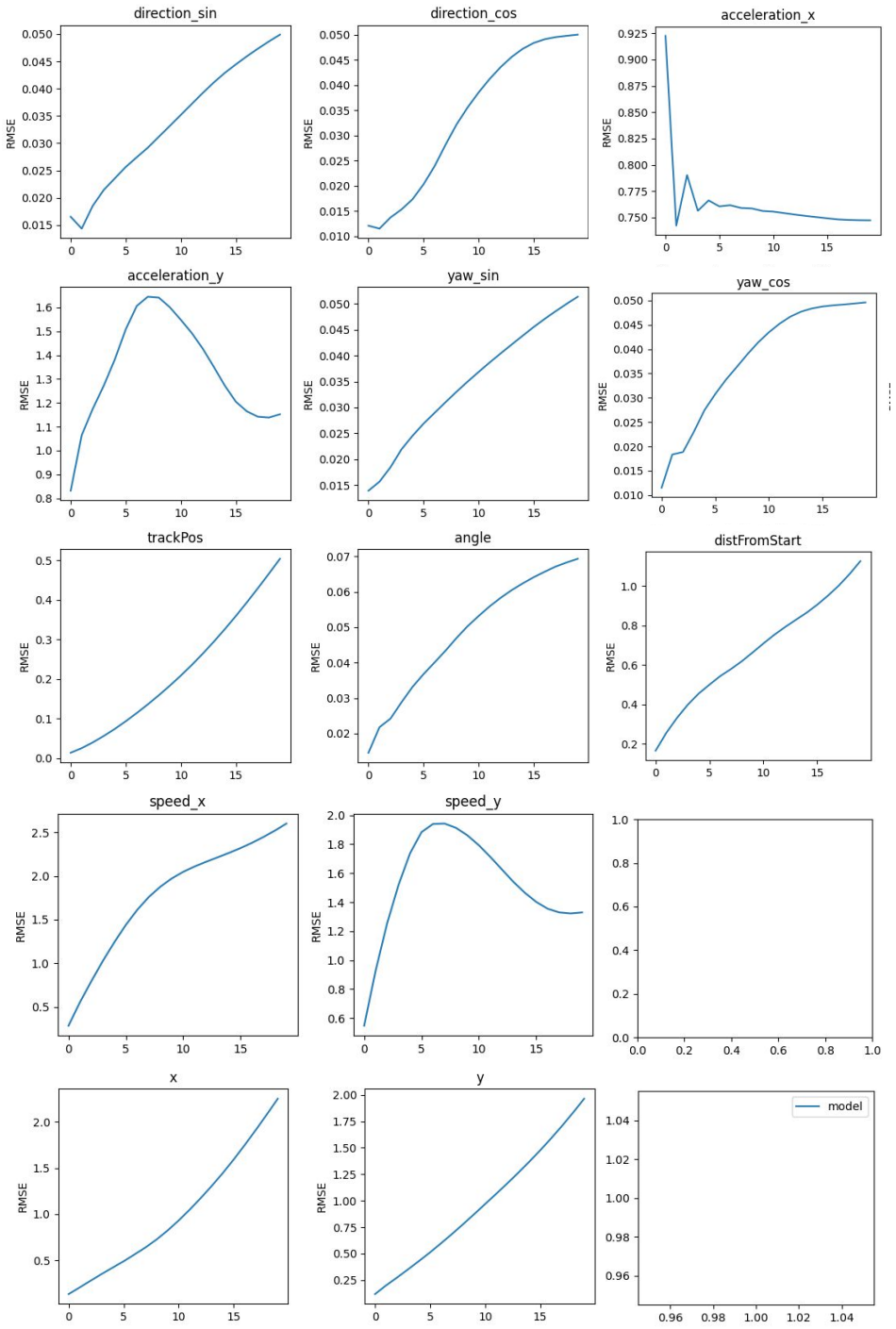


Figure 7.22: Experiment Section 7.2. RMSE considering the state features. The error strictly tends to increase considering the majority of features. The only features that present a different error shape, i.e. rather Gaussian, are the y components of acceleration and speed.

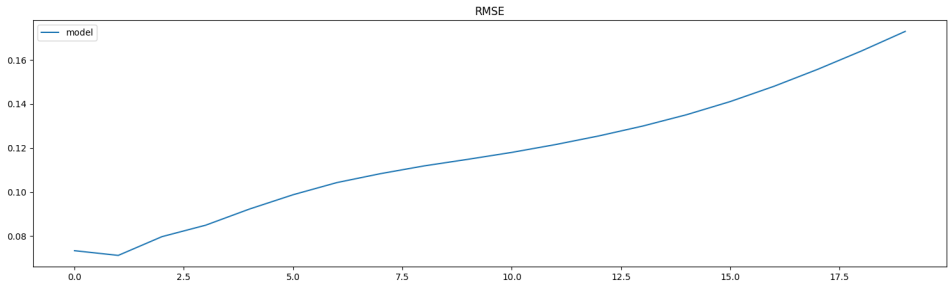


Figure 7.23: Experiment Section 7.2. RMSE of the model along the fitting of the transition model. The model arrives to a model error of around 0.16. Error shape is monotonically increasing. The error is calculated with the predictions of the transition model and the sequence of states.

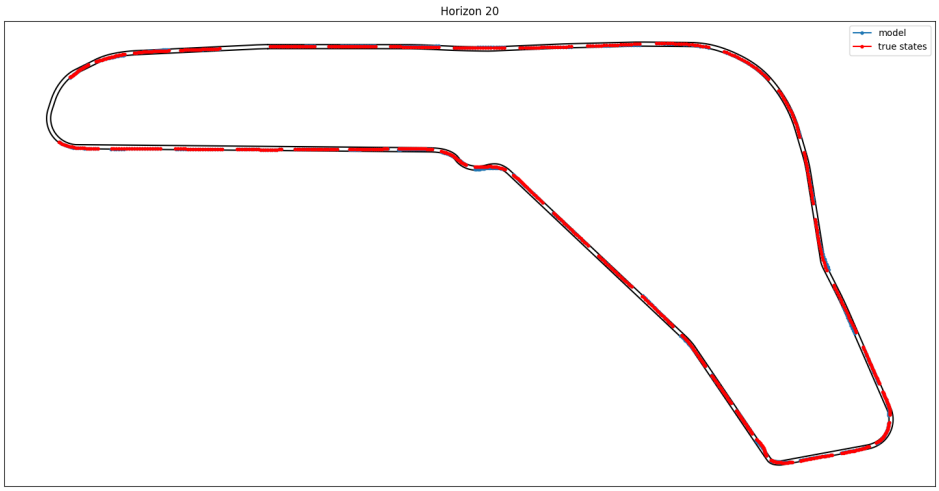


Figure 7.24: Experiment Section 7.2. Prediction at horizon 20.

