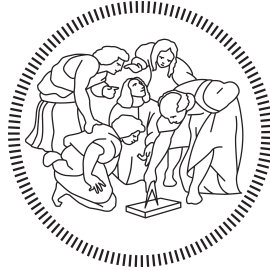


POLITECNICO DI MILANO
Computer Science and Engineering Master Degree
Dipartimento di Elettronica Informazione e Bioingegneria



Improving RL algorithms by human demonstrations for autonomous race driving

**Artificial Intelligence and Robotic Laboratory
of Politecnico di Milano**

Supervisor: Prof. Marcello Restelli

Co-supervisors: Alessandro Lavelli, Dott.

Master thesis of:
Umberto Fazio Matricola
Luca Fucci Matricola

Academic Year 2018-2019

Abstract

TESTO SOMMARIO IN INGLESE

Sommario

TESTO DEL SOMMARIO

Contents

Abstract	I
Sommario	III
1 Introduction	1
2 Context	5
3 Problem Statement	7
4 State of the Art	9
4.1 Theoretical Background	9
4.1.1 Reinforcement Learning	9
4.1.1.1 Markov Decision Process	10
4.1.1.2 Environment	11
4.1.1.3 Reward	11
4.1.1.4 Policy	11
4.1.2 Policy Evaluation	12
4.1.3 Temporal Difference Learning	12
4.1.3.1 SARSA	13
4.1.3.2 Q-Learning	13
4.1.3.3 Issues with Temporal Difference Learning . .	13
4.1.4 Deep Reinforcement Learning	13
4.1.4.1 Actor-Critic Methods	13
4.1.4.2 Fitted Q-Iteration	13
4.1.4.3 Deep Deterministic Policy Gradient	13
4.1.5 Imitation Learning	13
4.1.5.1 Teacher-Student Interaction for Behavioral Cloning	13
4.1.5.2 Policy Optimization via Importance Sampling	13
4.2 Related Works	16

4.2.1	Reinforcement Learning in Videogames	16
4.2.2	Autonomous Driving in Videogames	18
4.2.3	Reinforcement Learning in Racing Videogames	22
4.2.4	Imitation Learning in Videogames	24
4.2.5	Imitation Learning in Racing Videogames	25
5	Software Architecture	31
6	Methods	33
7	Experiments	35
8	Conclusion	37
	Bibliography	39

List of Figures

1	From [29]. The framework of Reinforcement Learning, Imitation Learning, and their integration. The demonstrations of human experts can be used to pretrain a model for RL, to shape the reward/policy of RL, and so on.	14
2	From [3]. An example of Bézier curve: black points are the control point of the curve; blue lines simply connect with straight segments the control points; red line is the resulting Bézier curve.	19
3	From [2]. Red vector, the yellow vector, green vectors, and the blue vector represent the zero sensor, the base sensor, auxiliary sensors, and the final angle to take the turn, respectively.	20
4	Kinematic Bicycle Model: x and y are the coordinates of the center of mass in an inertial frame (X, Y) . ψ is the inertial heading and v is the speed of the vehicle. l_f and l_r represent the distance from the center of the mass of the vehicle to the front and rear axles, respectively. β is the angle of the current velocity of the center of mass with respect to the longitudinal axis of the car. δ_f and δ_r are respectively the front and rear steering. Since in most vehicles the rear wheels cannot be steered, typically δ_r is assumed equal to 0.	23
5	From [27]. Comparison of the performance of trajectory tracking and time-optimal driving on the experimental setup. . . .	23
6	From [4]. Lookahead Sensor.	26
7	From [4]. Trajectories comparison in track G-track-1: bot Inferno (blue line), 8 segments lookahead (red line), 16 segments lookahead (black line).	27

List of Tables

Chapter 1

Introduction

In the last decade autonomous driving has been deeply studied in several fields of research: automotive, military, racing cars, and even flight. The goal is to disrupt the need of a human driver or pilot, resulting in a reduction in costs and an increase of safety, for instance by reducing the number of accidents per year due to human error. Moreover, they could learn from their errors, and transmit to each other autonomous vehicle instantaneously, so that each of them is constantly up to date. However, researches in the area of autonomous driving traces back to the birth of the first computers. In the '40, the scientific community were trying and gain insight about the human mind, in order to build artificial agents that could replicate -or even outperform- human in some specific tasks. That's how computer science branched into paths such as artificial intelligence and robotics. One of the main goal of the research of artificial intelligence has always been autonomous driving, that is the capability for an agent of perceiving the world and moving accordingly in order to reach a specific target. Autonomous cars consists in some piece of hardware capable of moving, such as a traditional car, or a drone, equipped with some sensors give insight on the environment, one or more computer units with a decision making algorithm, which compute an action to perform by means of some actuators, such as a throttle, a break, a steering, or a transmission. One of the first experiments in embodying some intelligence in an artificial agent were William Grey Walter's Turtle Robots in the '50s. They were capable of moving in the surroundings by sensing the environment in a simplified manner. They consisted in front wheel drive tricycle-like robots covered by a clear plastic shell, and were provided with a photocell and a bump detector as sensors, which resulted in the action of a motor. Despite their simple behaviors, the technique Walter used are reflected in today's reactive and biologically-inspired robots such as

those based on the B.E.A.M philosophy. Later on, in 1989, a new tentative of building an autonomous car was ALVINN, which stands for Autonomous Land Vehicle In a Neural Network, developed for military research. At that time the technology was not sophisticated enough to provide the computation required to drive in real time, but in a sense the premise of the algorithm used nowadays was already there. In fact, neural networks are today the essential tools for building an autonomous car. Later on with the advent of more and more sophisticated electronic components, such as sensors (Lidars and Radars, which are capable of scanning the environment at 360 degree via electromagnetic waves), and with the continuous improvement of electronic components, such as GPUs, autonomous driving started being a hot topic in industry beside scientific research. In the last decade, some of the traditional automotive companies, such as BMW, Mercedes-Benz, General Motors, Audi, started investing in this reasearch providing their cars with a multitude of sensors and algorithms to make their cars autonomous. Ford is another manufacturer with deployments already in play, with self-driving vehicles being tested in Pittsburgh, Palo Alto, Miami, Washington D.C. and Detroit, with Austin, Texas joining them soon. Together with its partner Argo AI, Ford has plans to trial its fleet of self-driving cars in Austin with a view towards launching a wider-reaching autonomous taxi and delivery service in 2021. Tesla, one of the companies founded by Elon Musk, is also making big steps forward in taking autonomy into mainstream use, both in terms of real world use cases and potential monetization of self-driving technologies. Tesla has supplied customers with more than 780,000 vehicles since launching, the majority of which arrive with pre-installed, self-driving capabilities available to users who purchase the requisite software. Tesla autonomous vehicles have logged huge levels of miles driven since their introduction, growing from 0.1 billion miles in May 2016 to an estimated 1.88 billion miles as of October 2019. Waymo, the newborn firm from Google's Alfabet, has been carrying out successful trials of autonomous taxis in California, transporting over 6,200 people in the first month and many thousands since. They're proving a practical business case for autonomous vehicles. Also in the U.S., Walmart is using autonomous cargo vans to deliver groceries in Arizona, while Pizza Hut is working with Toyota on a driverless electric delivery vehicle that even has a mobile kitchen in it to cook pizzas en route to your house. Parallely, in the military field, DARPA (Defense Advanced Research Projects Agency) has been proposing every year since 2007 a challenge called DARPA Grand Challenge, in which scientist teams dare each other to reach some targets as fast as possible. The reasons for this race to the autonomous driving are millions of possible accidents avoided

