POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Design and development of an autonomous agent for The Open Race Car Simulator using Deep Reinforcement Learning

Relatore: Ing. Daniele Loiacono

Tesi di Laurea Magistrale di:
Alessandro Giani
Matricola n. 875803

Anno Accademico 2018-2019

Abstract

This work studies the design and development of an autonomous agent in the context of racing-games using deep learning. As the deep learning techniques have progressed throughout the years more successful agents have been developed in games scenarios. Our work aims at being a first step toward the design and the development of an effective agent using a deep deterministic policy gradient algorithm to race in The Open Race Car Simulator (TORCS) a racing simulator that throughout the years has become the standard software for academic research. We focused on designing three different types of networks that use a different representation of the input to represent the environment state. The first network uses a numerical input representation already proved to be successful in performing deep learning on racing-games. For the second network we developed a custom method to generate images at each time-step of the experiment and use them as the state representation. The third network combines the two inputs described above by using both the images and numerical data to build the input of the neural network. After training these models in different scenarios we analyzed and compared the results obtained during the training experiments and also when the model was tested on unseen tracks of the game. Our results shows that the network using only images as input struggles to reach the performance of the other networks and form this results we make our conclusion and present the open problems and future work to be done on this topic.

Estratto in lingua Italiana

Questo lavoro studia il design e lo sviluppo di un agente autonomo che sfrutta le tecniche di deep learning nel contesto dei giochi di guida. Nel corso degli anni i progressi del deep learning hanno reso queste tecniche molto promettenti in questo contesto. Il nostro lavoro costituisce un punto di partenza per lo sviluppo di un agent che usa un algoritmo di deep deterministic policy gradient per The Open Race Car Simulator (TORCS) un simulatore di guida che nel corso degli anni diventato lo standard per ricerche accademiche di questo tipo. Ci siamo focalizzati nel design di tre diverse reti ognuna delle quali usa una differente rappresentazione dello stato dell'ambiente di learning. La prima rete usa un input numerico che gi stato provato ottenere buoni risultati nel applicazione del deep learning nei giochi di guida. Per la seconda rete abbiamo sviluppato un metodo custom per generare durante ogni step del processo di learning un immagine della pista che verr usata come rappresentazione dello stato. La terza rete usa un insieme dei due input descritti precedentemente e usa sia le immagini sia dei dati numerici per creare l'input della rete. Dopo aver concluso il processo di training della rete in diversi scenari abbiamo analizzato e confrontato i risultati ottenuti dalle tre reti durante il training e anche quando il modello generato dalle reti stato testato su piste su cui non aveva mai girato. I risultati ottenuti mostrano che la rete che usa solo immagini come input non raggiunge i risultati ottenuti dalle altre due reti e da questo abbiamo tratto le nostre conclusioni e presentato i possibili sviluppi futuri in questo ambito.

Contents

Abstract **3**

Estratto in lingua Italiana **5**

1 Introduction **13**
- 1.1 Scope of the thesis . . . 13
- 1.2 Thesis structure . . . 14
- 1.3 Original contribution . . . 14

2 State of the art **15**
- 2.1 Reinforcement Learning . . . 15
  - 2.1.1 Markov Decision Processes . . . 16
  - 2.1.2 Environment . . . 16
  - 2.1.3 Rewards/Goal . . . 16
  - 2.1.4 Policy . . . 17
  - 2.1.5 Value Function and Q-values . . . 17
  - 2.1.6 Temporal Difference Learning . . . 18
  - 2.1.7 SARSA (On-Policy) . . . 18
  - 2.1.8 Q-Learning (Off-Policy) . . . 19
  - 2.1.9 Exploration vs Exploitation . . . 20
  - 2.1.10 Epsilon-Greed Strategy Selection and Boltzmann exploration . . . 20
  - 2.1.11 On-Policy vs Off-Policy . . . 20
  - 2.1.12 Function Approximation . . . 21
- 2.2 Deep Reinforcement Learning . . . 21
  - 2.2.1 Convolutional Neural Networks . . . 22
  - 2.2.2 Deep Q Network . . . 23
  - 2.2.3 Deep Deterministic Policy Gradient . . . 24
- 2.3 Related Work . . . 26
  - 2.3.1 Deep Reinforcement Learning in Games . . . 26
  - 2.3.2 Deep Reinforcement Learning in Racing Games . . . 27

3 Software Architecture **29**
- 3.1 TORCS and the SnakeOil client . . . 29
- 3.2 Simulated Car Racing Competition . . . 31
- 3.3 OpenAI gym . . . 33
  - 3.3.1 TensorFlow . . . 34