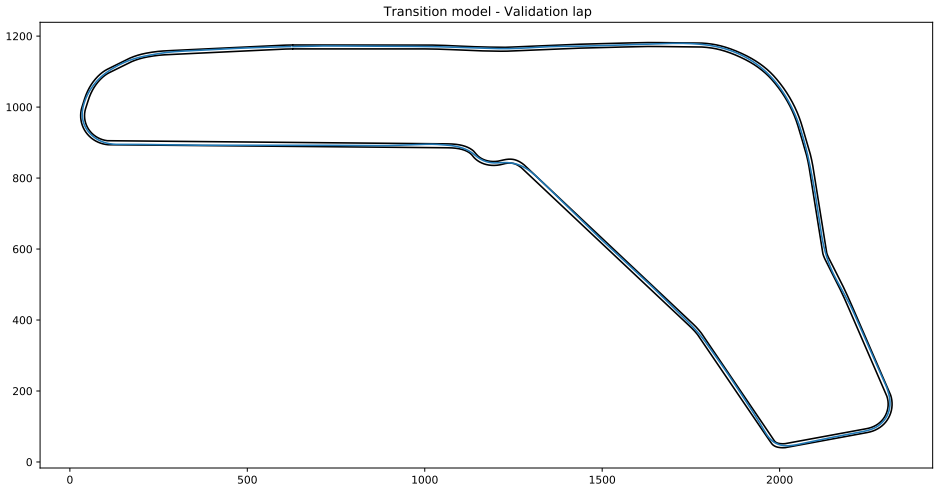


Figure 7.25: Experiment Section 7.2. Validation lap of the transition model.

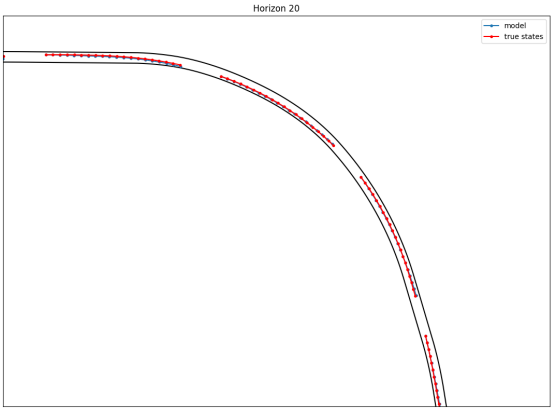


Figure 7.26: Section 7.2.1. First curve detail of transition model.

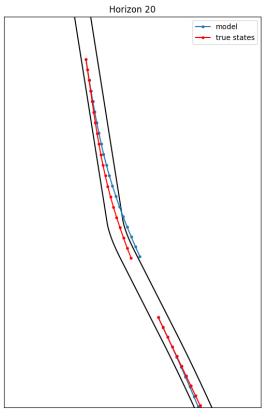


Figure 7.27: Section 7.2.1. Mini chicane detail of transition model.

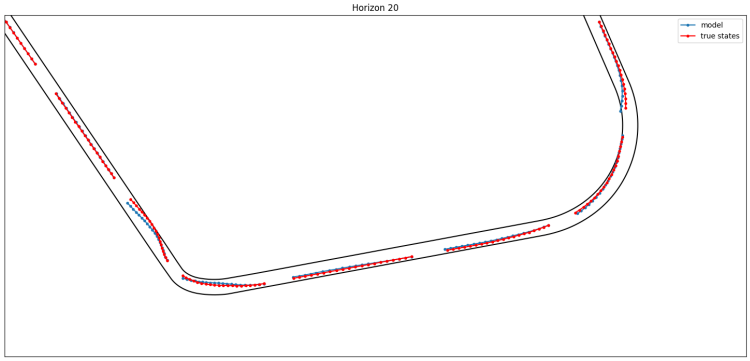


Figure 7.28: Section 7.2.1. First 90° curve and Second 90° curve detail of transition model.

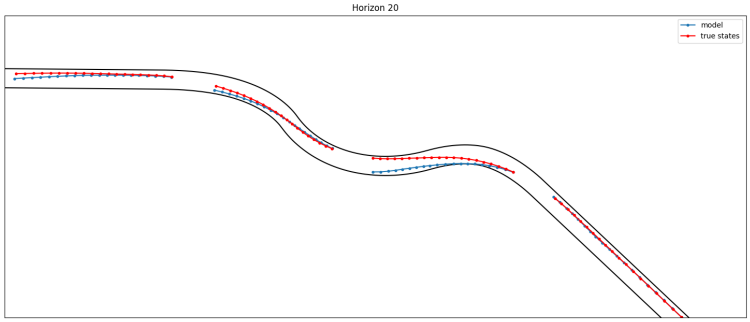


Figure 7.29: Section 7.2.1. Chicane detail of transition model.

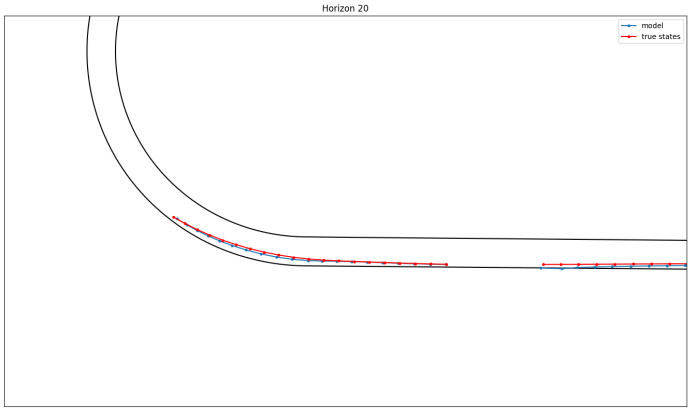


Figure 7.30: Section 7.2.1. Parabolic curve detail of transition model.