per year, and a reduction of pollution and costs by sharing cars which are capable of transporting people without human intervention needed. This growth in the research on autonomous cars led to a formalization of the levels of automation: Society of Automotive Engineers (SAE) introduced six levels of automation: level 0 is no automation, that is traditional cars we are accustomed to. With Level 1, driver assistance, relating to computer assistance of simple driving functions like the cruise control or automated braking systems. Cruise control consists in the capability of maintaining a certain target speed, whatever the slope of the road, weather condition, asphalt roughness. It can be accomplished via some speed sensors and a PID controller, there is no need of complex artificial intelligence. Automated braking systems involves stopping the car or reducing the speed whenever an object come across, be it a vehicle or a pedestrian. It requires proximity and distance sensors (ultrasonic or laser sensors) and may follow some manual rules based on thresholds. Lane Crossing Alert makes the car capable of notify whenever it crosses another lane, and can be achieved with camera sensors. Level 2 refers to partial automation, where the vehicle assists drivers with steering or acceleration, allowing drivers to disengage from some tasks. For example instance , Adaptive Cruise Control, Lane Keeping, or self-parking. Level 3 concerns conditional automation, where the vehicle takes over some of the monitoring of the environment from the driver, using sensor technology like LiDAR. That's what the company Tesla is currently developing: their cars are able of moving in the surroundings but the human intervention is still required in dangerous situations. Level 4 is high automation, where much greater control has been handed to the vehicle, which is in charge of steering, braking, accelerating, monitoring the vehicle and roads, and also responding to events like deciding when to change lanes, turn or use signals. Level 5 is full automation. No company is currently able to reach this level of automation. Currently, most of the cars are embodied with level 1 or 2. However, level 3 and 4 are still object of research, especially by tech companies such as Tesla and Waymo, whose promise is to reach these levels of automation in ten years, and level 5 in twenty, enabling their cars with the power of fully replacing human drivers. The reasons why today full autonomous driving has not being implemented yet are the extremely huge amount of data required for perceiving the complex world of the urban scenario and for computing the consequent actions. In fact, in order to be able to perform this vast computations, several computers and GPUs are needed aboard on the car, resulting in cost, weight, and power consumption. This is the strategy adopted by Tesla so far, whose cars to day span from a price of 85000 dollars to 120000 dollars, and weigh about

2000kg. Another approach, adopted by companies such as Google, is to perform computation on remote servers in their datacenters. However, current mobile connections such as 4G, makes impractical the transmission of data provided each second by the vast amount of sensors. In the future, the advent of 5G could be a game changer in this sense, which should provide a larger in bandwidth and more stable internet connection. That's why the scientific community is starting downstepping the complexity of the task, trying to build autonomous agents in a closed and controlled environment. This lead to a simplification of the problem, avoiding the need of a real time mapping of the environment, and excluding unattended events such as pedestrian coming across. Whether it's a matter of hardware or software, today such goal is still out of reach. Rather, some firms are focusing their attention on making a car which is autonomous in a specifing task. For example, good results have been achieved in keeping a lane on a highway, or stopping with a pedestrian coming across. Alternatively, good or complete levels of autonomy could be achieved in , where the perception part could be performed by simple sensors such as gps and odometry sensors. An example could be a race track, such as Formula 1 tracks, where the environment is known apriori, and that's where our thesis move into. In this paper we will tackle the problem of following a trajectory driven beforehand by a human driver on a race car and, if possible, to improve it.

Chapter 2

Context

TESTO CAPITOLO 2

Chapter 3

Problem Statement

TESTO CAPITOLO 3

Chapter 4

State of the Art

The objective of this chapter is to go into detail about key aspects of reinforcement learning, the core topic of this thesis. In particular, we will describe the concept of Decision Markov Process, the experiment environment, the concept of state, actions, and reward function. Eventually, we will give insight about the algorithms we used in this project: Fitted Q-Iteration, and Deep Deterministic Policy Gradient. Then, in the following section, we will show some successful application on reinforcement learning to autonomous driving, videogames, and in particular racing games.

4.1 Theoretical Background

4.1.1 Reinforcement Learning

Together with Supervised Learning and Unsupervised Learning, Reinforcement Learning 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. Whereas 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 Reinforcement Learning paradigm an autonomous agent learns to improve its performance at an assigned task by interacting with its environment. Typically, Reinforcement Learning agents are not explicitly taught how to act by an expert. Instead, the agent is free to explore the environment in which it lives, and its performance is evaluated by a reward function. Its goal is to maximize the reward by acting accordingly. The resulting reward results from the sum of the reward gained at every single step in its exploration process: for each state, the agent chooses an

action to take and receives a reward based on the usefulness of its decision. Eventually, the agent learns the way to obtain the highest reward by exploiting knowledge learned about the expected utility of different state-action pairs. The challenge of Reinforcement Learning is to design the best tradeoff between exploration and exploitation, that is the capability of an agent of 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.

4.1.1.1 Markov Decision Process

The Reinforcement Learning problem can be formalized as a Markov Decision Process. A MDP is a tuple composed by:

- A finite set of states D : it encompasses every possible state of the process
- A finite set of actions A : it represents the possible action the agent can take at any moment
- A reward function $r = \psi(s_t, a_t, s_{t+1})$, which represents the reward given to the agent at state s_t taking the action a_t landing in the state s_{t+1}
- A transition probability model $T(s_t, a_t, s_{t+1}) = p(s_{t+1}|s_t, a_t)$, which indicates the probability of landing in a state s_{t+1} being in a state s_t taking an action a_t

This provides the framework in which Reinforcement Learning operates: its goal is to find a policy π which specifies the actions to take in each state in order to maximize the reward over time i.e. fulfill the task. Reinforcement learning can solve Markov decision processes without explicit specification of the transition probabilities; the values of the transition probabilities are needed in value and policy iteration. In reinforcement learning, instead of explicit specification of the transition probabilities, the transition probabilities are accessed through a simulator that is typically restarted many times from a uniformly random initial state. Reinforcement learning can also be combined with function approximation to address problems with a very large number of states. Reinforcement Learning operates under the Markov Assumption, under which the current state represents all the information

needed to take an action, regardless from the past states and actions. In other words: "The future is independent of the past given the present"

4.1.1.2 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 (such as e.g. coordinates, velocities, acceleration) or raw (such as images or signals). On the representation of the environment depends the choice of the algorithm suitable to perform a learning process.