4 Deep Reinforcment Learning in TORCS **37**
- 4.1 Reinforcement Learning problem design . . . 37
  - 4.1.1 Input/Output . . . 37
  - 4.1.2 Exploration Strategy . . . 40
  - 4.1.3 Reward Function . . . 41
  - 4.1.4 Gradient Inverting . . . 42

4.1.5 Episode stopping criteria 42
4.2 Neural Networks Structure 45
4.2.1 Numeric network 45
4.2.2 Image Network 45
4.2.3 Mixed network 46

5 Experiment Results 51
5.1 Training results 51
5.1.1 Numeric Network 52
5.1.2 Image Network 53
5.1.3 Image Network steer only 54
5.1.4 Mixed Network 54
5.2 Testing results 65
5.2.1 Numeric Network 65
5.2.2 Image Network 66
5.2.3 Mixed Network 66

6 Conclusions and Future Work 75
6.1 Future Work 76

List of Figures

2.1 Example of a neural network with one hidden layer 22
2.3 Example of a neural network with one hidden layer 25

3.1 Schematic view of the software architecture used to perform our experiments . . 29
3.2 Torcs Example 30
3.3 Torcs Example 31
3.4 Visualizing the operation of a CNN in TensorBoard. 35

4.1 Torcs Example 38
4.2 The first 3000x3000 image build containing the overview of the Brondehach track. 39
4.3 The overview of the track after applying the rotation according to the agent direction. 40
4.4 Final images obtained when the agent is on a straight or when the agent is approaching a curve 40
4.5 Torcs Example 40
4.6 Reward Function 41
4.7 The experiment design scheme with the major operation performed at each step of each episode. Ellipsis are used when there is a direct communication with the TORCS server. 43
4.8 The Actor and Critic network scheme of the Numeric Network 47
4.9 The Actor and Critic network scheme of the Image Network 48
4.10 The Actor and Critic network scheme of the Mixed Network 49

5.1 Overview of the five tracks used during the training process 52
5.2 Reward and distance obtained by the Numeric Network; above when racing on five tracks, below when racing on a single track with random start 55
5.3 Set of graphs representing the acceleration, steering angle and track position recorded by the Numeric Network when racing on five tracks 56
5.4 Set of graphs representing the acceleration, steering angle and track position recorded by the Numeric Network when racing on one track 57
5.5 Reward and distance obtained by the Image Network; above when racing on five tracks, below when racing on a single track with random start 58
5.6 Set of graphs representing the acceleration, steering angle and track position recorded by the Image Network when racing on five tracks. 59
5.7 Set of graphs representing the acceleration, steering angle and track position recorded by the Image Network when racing on one track. 60
5.8 Reward and distance raced obtained by the Image Network controlling only the steering angle when racing on one track 61
5.9 Steering and track position of the Image Network controlling only the steering angle when racing on one track 61
5.10 Reward and distance obtained by the Mixed Network; above when racing on five tracks, below when racing on a single track with random start 62

5.11 Set of graphs representing the acceleration, steering angle and track position recorded by the Mixed Network when racing on five tracks. 63
5.12 Set of graphs representing the acceleration, steering angle and track position recorded by the Mixed Network when racing on one track. 64
5.13 Overview of the three tracks used during the testing process 65
5.14 Trajectory of the Numeric Network agent trained on five tracks on the Mueda-city track 67
5.15 Trajectory of the Numeric Network agent trained on one track on the ustka-city track 68
5.16 Trajectory of the Image Network agent trained on five tracks on the petit track . 69
5.17 Trajectory of the Image Network agent trained on one track on the Mueda-city track 70
5.18 Trajectory of the Mixed Network agent trained on five tracks on the petit track . 71
5.19 Trajectory of the Mixed Network agent trained on one track on the Mueda-city track 72

List of Tables

3.1 Description of the available sensors (part I). Ranges are reported with their unit of measure (where defined). 33
3.2 Description of the available effectors. 33

4.1 Parameters chosen according to [2] for the Ornstien-Uhlenbeck noise 41
4.2 Hyperparameters used by the three networks 44

5.1 Testing results obtained by our three network on the testing tracks together with the results obtained by the SnakeOil client. In bold the best distances achieved for each track 73