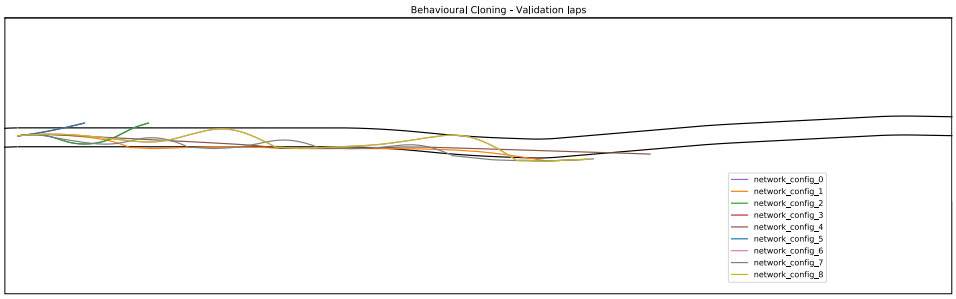


Figure 7.31: Experiment Section 7.2. Validation lap of the Behavioural Cloning Policies. Every line corresponds to a tested network topology. In total, 9 network topologies have been tested. The mere Behavioural Cloning policy is not enough to perform a lap of adequate quality.

7.2.3 MCTS planning phase

Once the transition model and base policy have been trained, the learning phase with MCTS takes place. In this phase, MCTS has been applied with the transition model described in Section 7.2.1 and a policy of behavioural cloning, in order to compute the expert prediction, i.e. the *oracle*, see Section 5.3. In this experiment, the MCTS have been applied with an horizon of 40, obtaining a good prediction result with the transition model defined in Section 7.2.1. This experiment has been performed with two approaches that characterize the learning process, the offline case and the online case.

In Figure 7.32, the difference between the *student* policy, in dashed green line, and MCTS prediction, in red line, is shown considering the sector of **Parabolic curve**, see Table 7.3. The input point was the distance from the start along the track, to identify whether to start the prediction and simulation of MCTS. The experimental choice was to iterate MCTS, as explained in Section 6.3.2.1, to obtain a complete trajectory, defined over the time horizon. As depicted in the Figure, the *oracle*, i.e. the MCTS line, tends to tighten in curve, while the *student* tends to drive with a smoother style. Using a larger horizon with respect to the value used for the transition model training proved to have good results, if considering limited values, i.e. 40. With higher horizon values, divergence from the track occurs.

7.2.3.1 Offline Planning

Considering the offline planning case, our goal is simply the direct estimation of the trajectory, given the Transition Model fitted as described in Section 7.2.1 and the Behavioural Cloning Policy described in Section 7.2.2. In this case, we are not interested in driving, only in the estimation of the trajectory, once the Teacher advice has been applied. To this extent, we applied the Transition Model to compute the new controller trajectory, considering the Teacher experience derived with MCTS planning. Once the MCTS plan has been computed, the teaching Algorithm 6 defined in Section 6.3 is applied, to verify the convergence of the *student* policy to the trajectory planned by MCTS. In Figure 7.33, the resulting trajectory of the transition model execution is shown together with the *student* trajectory and MCTS *oracle*. The blue trajectory refers to the learning that is performed through the teaching algorithm, and proceeds with the steps computed with MCTS. In fact, it overlaps with the dashed green line that represents the student. In that point, the teaching algorithm is applied, and the the student policy parameters are updated. Since the teaching algorithm updates the student policy, the blue line after the marked point refers to the new controller. The

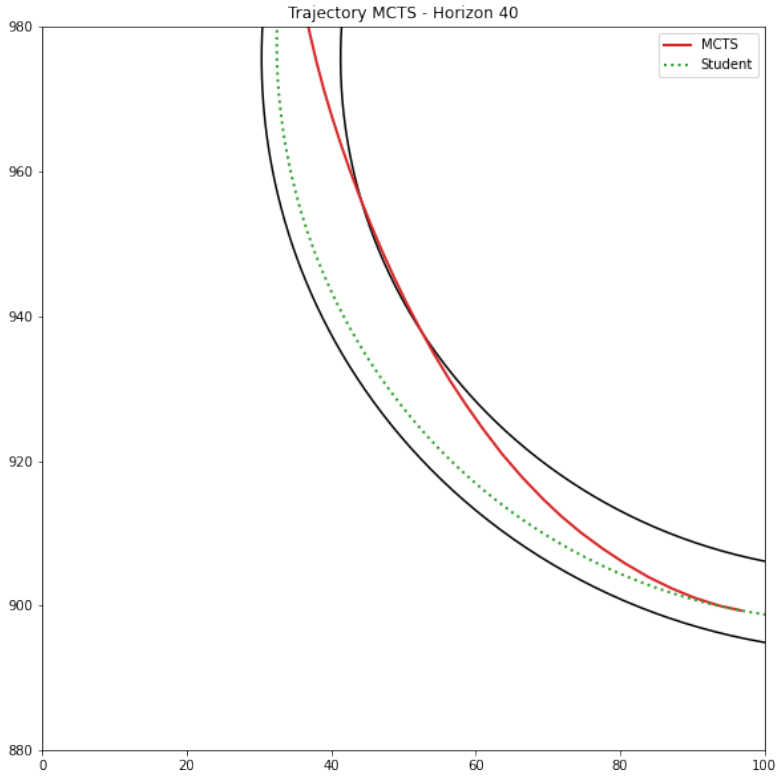


Figure 7.32: Experiment Section 7.2.3. MCTS prediction with horizon 40 in Parabolic curve. In green, the student policy following the reference trajectory as optimized in Section 7.1. In red, the prediction of MCTS, considering an horizon of 40.

algorithm is applied exclusively in the marked point, so, for one step only.