4.1.1.3 Reward

At each timestep, the agent receives a reward by the reinforcement learning algorithm, accordingly to the usefulness of action taken. At the end of an episode (a finite sequence of states) the return is typically computed as

$$R_t = \sum_{k=0}^T \gamma^k r_{t+k} + \gamma^T$$

where γ represents a discount factor comprised between 0 and 1, which stands for a learning parameter that influences how the agent considers future or present reward. A small γ results in a higher consideration of present rewards than the future (often called "myopic" evaluation), while with a γ near to 1, the agent will consider equally the reward coming from the present either from the future (also called "far-sighted" evaluation). The role of the discount factor resides in the fact that, besides being mathematically convenient, it avoids infinite returns in cyclic Markov Process, and it's capable of addressing a non-fully represented uncertainty about the future.

4.1.1.4 Policy

The goal of reinforcement learning is finding a policy π , which maps states to a probability distribution over the actions, in order to maximize the return. This policy is called optimal policy π^* . If from a state s_t the agent takes always the same action a_t , the policy is deterministic: $\pi(s_t) = a_t$. Otherwise, if the taking of an action rather than another is due to a probability distribution, the policy is called stochastic: $\pi(a_t|s_t) = p_i, 0 \leq p_i \leq 1$

4.1.2 Policy Evaluation

In order to evaluate the usefulness of a policy, we now introduce the concept of Value Function, which is computed as the discounted sum of rewards as discussed before in the return.

$$V^\pi(s) = \mathbb{E}_\pi[R_t | s_t = s] = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} | s_t = s \right] = \mathbb{E}_\pi \left[r_{t+1} + \sum_{i=0}^{\infty} \gamma^i r_{t+i+2} | s_t = s \right]$$

From which one can derive the recursive so-called Bellman Equation:

Analog to the value functions is the concept of action-value function, or Q-value, which is the expected return after taking an action a_t in state s_t and thereafter following policy π :

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i > t} \sim, a_{i > t} \sim \pi} [R_t | s_t, a_t]$$

and the relative Bellman Equation:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E} [r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi} [Q^\pi(s_{t+1}, a_{t+1})]]$$

To obtain the best policy that yields the best value function there are several methods, the one discussed here takes into consideration a quality function (also known as Q-value) to evaluate the estimated accumulated rewards obtained following a certain policy. It has a similar definition of the value function but the Q-value takes also into consideration the action. So it established the long term rewards of applying an action to a state and then following the considered policy.

4.1.3 Temporal Difference Learning

Temporal difference learning (TD) [23] algorithms are able to learn from raw experiences without the need for a model of the environment. They combine ideas from dynamic-programming and from Monte Carlo methods but while Monte Carlo algorithms need to reach the end of a learning episode to update the value function TD is able to update it at the end of each step and so there is not the need to ignore some episode as in typical Monte Carlo algorithms. To be able to change the value function at each step it is needed to make estimations based on already learned ones, and this can be done with some ideas from dynamic-programming, but while DP needs a precise model of the environment dynamics TD does not and is so more suitable for unpredictable tasks.

4.1.3.1 SARSA

4.1.3.2 Q-Learning

4.1.3.3 Issues with Temporal Difference Learning

4.1.4 Deep Reinforcement Learning

4.1.4.1 Actor-Critic Methods

The "Critic" estimates the value function. This could be the action-value (the Q value) or state-value (the V value). The "Actor" updates the policy distribution in the direction suggested by the Critic (such as with policy gradients). and both the Critic and Actor functions are parameterized with neural networks. In the derivation above, the Critic neural network parameterizes the Q value - so, it is called Q Actor Critic.

4.1.4.2 Fitted Q-Iteration

4.1.4.3 Deep Deterministic Policy Gradient

It is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of a θ at every timestep; this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces. Instead, here we used an actor-critic approach based on the DPG algorithm.

4.1.5 Imitation Learning

4.1.5.1 Teacher-Student Interaction for Behavioral Cloning

[17]

policy gradients

ppge

4.1.5.2 Policy Optimization via Importance Sampling

[15]

When the state and action spaces are finite and small enough, the Q-function can be represented in tabular form, and its approximation (in batch and in on-line mode) as well as the control policy derivation are straightforward. However, when dealing with continuous or very large discrete state and/or action spaces, the Q-function cannot be represented anymore by a table with one entry for each state-action pair. Moreover, in the context

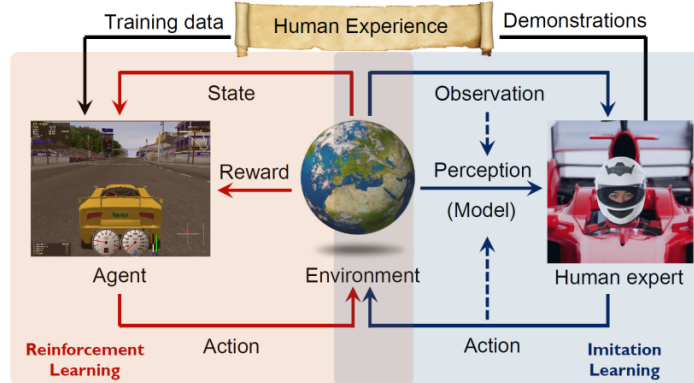


Figure 1: From [29]. The framework of Reinforcement Learning, Imitation Learning, and their integration. The demonstrations of human experts can be used to pretrain a model for RL, to shape the reward/policy of RL, and so on.

of reinforcement learning an approximation of the Q -function all over the state-action space must be determined from finite and generally very sparse sets of four-tuples. To overcome this generalization problem, a particularly attractive framework is the one used by Ormonet and Sen (2002) which applies the idea of fitted value iteration (Gordon, 1999) to kernel-based reinforcement learning, and reformulates the Q -function determination problem as a sequence of kernel-based regression problems. Actually, this framework makes it possible to take full advantage in the context of reinforcement learning of the generalization capabilities of any regression algorithm, and this contrary to stochastic approximation algorithms (Sutton, 1988; Tsitsiklis, 1994) which can only use parametric function approximators (for example, linear combinations of feature vectors or neural networks). In the rest of this paper we will call this framework the fitted Q iteration algorithm so as to stress the fact that it allows to fit (using a set of four-tuples) any (parametric or non-parametric) approximation architecture to the Q -function. The fitted Q iteration algorithm is a batch mode reinforcement learning algorithm which yields an approximation of the Q -function corresponding to an infinite horizon optimal control problem with discounted rewards, by iteratively extending the optimization horizon (Ernst et al., 2003). At each step this algorithm may use the full set of four-tuples gathered from observation of the system together with the function computed at the previous step to determine a new training set which is used by a supervised learning (regression) method to compute the next function of the sequence. It produces a

sequence of Q N -functions, approximations of the Q N -functions defined by Eqn (5). -i mettere algoritmo a pag 6 di tree based batch reinforcement learning

- extra trees Besides Tree Bagging, several other methods to build tree ensembles have been proposed that often improve the accuracy with respect to Tree Bagging (e.g. Random Forests, Breiman, 2001). In this paper, we evaluate our recently developed algorithm that we call "Extra-Trees", for extremely randomized trees (Geurts et al., 2004). Like Tree Bagging, this algorithm works by building several (M) trees. However, contrary to Tree Bagging which uses the standard CART algorithm to derive the trees from a bootstrap sample, in the case of Extra-Trees, each tree is built from the complete original training set. To determine a test at a node, this algorithm selects K cut-directions at random and for each cut-direction, a cut-point at random. It then computes a score for each of the K tests and chooses among these K tests the one that maximizes the score. Again, the algorithm stops splitting a node when the number of elements in this node is less than a parameter n_{min} . Three parameters are associated to this algorithm: the number M of trees to build, the number K of candidate tests at each node and the minimal leaf size n_{min} . The detailed tree building procedure is given in Appendix A.

- Double learning - Double FQI
- DDPG - DPG
- background: pag 3-4 cinesi

Presented in [16], Deep Q-learning from Demonstrations (DQfD) vastly accelerates DQN by pretraining an initial behavior network, and also introducing a supervised loss and a L2 regularization loss when training the target network.

A combination of the advantages of both, the speed of the Riccati controller and the generality of MPC, can be achieved by finding a function that maps state values to control variables, e. g., by training a deep neural network. Such a model could, for example, be learned supervised, as done for PILOTNET, or by reinforcement learning. The latter in particular led to excellent results in the training of such agents for controlling real-world systems such as robots or helicopters. Recent work also shows promising applications of reinforcement learning for autonomous driving by making strategic decisions. Autonomous driving tasks where RL could be applied include: controller optimization, path planning and trajectory optimization, motion planning and dynamic path planning, development of high-level driving policies for complex navigation tasks, scenario-based policy learning for highways, intersections, merges and splits, reward learning

with inverse reinforcement learning from expert data for intent prediction for traffic actors such as pedestrian, vehicles and finally learning of policies that ensures safety and perform risk estimation. Further, it turns out to be suitable in contexts of autonomous racing: the driverless racer could learn a policy that is able to outperform the performance of a human driver, or a policy taught by experts.

4.2 Related Works

In this section we will describe some of the applications of Reinforcement Learning in the context of videogames, autonomous driving and racing.

4.2.1 Reinforcement Learning in Videogames

All the Reinforcement Learning algorithms require interaction with the environment to learn a policy from the gathered experience. Trajectories can be collected by using real world environment or through a simulator. Both approaches have pros and cons, and the choice mainly depends on the context and the application of the learning agent. A physical environment is preferred when the exploration is not dangerous, whereas simulator when the real agent can hurt itself or damage environment. The drawbacks of learning in a physical system reside mainly in the fact that exploration is expensive and constrained to states that are safe and reachable. A learning agent is often a robot, composed of mechanical parts, which may perform a range of movements that can be harmful - to itself, to other people, to the environment itself if not behaving in an intended manner. Think about a car hitting a wall, or with a person aboard. Simulators, on the other hand, are safe: everything is happening inside a computer, in a virtual world, and everything - or almost - can be properly tuned before putting the code on the physical robot. Another downside of learning in the real world is expensiveness: machine learning, relying on statistics, requires vast amount of data to be reliable, which in RL context translates into a big number of repetitions of an experiment. In a physical environment the components are typically more expensive than on a simulator, which requires AC current to power one or more machines. A robot requires one or more engines, which can be electrical (consuming much more than computers) or combustion-like, which require some kind of fuel, like gasoline. Moreover, wear comes into play, which requires fixing, or replacing, for instance the wear of tyres due by friction with asphalt. Moreover, possible repairs are required when the robot crashes or hurts itself or the environment must be taken into ac-

count. Another advantage in using a simulated environment is the simplicity of its usage: in a **real world** scenario, one would need to deal with sensors, data processing, and physical actuators, which might require spending a vast amount of time in managing hardware and software aspects (e.g. hardware requirements, input-output compatibility between sensors, processing unit and actuators, errors and uncertainties of measures). Moreover, some simulators provide high-level data, which makes it possible to skip the information extraction stages, **which** could be challenging. For instance, The Open Racing Car Simulator (TORCS) **provides** telemetry data for the cars, including speed and angle of steering, which otherwise should be computed with some kind of image analysis, such as image segmentation via supervised learning through a neural network. **Which** is not a trivial task at all. Whether **simulators** may be preferred for the reasons listed above, they have drawbacks, too. A simulated environment must replicate the world and an agent to a certain degree of accuracy: replicating a perfect copy would be unfeasible, because of the power of computation required to run it, and the intrinsic complexity of the rules which govern it, which today are known to a certain extent. On the other side, a simple representation of the world is much more lean to run, and simple to be coded, but may be undermodeling with respect to the task. The main difficulty here is finding the best trade off. Another aspect to take into consideration when choosing to learn on a simulated environment is that the computed parameters of the **actions** ruling the world and the actions to fulfill a task may be different **to** the real ones, thus **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. For instance, wear of components and deformation dynamics are ruled by complex models which are hard to replicate on simulators. As said, the choice between real environment and a simulated one mainly depends on the context and the application of the learning agent. When exploration is not dangerous, real world environments are preferred whereas simulator must be used when the real agent can hurt itself or damage environment. RL is particularly suitable to contexts **that prevent to perform** many tries of a particular task, because **learning** algorithms enable an agent to learn from its experience by giving a representation of the environment without an a priori knowledge **of** its dynamics. For this reason, videogames and computer simulators are **adapted** to run a reinforcement learning algorithm on. A notorious videogame played by a RL autonomous agent is Atari [16] by Google's Deepmind. By learning a deep convolutional neural network to approximate the Q-function, Mnih et al. successfully construct a Deep Reinforcement Learning framework called Deep Q-Network (DQN)

which plays Atari games at human level. It takes as representation of the world raw pixels. It learns an end-to-end policy which maximizes the q-values. Another famous example is Alphago also by Deepmind [21], which beated for several years consecutively the human champion at Go game. This is not a videogame but a board game, however the case is noteworthy, because such game is known for being one of the most complex for his huge set of rules, which makes brute-force approaches infeasible. To achieve this result, it has been trained with a supervised learning neural network from past experience, and then, a reinforcement learning algorithms has been applied to try and beat itself's own play. In racing games, which is the scope of our thesis, different solutions have been proposed over the years. In the next section we will introduce the concept of autonomous driving and in particular in the racing context. We will expose the problem of finding the optimal trajectory and explore some of the state-of-the-art techniques which tackle this issue. In our thesis we chose to use a simulator and in particular we worked with TORCS.

4.2.2 Autonomous Driving in Videogames

Finding a racing line that allows to achieve a competitive lap-time is a key problem in real-world car racing 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 lap-time possible on a given track with a given car. The optimal racing line is the path that a driver should follow to complete a lap on a given track in the smallest amount of time possible. As the lap-time depends both on the distance raced and on the average racing speed, finding the optimal racing involves two different sub-problems: racing the shortest distance possible and racing as fast as possible along the track.