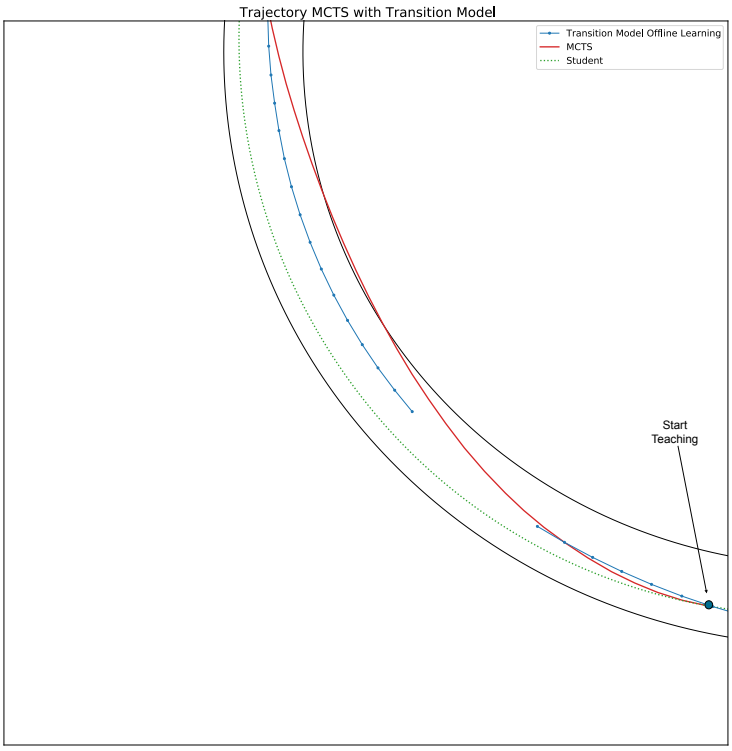


Figure 7.33: Experiment Section 7.2.3. MCTS prediction with horizon 40 in Parabolic curve. In green, the student policy following the reference trajectory as optimized in Section 7.1. In red, the prediction of MCTS, considering an horizon of 40. In blue, the trajectory of the student with the teaching applied. The old controller runs till the marked point. From that distance on, the teaching algorithm is applied and the new controller is executed, the blue line.

7.2.3.2 Online Planning

In the online case, the goal is directly driving, not only deriving an improved trajectory. This application requires the use of the TORCS simulator, to apply the teacher advice while the Student is driving, and correcting the trajectory, through the updated controller, after that point. Therefore, after the expert plan generated with MCTS has been realized, the teaching Algorithm 6 defined in Section 6.3 is applied, to verify the convergence of the *student* policy to the trajectory planned by MCTS. In Figure 7.34, the resulting trajectory of the algorithm execution is shown together with the *student* trajectory and MCTS *oracle*. The blue trajectory before the marked point refers to the old controller. In fact, it overlaps with the dashed green line that represents the student. In that point, the teaching algorithm is ap-

plied, and the the student policy parameters are updated. Since the teaching algorithm updates the student policy, the blue line after the marked point refers to the new controller. The computation of the corrected action parameters has been applied only for the first step. The student is let free to follow its reference afterwards. The student loss is calculated and applied with the *student* policy steer parameters, i.e. the coefficient of the boost in sectors, see boost Equation (6.16), that allows for a more effective steer. We can see that the learning trajectory, in blue, asymptotically converges to the *teacher* position in the space, given the time horizon of the prediction, 40.

In Figure 7.34, also the trajectory obtained with the teaching applied over $k = 20$ steps is applied, value equal to the horizon of training of the transition model. With a larger horizon, the trajectory does not behave well. Also for the purple line the marked point represents the division between the old controller and the new one, i.e. after teaching. In this case, the Teaching Algorithm has been applied not only for the first step (i.e. marked point for the start of teaching), but also to the subsequent 14 steps (i.e. points in the trajectory). The purple line converges faster with only a limited number of steps to the optimal trajectory of the teacher. The difference between the controller that learns with single step teaching and the controller that learns through multiple steps is consistent, even if the single step teaching still performs well.

As it is possible to observe from Figure 7.35, the basic student configuration, in the left plot, is not able to immediately follow the teacher trajectory. After that the teaching algorithm is applied, the student trajectory improves and tends towards the teacher trajectory. In the right plot, it is evident that the student has improved consistently its trajectory, being able to better emulate teacher trajectory. In this case the student has tried to directly follow the reference proposed by the teacher before the teaching algorithm has been applied, i.e. with the optimized student settings, in the left plot. After the teaching algorithm has been applied, the student applies an improved policy (in the right plot), therefore it is able to better follow the reference provided by the teacher.

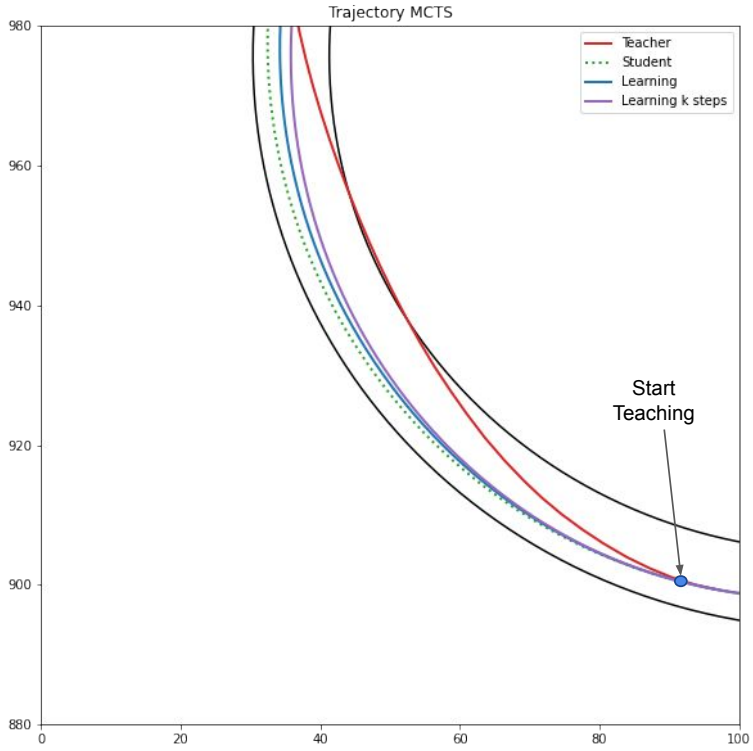


Figure 7.34: Experiment Section 7.2.3. MCTS prediction with horizon 40 in Parabolic curve. In green, the student policy following the reference trajectory as optimized in Section 7.1. In red, the prediction of MCTS, considering an horizon of 40. In blue, the trajectory of the student with the teaching applied. The old controller runs till the marked point. From that distance on, the teaching algorithm is applied and the new controller runs, the blue line. In purple, the trajectory of the student with the teaching applied considering that teaching is performed over k=20 steps in the prediction.

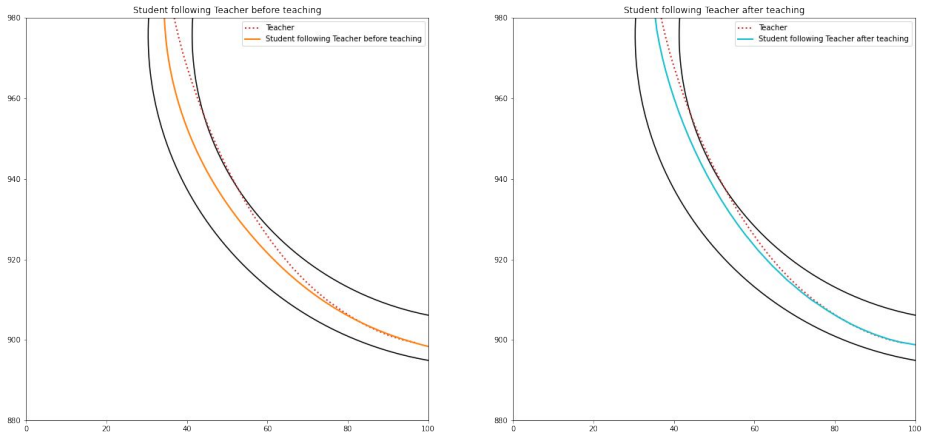


Figure 7.35: Experiment Section 7.2.3. Student follows the Teacher. In both plots, the dashed red line represents the teacher. In the left plot, the orange line represents the trajectory obtained by the student that tries to follow the teacher before that the teaching algorithm is applied. In the right plot, the cyan line represents the trajectory obtained by the student that tries to follow the teacher after that the teaching algorithm is applied.

Chapter 8

Conclusions

In this thesis work, we faced the *Teacher-Student* Reinforcement Learning framework for the field of autonomous driving. Our approach consisted in the definition of an autonomous agent, the Student, consisting in a parametrized policy, able to improve its driving performance with the support of an expert entity, i.e. the Teacher. The Environment was modelled as a Markov Decision Process on TORCS simulator, on which experimental results' evaluation has been performed. We empirically defined the student policy, and performed optimization over it according to the RL framework and PGPE algorithm. For the teacher, we modified the planning algorithm MCTS, since the entire sequence of states and actions of the simulation is returned, i.e. not only the next action. We used a transition model with a multi-step loss function adopted in the prediction of the planning algorithm exploited in this thesis work, i.e. Monte Carlo Tree Search, to outperform the effects that can be obtained with a basic one-step loss function. We proposed and implemented an offline teaching algorithm to generate an improved trajectory, starting from the Student basic policy and the MCTS prediction. We perform several experiments to optimize the Student policy. First category of successful experiments were related to the optimization of the speed. The second category of experiments that improved the Student policy was related to the optimization of the boundaries that defined the sectors of the variable that identifies the position with respect to the vehicle. The shape of the functions dependent on this variable was retrieved from human demonstrations, and the intermediate points, i.e. the internal points that fragmented sectors, were learnt, to better approach critical curves. The last effective student optimization regards the position and intensity of the boost in steering actions. This procedure brought the optimization of the student for the actions of brake and throttle (see Section 7.1.1), and the

steering action (see Sections 7.1.2, and 7.1.3). Experiments regarding the teacher component were mainly related to the definition of the most appropriate transition model, for which the multi-step loss function has been adopted. Once the best configuration for the transition model and for the behavioural cloning policy were found, the experiments proceeded in the successful application of the proposed teaching algorithm both in the offline and online case, see Section 7.2.3.

8.1 Future Work

This thesis work is located in an open and new research field. Several future extensions of this work are possible. First, the transition model adopted in MCTS prediction can be further studied and developed. Extensions can regard the definition of a loss able to outperform over larger horizons. Another possible extension can be the increase of flexibility of the teaching algorithm, in order to allow suppleness when the student is unable to improve with the advice he receives from the teacher. Another extension can be the definition of the teaching process in the online case for the multiple laps case. In this thesis work we limit the study to the single lap case. In a future extension, the performance on more laps on the circuit can be studied. This approach leads to the open study of the modelling of other state features, e.g. tire wear or fuel consumption over several laps, that are fundamental to characterize the driving style in the non-single lap case, and can be object of study for the further development of the Student controller.

Appendix A

Definitions, Theorems and Proofs

A.1 Definitions

Definition A.1.1 (GLIE)**.** Greedy in the Limit of Infinite Exploration (GLIE):

- All state-action pairs are explored infinitely many times

$$\lim_{k\to\infty} N_k(s,a) = \infty.$$

- The policy converges on a greedy policy

$$\lim_{k\to\infty} \pi_k(a|s) = \left(a = \arg\max_{a'\in\mathcal{A}} Q_k(s',a')\right).$$

The concept is that GLIE assumption requires that exploration factor disappears in the limit, since the policy that is selected in each state converges to the greedy policy asymptotically.

Definition A.1.2 (Markov Property)**.** In a Markov Decision Process, the state must include information about all aspects of the past agent–environment interaction that make a difference for the future. If it does, then the state is said to have the Markov property [75].