Over the years, different attempts to achieve time-optimal racing have been made, evolving together with technology. Besides Reinforcement Learning, controlling a self-driving car can be done with a planner - if the optimal trajectory is computed apriori - or by a controller. In this section, we provide an overview of such techniques.

Botta et al. [3] show how to encode a racing line by a set of connected Bézier curves, such that each one defines a small portion of the racing line. Bézier curves are a family of parametrized curves widely used in computer graphics and in related fields. Initially used for drawing cars, nowadays Bézier curves are frequently used in vector graphics to model smooth paths, as they offer a compact and convenient representation. A Bézier curve is

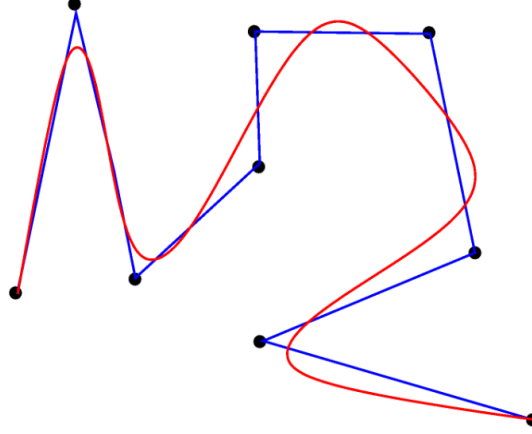


Figure 2: From [3]. An example of Bézier curve: black points are the control point of the curve; blue lines simply connect with straight segments the control points; red line is the resulting Bézier curve.

defined by a set of control points, with the first and the last one being respectively the beginning and the end of the curve, while the intermediate control points do not usually lie on it. Therefore, the evolution is responsible of the entire design of the racing line. Figure 2 shows an example of Bézier curve defined by a set of 9 control points. In addition, the authors compare two different methods to evaluate the evolved racing line; the first one is based on testing the evolved racing lines in a racing simulator; the second one consists of estimating the performance of a racing line through a computational model. The former consists in making a TORCS predefined controller follow a determined trajectory. Then, a fitness function is computed as the lap-time achieved during the best lap. While this approach does not require any previous domain knowledge, as the fitness is the result of simulation, it is also rather expensive in computational terms as it requires a full simulation. On the contrary, the latter relies on a computational model which provides an estimate of the lap-time that could be achieved following the racing line to evaluate. In particular, as soon as an estimate of the lap-time is available, the fitness function is computed as in the simulation-based method. This way, the computation is generally significantly less expensive than the simulation-based evaluation. As a drawback, a previous domain knowledge is required in order to create a model able to estimate the speed that can be reached in each point of the racing line. Moreover, the accuracy of the estimation

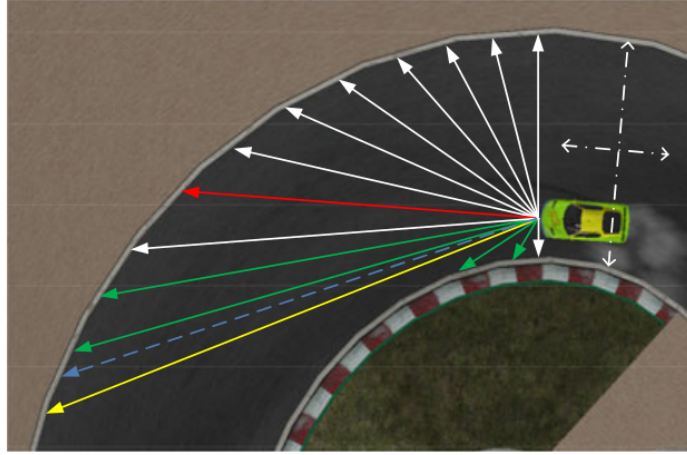


Figure 3: From [2]. Red vector, the yellow vector, green vectors, and the blue vector represent the zero sensor, the base sensor, auxiliary sensors, and the final angle to take the turn, respectively.

depends on the precision of the model itself.

Another approach has been proposed by Bonyadi et al. in [2]. They implemented a controller called Ahura for TORCS based on heuristics. The controller uses five modules:

- **Steer controller:** This module uses the estimated angle of the turn in front and the vacant distance in front to determine the steer angle. The module can control how smooth or sharp the vehicle is going to turn. The main idea behind calculation of the steer angle is to find the proximity sensor that has the maximum empty space in front (called the base sensor). The angle of the base sensor, together with some other auxiliary proximity sensors (as shown in Figure 3) is then used to set the angle of the steer.
- **Speed controller:** The aim of the speed controller is to determine the speed of the vehicle according to the current situation without considering opponents. This decision then is translated to the acceleration/breaking pedals. This module uses the estimated turn angle together with the vacant distance in front to decide the safe speed. The target speed is mapped with a nonlinear function to values of acceleration and brake. Then, it is necessary to make sure that these values are applied in an appropriate way. During the brake, the spin of the wheels might become smaller than the speed of the vehicle.

means the wheels are slipping. Also, during the acceleration, the spin of the wheels might increase more than the speed of the vehicle. It means the vehicle is in traction. The ABS and ASR technologies solve these issues through keeping the speed of the spinning wheels and the speed of vehicle as close as possible. Moreover, a gearing system is implemented, based on the minimum and maximum values of the rpm for each gear;

- Opponent manager: This module creates a map of opponents around and finds the vacant slot to overtake. This action may entail modification of the speed and steer calculated by the speed and steer controller modules. Ahura's opponent manager contains two modules, steer reviser and speed reviser, that are responsible to revise steer and speed for overtaking. The information about opponents provided by opponent's sensors contain their distance and angle from the current position of the vehicle. This means that the position of opponents is provided in a polar coordinates system with the center of the measuring the vehicle. Ahura builds a spatial map of the position of opponents, which is used to revise steer and speed of the vehicle for overtaking purposes;
- Dynamic adjuster: This module uses the mechanical specifications of the track (friction, bumps) as well as recorded difficulties the controller has experienced during the earlier laps and adjusts the current driving style. The dynamics parameters are estimated using an optimization-based approach to find the best values so that Ahura can drive a specific bot in TORCS. There are 23 parameters in Ahura that need to be determined: eight parameters for the steer controller, ten parameters for the speed controller, five parameters for the opponent manager. They were determined by using the optimization algorithm CMA-ES [9], which has a good performance in continuous space, works with nonlinear systems, no constraint handling technique is required, and it is appropriate for nonseparable search spaces;
- Stuck manager: This module controls the vehicle when it is out of the track or it has stuck somewhere. The calculated parameters for the speed and steer controllers need to be revised based on the specifications of the track. The reason is that the parameters have been set for a limited number of tracks while new tracks might have different specifications. The main characteristics of tracks that may affect the best choice for parameters are friction, width, and bumps. Also, Ahura is

able to handle stuck situation. In fact, if Ahura recognizes the car is off the track, it reduces the maximum value of the acceleration pedal to prevent too much traction. It finds the correct direction first and tries to get back to the track. If it detects that the car is not moving, the gear is changed to -1 (rear gear) and the steer is adjusted accordingly to repair the direction of movement.