A.2 Theorems

Theorem A.2.1 (Policy Improvement Theorem)**.** *Let π and π' be any pair of deterministic policies, such that*

$$Q^{\pi}\left(s,\pi'(s)\right) \geq V^{\pi}(s), \forall s \in \mathcal{S}. \tag{A.1}$$

Then the policy π' must be as good as, or even better, than π:

$$V^{\pi'}(s) \geq V^{\pi}(s), \forall s \in \mathcal{S}. \tag{A.2}$$

Proof. To prove this theorem, is sufficient to exploit the relationship between the state-value function and the action-value function, and notice that:

$$\begin{aligned} V^{\pi}(s) &\leq Q^{\pi}\left(s, \pi'(s)\right) = \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi'(s_{t+1})) | s_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 Q^{\pi}(s_{t+2}, \pi'(s_{t+2})) | s_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \ldots | s_t = s\right] = V^{\pi'}(s). \end{aligned} \tag{A.3}$$

□

Theorem A.2.2 (Convergence and Contractions Theorem)**.** *Define the max–norm as $\|V\|_\infty = \max_s |V(s)|$.*

Value Iteration, see Section 3.1.2.2, converges to the optimal state-value function $\lim_{k\to\infty} V_k = V^*$.

Proof. To prove this theorem, is sufficient to exploit the relationship between the state-value function and the action-value function, and notice that:

$$\begin{aligned} \|V_{k+1} - V^*\|_\infty = \|T^* V_k - T^* V^*\|_\infty & \\ &\leq \gamma \|V_k - V^*\|_\infty \\ &\ldots \\ &\leq \gamma^{k+1} \|V_0 - V^*\|_\infty \to \infty. \end{aligned} \tag{A.4}$$

□

Theorem A.2.3.

$$\|V^{(t+1)} - V^{(t)}\|_\infty < \epsilon \implies \|V^{(t+1)} - V^*\|_\infty < \frac{2\epsilon\gamma}{1-\gamma}. \tag{A.5}$$

A.3 Properties of Bellman Operators

Operators are reported again in this Section. There are two Bellman Expectation Operators and two Bellman Optimality Operators:

- Bellman Expectation Operator for V^π, defined as $T^\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$:

$$(T^\pi V^\pi)(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \right).$$

- Bellman Expectation Operator for Q^π, defined as $T^\pi : \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|} \to \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$:

$$(T^\pi Q^\pi)(s,a) = R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) \sum_{a'\in\mathcal{A}} \pi(a'|s') Q^\pi(s',a').$$

- Bellman Optimality Operator for V^*, defined as $T^* : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$:

$$(T^* V^*)(s) = \max_{a\in\mathcal{A}(s)} \left(R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) V^*(s') \right).$$

- Bellman Optimality Operator for Q^*, defined as $T^* : \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|} \to \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$:

$$(T^* Q^*)(s,a) = R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) \max_{a'\in\mathcal{A}(s')} Q^*(s',a').$$

Properties of Bellman operators:

- **Monotonicity**: if $f_1 \le f_2$ component-wise, then:

$$T^\pi f_1 \le T^\pi f_2, \qquad T^* f_1 \le T^* f_2$$

- **Max-Norm Contradiction**: for two vectors f_1 and f_2:

$$\|T^\pi f_1 - T^\pi f_2\|_\infty \le \gamma \|f_1 - f_2\|_\infty$$

$$\|T^* f_1 - T^* f_2\|_\infty \le \gamma \|f_1 - f_2\|_\infty$$

- V^π is the unique fixed point of T^π
- V^* is the unique fixed point of T^*
- For any vector $f \in \mathbb{R}^{|\mathcal{S}|}$ and any policy π, we have:

$$\lim_{k\to\infty} (T^\pi)^k f = V^\pi$$

$$\lim_{k\to\infty} (T^*)^k f = V^*.$$

Bibliography

[1] Abdul Afram and Farrokh Janabi-Sharifi. Theory and applications of hvac control systems–a review of model predictive control (mpc). *Building and Environment*, 72:343–355, 2014.

[2] Alexandros Agapitos, Julian Togelius, Simon M Lucas, Jurgen Schmidhuber, and Andreas Konstantinidis. Generating diverse opponents with multiobjective evolution. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 135–142. IEEE, 2008.

[3] Karl J Åström and Tore Hägglund. Pid control. *IEEE Control Systems Magazine*, 1066(033X/06), 2006.

[4] Mohammad Gheshlaghi Azar, Alessandro Lazaric, and Emma Brunskill. Regret bounds for reinforcement learning with policy advice. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 97–112. Springer, 2013.

[5] Bharathan Balaji, Sunil Mallya, Sahika Genc, Saurabh Gupta, Leo Dirac, Vineet Khare, Gourav Roy, Tao Sun, Yunzhe Tao, Brian Townsend, Eddie Calleja, Sunil Muralidhara, and Dhanasekar Karuppasamy. Deepracer: Educational autonomous racing platform for experimentation with sim2real reinforcement learning, 2019.

[6] Lakshmi Dhevi Baskar, Bart De Schutter, and Hans Hellendoorn. Model-based predictive traffic control for intelligent vehicles: Dynamic speed limits and dynamic lane allocation. In *2008 IEEE intelligent vehicles symposium*, pages 174–179. IEEE, 2008.

[7] C.M. Bishop. *Pattern recognition and machine learning*, chapter Linear models for regression, pages 137–147. Springer, 2006.

[8] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Mon-

fort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[9] Paolo Giuseppe Emilio Bolzern, Riccardo Scattolini, and Nicola Luigi Schiavoni. *Fondamenti di controlli automatici*. McGraw-Hill, 2008.

[10] Georgios Boutsioukis, Ioannis Partalas, and Ioannis Vlahavas. Transfer learning in multi-agent reinforcement learning domains. In *European Workshop on Reinforcement Learning*, pages 249–260. Springer, 2011.

[11] Facundo Bre, Juan M Gimenez, and Víctor D Fachinotti. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, 158:1429–1441, 2018.

[12] Stephen T Buckland. Monte carlo confidence intervals. *Biometrics*, pages 811–817, 1984.

[13] Maya Cakmak and Manuel Lopes. Algorithmic and human teaching of sequential decision tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, 2012.

[14] Emilio Capo and Daniele Loiacono. Short-term trajectory planning in torcs using deep reinforcement learning. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2327–2334. IEEE, 2020.

[15] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Learning drivers for torcs through imitation using supervised methods. In *2009 IEEE symposium on computational intelligence and games*, pages 148–155. IEEE, 2009.

[16] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. On-line neuroevolution applied to the open racing car simulator. In *2009 IEEE Congress on Evolutionary Computation*, pages 2622–2629. IEEE, 2009.

[17] James L Carroll and Kevin Seppi. Task similarity measures for transfer in reinforcement learning task libraries. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 803–808. IEEE, 2005.

[18] Hyunmin Chae, Chang Mook Kang, ByeoungDo Kim, Jaekyum Kim, Chung Choo Chung, and Jun Won Choi. Autonomous braking system via deep reinforcement learning. In *2017 IEEE 20th International conference on intelligent transportation systems (ITSC)*, pages 1–6. IEEE, 2017.

[19] Marvin T Chan, Christine W Chan, and Craig Gelowitz. Development of a car racing simulator game using artificial intelligence techniques. *International Journal of Computer Games Technology*, 2015, 2015.

[20] Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*, volume 5. Springer, 2007.

[21] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[22] Felipe Leno Da Silva and Anna Helena Reali Costa. Transfer learning for multiagent reinforcement learning systems. In *IJCAI*, pages 3982–3983, 2016.

[23] Felipe Leno Da Silva, Ruben Glatt, and Anna Helena Reali Costa. Simultaneously learning and advising in multiagent reinforcement learning. In *Proceedings of the 16th conference on autonomous agents and multiagent systems*, pages 1100–1108, 2017.

[24] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[25] Marco Dorigo and Marco Colombetti. *Robot shaping: an experiment in behavior engineering*. MIT press, 1998.

[26] Tom Erez and William D Smart. What does shaping mean for computational reinforcement learning? In *2008 7th IEEE International Conference on Development and Learning*, pages 215–219. IEEE, 2008.

[27] Anestis Fachantidis, Matthew E. Taylor, and Ioannis Vlahavas. Learning to teach reinforcement learning agents. *Machine Learning and Knowledge Extraction*, 1(1):21–42, 2019.

[28] Farbod Farshidian, Edo Jelavic, Asutosh Satapathy, Markus Giftthaler, and Jonas Buchli. Real-time motion planning of legged robots: A model predictive control approach. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 577–584. IEEE, 2017.

[29] Aurélie Foucquier, Sylvain Robert, Frédéric Suard, Louis Stéphan, and Arnaud Jay. State of the art in building modelling and energy performances prediction: A review. *Renewable and Sustainable Energy Reviews*, 23:272–288, 2013.

[30] Carlos E Garcia, David M Prett, and Manfred Morari. Model predictive control: Theory and practice—a survey. *Automatica*, 25(3):335–348, 1989.

[31] Wade Genders and Saiedeh Razavi. Evaluating reinforcement learning state representations for adaptive traffic signal control. *Procedia computer science*, 130:26–33, 2018.

[32] Daniel Graves, Nhat M Nguyen, Kimia Hassanzadeh, and Jun Jin. Learning predictive representations in autonomous driving to improve deep reinforcement learning. *arXiv preprint arXiv:2006.15110*, 2020.

[33] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[34] Xuemin Hu, Long Chen, Bo Tang, Dongpu Cao, and Haibo He. Dynamic path planning for autonomous driving on various roads with avoidance of static and moving obstacles. *Mechanical Systems and Signal Processing*, 100:482–500, 2018.

[35] Zhiqing Huang, Ji Zhang, Rui Tian, and Yanxin Zhang. End-to-end autonomous driving decision based on deep reinforcement learning. In *2019 5th International Conference on Control, Automation and Robotics (ICCAR)*, pages 658–662. IEEE, 2019.

[36] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[37] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[38] Ercüment Ilhan, Jeremy Gow, and Diego Perez-Liebana. Teaching on a budget in multi-agent deep reinforcement learning. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[39] David Isele, Reza Rahimi, Akansel Cosgun, Kaushik Subramanian, and Kikuo Fujimura. Navigating occluded intersections with autonomous vehicles using deep reinforcement learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2034–2039. IEEE, 2018.

[40] Jie Ji, Amir Khajepour, Wael William Melek, and Yanjun Huang. Path planning and tracking for vehicle collision avoidance based on model predictive control with multiconstraints. *IEEE Transactions on Vehicular Technology*, 66(2):952–964, 2016.

[41] Michael A Johnson and Mohammad H Moradi. *PID control*. Springer, 2005.

[42] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[43] László T Kóczy, Péter Földesi, and Boldizsár Tüű-Szabó. Enhanced discrete bacterial memetic evolutionary algorithm-an efficacious metaheuristic for the traveling salesman optimization. *Information Sciences*, 460:389–400, 2018.

[44] John R Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT press, 1994.

[45] Alessandro Lazaric. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*, pages 143–173. Springer, 2012.

[46] Jinyu Li, Rui Zhao, Zhuo Chen, Changliang Liu, Xiong Xiao, Guoli Ye, and Yifan Gong. Developing far-field speaker system via teacher-student learning. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5699–5703, 2018.

[47] Yandong Li, ZB Hao, and Hang Lei. Survey of convolutional neural network. *Journal of Computer Applications*, 36(9):2508–2515, 2016.

[48] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[49] Hui Liu, Chengqing Yu, Haiping Wu, Zhu Duan, and Guangxi Yan. A new hybrid ensemble deep reinforcement learning model for wind speed short term forecasting. *Energy*, 202:117794, 2020.

[50] Daniele Loiacono, Pier Luca Lanzi, Julian Togelius, Enrique Onieva, David A Pelta, Martin V Butz, Thies D Lönneker, Luigi Cardamone, Diego Perez, Yago Sáez, et al. The 2009 simulated car racing championship. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):131–147, 2010.

[51] Alie El-Din Mady, Gregory Provan, Conor Ryan, and Kenneth Brown. Stochastic model predictive controller for the integration of building use and temperature regulation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, 2011.

[52] Yishay Mansour and Satinder Singh. On the complexity of policy iteration. *arXiv preprint arXiv:1301.6718*, 2013.

[53] James L McClelland, David E Rumelhart, PDP Research Group, et al. *Parallel distributed processing*, volume 2. MIT press Cambridge, MA, 1986.