Existing advanced control based attempts to minimum-time driving usually provide only offline open-loop solutions. In [27], the authors experimentally compared different time-optimal nonlinear MPC [28] formulations based on least squares objectives with an economic cost function in a real-time setup of small-scale model race cars, with the help of ACADO simulation toolkit from MATLAB [14]. MPC, standing for Model Predictive Control is a multivariable control algorithm that uses an internal dynamic model of the process, a cost function over the receding horizon and an optimization algorithm minimizing the cost function using the control input. The tight real-time bounds imposed on computational times make it necessary to reformulate the problem so as to allow for the use of efficient algorithms. To this end, they employed a nonlinear bicycle model [12] in a reformulation to spatial coordinates and proposed an infeasible-time-tracking objective for the best practical results. In the literature, bicycle models are often adopted because, compared to higher fidelity vehicle models, system identification is easier, being there only two parameters to identify, l_f and l_r . For an approximate solution of the time-optimal driving problem, the aim is to minimize the time required for the race car to reach the end of the fixed-length spatial prediction horizon. For prediction horizons which tend to infinity this objective tends to the goal of driving time-optimally. As a consequence, long horizons are expected to yield a good approximation of the original problem in practice. In the author's formulation, by providing a sufficiently small (i.e., infeasible) "target time" T , they can have an approximate time-optimal MPC formulation in least-squares form. The performance of the car with this formulation is shown in Figure 4.2.3 for several laps, where it can be seen that the cost function allows the car to deviate from the centerline in order to minimize the time tracking error. The real-world experiments showed that this approach has potential, but the bicycle model proved insufficient at high velocities due to slip effects not being modelled.

4.2.3 Reinforcement Learning in Racing Videogames

With the evolution of technologies and algorithms, the focus has been shifting towards reinforcement learning and general data-driven algorithms,

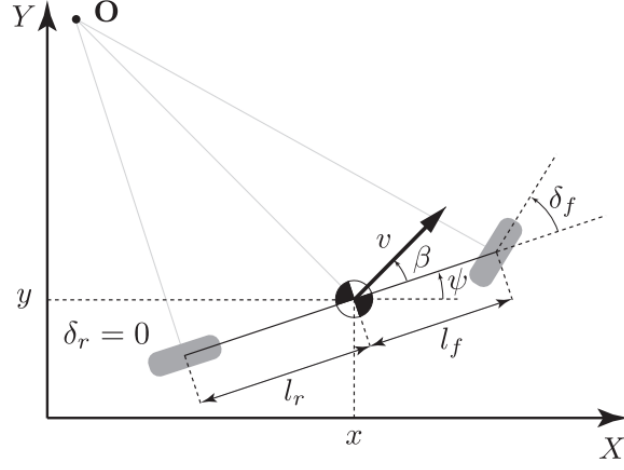


Figure 4: Automatic Bicycle Model: x and y are the coordinates of the center of mass in an inertial frame (X, Y) . ψ is the inertial heading and v is the speed of the vehicle. l_f and l_r represent the distance from the center of the mass of the vehicle to the front and rear axles, respectively. β is the angle of the current velocity of the center of mass with respect to the longitudinal axis of the car. δ_f and δ_r are respectively the front and rear steering. Since in most vehicles the rear wheels cannot be steered, typically δ_r is assumed equal to 0.

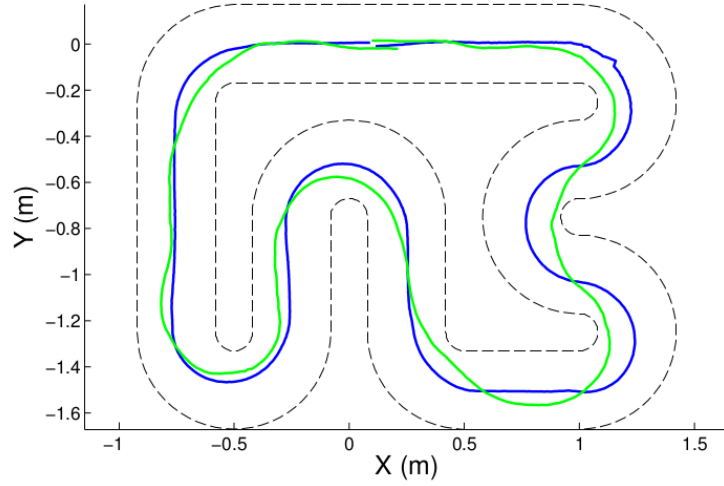


Figure 5: From [27]. Comparison of the performance of trajectory tracking and time-optimal driving on the experimental setup.

that has been proven to outperform **classical algorithms** in a variety of applications, including autonomous driving and racing.

In [19] the authors tackle the problem by exploring different formulations of the DDPG algorithm [13] in TORCS. Here, telemetry data are used in order to make an agent learn to drive and outperform the standard bots included in the simulator. To achieve that, they provide the critic network of DDPG with a Long Short Term Memory [11], making the network recurrent, and enabling the agent to exploit the knowledge of the past. To this end, they also make the algorithm consider a window of multiple states and multiple reward instead of single ones. While in the standard algorithm a one-step target is used for updating the critic function, they adopt a multi-step targets strategy, which incorporate the next n rewards obtained along the trajectory starting from state s_t and following a policy close to the current policy π at time step t . **As compared to** the multiple states strategy, called Window sampling, the intuition is that in partially observable environments, accessing a single state does not reveal the full underlying state of the environment at each time step. Window sampling provides the agent additional information by feeding a window of the last w states to the actor and the critic network. Finally, they substitute the uniform distribution used by DDPG to sample the transition from the replay buffer with a Prioritized Experience Replay, which attempts to make learning more efficient by sampling more frequently transitions that are more important for learning. Their study was split in three parts, by training three different models, in order to enhance the generalization capability. The first one uses a simple track and trained for 500 episodes. The learned model was tested without exploration on the same track and the results were used to select the hyperparameters for each of the algorithms used. In the second one the algorithm has been tested on a technically more complex track, while in the last part they evaluated the impact of adding future information of the track by adding a look ahead curvature and training in all three tracks.

4.2.4 Imitation Learning in Videogames

A promising approach in building human-like artificial players is what is commonly known as Imitation Learning **as proposed in** [8], or Learning from Demonstration (LfD) [1], which consists in integrating some human expertise in the algorithm.

A popular algorithm called DAGGER [20] provides an easy way for incorporating human experience. However, a human expert should always be available to provide feedback which is **not convenient** in practice. Finn

et al. proposed the Inverse Reinforcement Learning (IRL) [7], which automatically learns reward functions from human demonstrations for robotic control. Since it does not provide a mechanism to obtain feedback from the interaction with the environment, the generalization ability of imitation learning is limited.

Presented in [10], Deep Q-learning from Demonstrations (DQfD) vastly accelerates DQN by pretraining an initial behavior network, and also introducing a supervised loss and a L2 regularization loss when training the target network.