[54] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[55] Oscar Montiel, Ulises Orozco-Rosas, and Roberto Sepúlveda. Path planning for mobile robots using bacterial potential field for avoiding static and dynamic obstacles. *Expert Systems with Applications*, 42(12):5177–5191, 2015.

[56] Manfred Morari and Jay H Lee. Model predictive control: past, present and future. *Computers & Chemical Engineering*, 23(4-5):667–682, 1999.

[57] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000.

[58] Umit Ozguner, Christoph Stiller, and Keith Redmill. Systems for safety and autonomous behavior in cars: The darpa grand challenge experience. *Proceedings of the IEEE*, 95(2):397–412, 2007.

[59] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[60] Martin L Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11):1127–1137, 1978.

[61] James B Rawlings. Tutorial overview of model predictive control. *IEEE control systems magazine*, 20(3):38–52, 2000.

[62] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):25–41, 2015.

[63] Thomas Rückstiess, Frank Sehnke, Tom Schaul, Daan Wierstra, Yi Sun, and Jürgen Schmidhuber. Exploring parameter space in reinforcement learning. *Paladyn*, 1(1):14–24, 2010.

[64] Hans-Paul Schwefel and Günter Rudolph. Contemporary evolution strategies. In *European conference on artificial life*, pages 891–907. Springer, 1995.

[65] Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy gradients with parameter-based exploration for control. In *International Conference on Artificial Neural Networks*, pages 387–396. Springer, 2008.

[66] Claudio Semini, Nikos G Tsagarakis, Emanuele Guglielmino, and Darwin G Caldwell. Design and experimental evaluation of the hydraulically actuated prototype leg of the hyq robot. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3640–3645. IEEE, 2010.

[67] Gianluca Serale, Massimo Fiorentini, Alfonso Capozzoli, Daniele Bernardini, and Alberto Bemporad. Model predictive control (mpc) for enhancing building and hvac system energy efficiency: Problem formulation, applications and opportunities. *Energies*, 11(3):631, 2018.

[68] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[69] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*, 2016.

[70] Sahand Sharifzadeh, Ioannis Chiotellis, Rudolph Triebel, and Daniel Cremers. Learning to drive using inverse reinforcement learning and deep q-networks. *arXiv preprint arXiv:1612.03653*, 2016.

[71] David Silver. Reinforcement learning. *University Lecture, University College London*, 2015.

[72] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[73] James C Spall. An overview of the simultaneous perturbation method for efficient optimization. *Johns Hopkins apl technical digest*, 19(4):482–492, 1998.

[74] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[75] Richard S Sutton, Andrew G Barto, et al. Reinforcement learning. *Journal of Cognitive Neuroscience*, 11(1):126–134, 1999.

[76] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768, 2011.

[77] Joan Tarragona, Alvaro de Gracia, and Luisa F Cabeza. Bibliometric analysis of smart control applications in thermal energy storage systems. a model predictive control approach. *Journal of Energy Storage*, 32:101704, 2020.

[78] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

[79] Andrea Tirinzoni, Rafael Rodriguez Sanchez, and Marcello Restelli. Transfer of value functions via variational methods. *Advances in Neural Information Processing Systems*, 31:6179–6189, 2018.

[80] Julian Togelius. *Optimization, imitation and innovation: Computational intelligence and games*. PhD thesis, Citeseer, 2007.

[81] Julian Togelius, Simon Lucas, Ho Duc Thang, Jonathan M Garibaldi, Tomoharu Nakashima, Chin Hiong Tan, Itamar Elhanany, Shay Berant, Philip Hingston, Robert M MacCallum, et al. The 2007 ieee cec simulated car racing competition. *Genetic Programming and Evolvable Machines*, 9(4):295–329, 2008.

[82] Julian Togelius and Simon M Lucas. Evolving controllers for simulated car racing. In *2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1906–1913. IEEE, 2005.

[83] Julian Togelius, Simon M Lucas, and Renzo De Nardi. Computational intelligence in racing games. *Advanced Intelligent Paradigms in Computer Games*, pages 39–69, 2007.

[84] Julian Togelius, Mike Preuss, and Georgios N Yannakakis. Towards multiobjective procedural map generation. In *Proceedings of the 2010 workshop on procedural content generation in games*, pages 1–8, 2010.

[85] Lisa Torrey and Matthew Taylor. Teaching on a budget: Agents advising agents in reinforcement learning. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 1053–1060, 2013.

[86] Niels Van Hoorn, Julian Togelius, Daan Wierstra, and Jurgen Schmidhuber. Robust player imitation using multiobjective evolution. In *2009 IEEE Congress on Evolutionary Computation*, pages 652–659. IEEE, 2009.

[87] Sen Wang, Daoyuan Jia, and Xinshuo Weng. Deep reinforcement learning for autonomous driving. *arXiv preprint arXiv:1811.11329*, 2018.

[88] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[89] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator (2014). *URL: http://www. torcs. org*, 2018.

[90] Qiang Yang. An introduction to transfer learning. In *ADMA*, page 1, 2008.

[91] Shun Yang, Wenshuo Wang, Chang Liu, Weiwen Deng, and J Karl Hedrick. Feature analysis and selection for training an end-to-end autonomous vehicle controller using deep learning approach. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1033–1038. IEEE, 2017.

[92] Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*. Springer, 2018.

[93] Yusen Zhan and Matthew E Taylor. Online transfer learning in reinforcement learning domains. *arXiv preprint arXiv:1507.00436*, 2015.

[94] Peilin Zhao, Steven CH Hoi, Jialei Wang, and Bin Li. Online transfer learning. *Artificial Intelligence*, 216:76–102, 2014.

[95] Luisa Zintgraf, Kyriacos Shiarli, Vitaly Kurin, Katja Hofmann, and Shimon Whiteson. Fast context adaptation via meta-learning. In *International Conference on Machine Learning*, pages 7693–7702. PMLR, 2019.

[96] Qijie Zou, Kang Xiong, and Yingli Hou. An end-to-end learning of driving strategies based on ddpg and imitation learning. In *2020 Chinese Control And Decision Conference (CCDC)*, pages 3190–3195. IEEE, 2020.