In [5] the authors considered rewards computed by a predictor trained with human feedback rather than from interactions with the environment. This method separates the goal learning from the behavior learning by training a reward predictor with non-expert human preferences, and the behavior of the agent is improved accordingly.

4.2.5 Imitation Learning in Racing Videogames

Togelius and colleagues focused on the learning of a specific driving style using a simple 2D car simulator [26], [24] and TORCS [25]. In [26], the authors applied supervised learning (more precisely, neural networks and k-nearest neighbor [6]) to model driving styles directly, that is, by building models that could accurately predict the player actions based on the current game state. However, direct modeling resulted limited performance, accordingly, they moved to indirect modeling and used a genetic algorithm to evolve a controller with a driving style similar to a target human player. In [24], the approach based on indirect modeling was improved by using a different fitness function while later, in [25], it was extended by introducing a multi-objective evolutionary algorithm to evolve controllers which could be both robust (in that, they never run out of the track) and could demonstrate a driving style similar to one recorded from a user (in that, they accurately predict the user actions).

In [4], Cardamone, Loiacono and Lanzi applied supervised learning to develop car controllers in TORCS from the logs collected from other driver, and developed controllers capable of driving in non-trivial tracks reaching a performance that is in some cases 15% lower than that of the fastest bot available in the simulator (a good result if considered that supervised learning approaches generally yield poor performances). They considered two representations of the current state of the car:

- The first sensory representation is based on a rangefinder inputs usually employed in simulated car racing competitions. The rangefinder

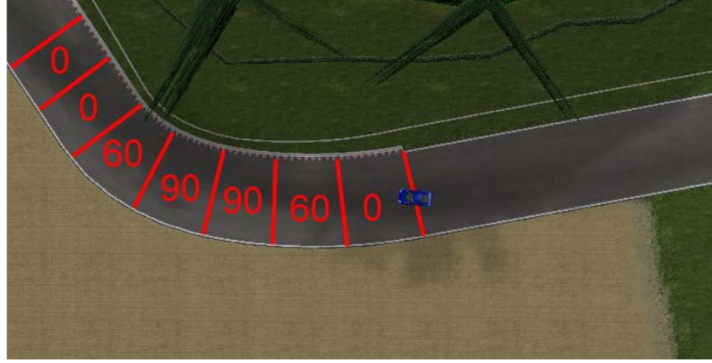





Figure 6: From [4]. Lookahead Sensor.

casts an array of beams, with a range of 100m, around the car and returns the distance from the car to the track edge;

- A high-level, qualitative, representation involving basic lookahead information about the track in front of the car, based on a novel sensor model, called lookahead sensor . This representation is inspired by the behavior of human drivers whose decisions are usually based on high-level information about the current speed and trajectory and on the shape for the track ahead. For this purpose, the next h meters of the track ahead are considered and divided into segments of the same length, and the lookahead sensor returns the bending radius for each segment: a rectilinear segment corresponds to a 0; a positive radius corresponds to a right turn; while a negative radius corresponds to a left turn.

Regarding the control phase, rather than acting on steer, throttle and break pedal, they propose to control the car at a higher level,  they try to learn the trajectories and the speeds along the track rather than low level commands. In other words, instead of predicting the typical low-level actions on the car actuators available in TORCS, their approach is to predict a target speed and the car position with respect to the track axis. A trajectory can be represented as the distance from the track axis in each segment of the track. So the output variables of the controller are the speed and the axis distance in the current segment  of. The control is realized by means of three simple algorithms:

- Algorithm 1 computes the steer value: it describes the simple policy that controls the steering command to reach the target distance from

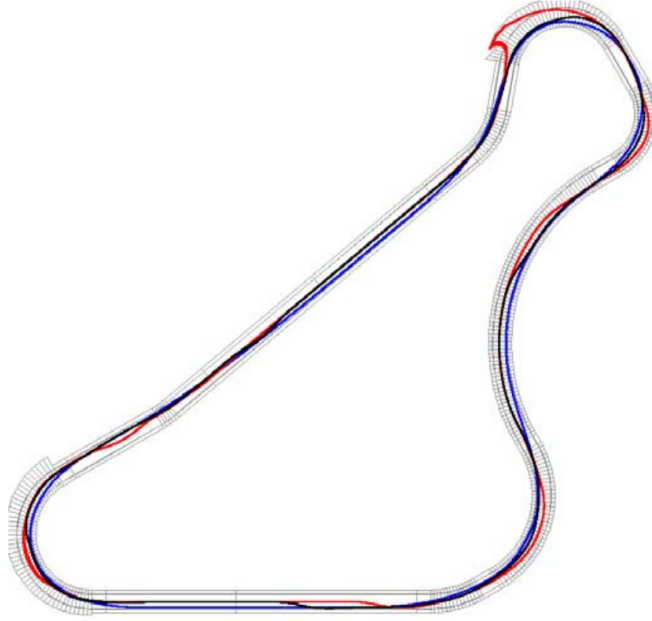


Figure 7: From [4]. Trajectories comparison in track G-track-1: bot Inferno (blue line), 8 segments lookahead (red line), 16 segments lookahead (black line).

the track axis. First, the wheel is aligned to the car axis and then a correction is applied to reduce the error;

- Algorithms 2 computes the acceleration value;
- Algorithms 3 computes the brake value. Algorithms 2 and 3 work this way: simply accelerate at full speed when the current speed is low and brake when is too high.

A soft transition is finally applied between brake and accelerator when the actual speed is near to the target speed [6] this way switching from full acceleration to full brake is avoided, resulting in no fluctuations around the target speed.

Finally, as concerns the learning phase, they considered two supervised learning methods, k-nearest neighbor classifiers [6] and multi-layer neural networks [22] and applied them to compute a model mapping input sensors to actions which could imitate the behavior of an observed driver. The model was then deployed to a driver controller that would map the high-level actions predicted by the models to low-level actions of the usual TORCS effectors using the algorithms discussed in the previous section.

The k-nearest neighbour does not involve any training in that the collected data represent the model. Accordingly, it has been directly applied during the evaluation, i.e. during a race using TORCS. At each game tic, the logged data are searched to find the k most similar instances to the current sensory input using either the typical rangefinder representation or the new lookahead representation. The k similar instances are selected and the respective outputs variable are averaged to predict the target value. The similarity measure used is simply the Manhattan distance among instances; **Neural Networks**. To avoid the issue of selecting a particular neural network structure, they applied Neuroevolution with Augmenting Topology (NEAT) [22] which applies a genetic algorithm to evolve both the weights and the topology of a neural network. They evolved two separate neural networks, one to predict the target speed and one to predict the target position for a given input configuration. The fitness was defined as the prediction error computed as the total sum of the absolute error between the true value and the predicted value, for each instance in the training data. Their experiment consisted in running the Inferno bot on three tracks: G-track-1, a simple track; Wheel-1, a more difficult track with many fast turns; and Aalborg, a difficult track with many slow sharp turns. They logged the data from the rangefinders and from the lookahead sensors, together with the car distance from the track axis and the car speed which we use as outputs. The learned trajectories are shown in

This paper [29] aims at improving the achievements of DQfD [10] in integrating reinforcement learning with human demonstrations using TORCS. Their goal is to bring experts demonstrations together with reinforcement learning, try to let the agent learn an appropriate policy by interacting with the environment on itself, and learn from expert's demonstrations at the same time. Their **contribution** develops in three parts:

- They shift from the discrete action domain **considered** in [10] to a continuous action domain. In order to determine the similarity between actions, they approximately evaluate the two considered actions by their mean squared error. As DQN uses a greedy way to find the optimal policy, and the actions are finite discrete values, it is easy to calculate the supervised error for DQfD. However, in continuous action domain, the greedy policy is inapplicable, so they try to use the output of actor network directly for supervision. They combine a supervised loss with the original TD-loss to update the parameters of the critic network, and control the weighting between the losses by a given parameter;

- They adjust the Experience Replay of DQN and DDPG by constructing an integrated replay buffer, which improves DDPG stability. In DQN, the authors use a replay buffer to store the transitions generated by interacting with the environment, and randomly sample batches for training. In DQfD, two replay buffers are utilized to store the self-generated data and demonstrator’s data respectively. Training data is sampled from these two buffers by a certain proportion. In this paper, the authors added another buffer to store the self-generated training data with good performances. For the beginning episodes, which usually do not contain enough good data, they instead use another buffer to collect the best transitions in every training episode, and substitute for the good performance buffer temporally. Their claim is that this procedure may improve the performance and stability of training;
- They attempt to learn human preferences without contradicting to the overall target of the task. Most RL methods have only one overall goal, but do not consider the human behavior in realizing the goal, which can be reflected from the slightly differences between human personal preferences in completing the same task. They tackle this problem by recording demonstrations with human preferences and define specific optimization objectives to ensure the consistence with human preferences. To do that, they use demonstration data to train the critic network’s input. The actor network yields an action, and the actor network yields an action the must be similar enough with the demonstrator’s action. Meanwhile, this action needs to satisfy the Bellman Equation and to yield higher rewards when interacting with the environment. So, they use a combined loss to update the critic network’s parameters.

Their claim is that their method not only masters the preferences in choosing the lane, but also outperforms human expert demonstration in average reward.

Our work takes place in this context of imitation learning integrated with reinforcement learning applied to autonomous driving in TORCS without using sensory information. In the next section, we illustrate the software architecture and thereafter we describe techniques we used.

Chapter 5

Software Architecture

Chapter 6

Methods

Chapter 7

Experiments

TESTO CAPITOLO 7

Chapter 8

Conclusion

TESTO CAPITOLO 8

Bibliography

- [1] Aude G. Billard, Sylvain Calinon, and Rüdiger Dillmann. *Learning from Humans*, pages 1995–2014. Springer International Publishing, Cham, 2016.
- [2] M. R. Bonyadi, Z. Michalewicz, S. Nallaperuma, and F. Neumann. Ahura: A heuristic-based racer for the open racing car simulator. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(3):290–304, 2017.
- [3] M. Botta, V. Gautieri, D. Loiacono, and P. L. Lanzi. Evolving the optimal racing line in a high-end racing game. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 108–115, 2012.
- [4] L. Cardamone, D. Loiacono, and P. L. Lanzi. Learning drivers for torcs through imitation using supervised methods. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 148–155, 2009.
- [5] Paul Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. 06 2017.
- [6] Padraig Cunningham and Sarah Delany. k-nearest neighbour classifiers. *Mult Classif Syst*, 04 2007.
- [7] Abbeel Finn, Levine. Guided cost learning: Deep inverse optimal control via policy optimization. *International Conference on Machine Learning (ICML)*, 2016.
- [8] B. Gorman, C. Thureau, C. Bauckhage, and M. Humphrys. Bayesian imitation of human behavior in interactive computer games. In *18th International Conference on Pattern Recognition (ICPR’06)*, volume 1, pages 1244–1247, 2006.

- [9] N. Hansen, S. D. Müller, and P. Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, 11(1):1–18, 2003.
- [10] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning. *ArXiv*, abs/1704.03732, 2017.
- [11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.
- [12] Georg Schildbach Jason Kong, Mark Pfeiffer. Kinematic and dynamic vehicle models for autonomous driving control design.
- [13] Alexander Pritzel Lillicrap, Hunt. Continuous control with deep reinforcement learning. *International Conference on Learning Representations (ICLR)*, 2016.
- [14] MATLAB. Acado toolkit. <https://acado.github.io/documentation.html>.
- [15] Alberto Maria Metelli, Matteo Papini, Francesco Faccio, and Marcello Restelli. Policy optimization via importance sampling. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 5442–5454. Curran Associates, Inc., 2018.
- [16] Silver Mnih, Kavukcuoglu. Playing atari with deep reinforcement learning.
- [17] Takayuki Osa, Joni Pajarinen, and Gerhard Neumann. *An Algorithmic Perspective on Imitation Learning*. Now Publishers Inc., Hanover, MA, USA, 2018.
- [18] Steffen Priesterjahn, Oliver Kramer, Alexander Weimer, and Andreas Goebels. Evolution of reactive rules in multi player computer games based on imitation. pages 744–755, 08 2005.
- [19] Adrian Remonda, Sarah Krebs, Eduardo Veas, Granit Luzhnica, and Roman Kern. Formula rl: Deep reinforcement learning for autonomous racing using telemetry data. 06 2019.

- [20] Stephane Ross, Geoffrey Gordon, and J. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. *Journal of Machine Learning Research - Proceedings Track*, 15, 11 2010.
- [21] Maddison Silver, Huang. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.
- [22] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.
- [23] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.
- [24] Julian Togelius, Renzo de nardi, and Simon Lucas. Towards automatic personalised content creation for racing games. pages 252 – 259, 05 2007.
- [25] N. van Hoorn, J. Togelius, D. Wierstra, and J. Schmidhuber. Robust player imitation using multiobjective evolution. In *2009 IEEE Congress on Evolutionary Computation*, pages 652–659, 2009.
- [26] Niels Van Hoorn, Julian Togelius, Daan Wierstra, and Jürgen Schmidhuber. Robust player imitation using multiobjective evolution. In *Proceedings of the Eleventh Conference on Congress on Evolutionary Computation, CEC’09*, page 652–659. IEEE Press, 2009.
- [27] Robin Verschueren, Stijn Bruyne, Mario Zanon, Janick Frasch, and Moritz Diehl. Towards time-optimal race car driving using nonlinear mpc in real-time. volume 2015, 12 2014.
- [28] Yu-Geng XI, Dewei Li, and Shu Lin. Model predictive control - status and challenges. *Acta Automatica Sinica*, 39, 03 2013.
- [29] Zhu Zuo, Wang. Continuous reinforcement learning from human demonstrations with integrated experience replay for autonomous driving. *International Conference on Robotics and Biometrics*, 2017.