Chapter 1

Introduction

During the years the technology concerning video-games artificial intelligence has improved significantly. In this scenario machine learning played a critical role in the development of better and better autonomous agents capable of producing performances comparable, or even better, than human control. Starting from the discoveries made by Google DeepMind Group [1] that, using deep reinforcement learning methods, were able to develop an agent able to play different Atari 2600 games with the same neural network structure and reward system, the interest concerning deep learning and its application to gaming agents is higher and higher.
In particular, deep learning is a promising approach to improve AI in racing games. In fact, although modern racing games generally feature good AI, there are three major open problems: (i) the AI is usually not able to deal with very different types of vehicles and typically drives using a simplified physics and model of the vehicle; (ii) the AI still struggles to effectively overtake or lap the opponents, as well as to manage properly group behaviors, like the race start; (iii) the computer controlled drivers are very similar and usually do not exhibit different driving styles. Besides driving the cars, a better AI in racing games could have several applications, such as assisting the players, teaching how to drive properly, evaluating the car setup, evaluate the design of cars and tracks, etc.
The applications of these discoveries become even more important if applied to racing games. These games can be used as the foundation to test and develop artificial intelligence able to control autonomous driving cars without the risk of testing them in the real world.
Thanks to the new interest in deep learning techniques applied to racing games, a lot of new methodologies to apply learning have been discovered. Among those the one presented in [2] that uses deep deterministic policy gradient is the more promising when looking at the results obtained and is the main inspiration for this work.

1.1 Scope of the thesis

In this work, we evaluate the applicability of a deep deterministic policy gradient algorithm to generate an autonomous agent able to race on TORCS by directly controlling the acceleration and the steering angle of the racing car. We develop three agents each of them using a different input as state representation of the learning environment. At first, we produce and test an agent having a numerical state representation; then, we explore the possibility of having a top-down image of the track segment where the agent is as the input of the network. This differs from the classical first-person view state representation usually used when applying machine learning to create video-games artificial intelligence. We also develop a third network that combines both the image mentioned before and some numerical data to create the input for the neural network. At the end of the work, we also test the models produced to evaluate the generalization capability of our agents.

1.2 Thesis structure

Chapter 2 contains the theory needed to understand how a deep reinforcement learning experiment is designed and the related work done prior to this thesis.
Chapter 3 describes the software architecture we used to perform the deep learning experiment and how this framework communicates and is integrated with our game of choice.
Chapter 4 shows the design process and the decisions made when setting-up our experiment. It illustrates the problem definition and the three network structures used to perform the experiments proposed in this work.
Chapter 5 contains the results obtained during the training process and also the results the developed agents obtained during testing. Chapter 6 collects our conclusions about the experiments done while also presenting the future works that can be done on this topic.

1.3 Original contribution

- We developed, trained and tested a deep reinforcement learning agent capable of racing on TORCS using a top-down representation of the track instead of a first-person view as the state representation of the deep learning experiment;

- We applied deep reinforcement learning to learn from scratch driving behaviors in TORCS, using as input both image-based and telemetry-based representations of the racing state; we also evaluated the learned driving behaviors on unseen racing track and compared them to a human programmed behavior.

Chapter 2

State of the art

In this chapter, we will introduce the theoretical aspects on which our work is based on thus giving the reader some knowledge about the machine learning techniques used in our system. In Section 2.1 we will explain what is a reinforcement learning experiment, starting from the concept of *Markov Decision Process* which is the base of most machine learning experiments. We will explain all the key aspects going into detail about what is the environment of an experiment, the reward/goal functions that the agent gets and the policy it develops and various techniques to do so.
In the following sections we will then give the reader the base to understand the concepts of *Deep Reinforcement Learning* starting from its fundamentals and then describing two of the many algorithms of this category, the *Deep Q-Networks* and the *Deep Deterministic Policy Gradient* that is the main algorithm we will use in this thesis. Obviously, there are many more algorithms that can be described but it is not the scope of this thesis to discuss them and if the reader is interested later will present some suggested material.
We will also describe the concept of *Convolutional Neural Networks* that are also used in this thesis to process images coming from the game. We will explain the concepts of filters and strides of the network to give the reader a better understanding of how a convolutional network works.
Finally, in the last section of this chapter, we will present examples of how reinforcement learning has been successfully used in the past in the scope of AI applied in video-games focusing on racing games that are the ones used in this thesis.

2.1 Reinforcement Learning

Reinforcement learning is a method that has been developed to resolve decision-making problems, it has been inspired by behavioral psychology [3], and works by using a different reward over time. Differing from other machine learning methods, with reinforcement learning the agent is not told the proper actions to take at each time-step. Alternatively, the agent explores the environment in which it is placed to maximize the total sum of future rewards (or the highest sum of expected rewards), usually with the goal (target space) to reach a terminal state which is represented numerically by a high reward. Since the agent is performing actions in the environment and learning at the same time, new and better methods needed to be examined, differing from classical Supervised Learning methods where the designers of the experiment directly sample interactions and apply statistical patterns. These methods such as Adaptive Heuristic Critic algorithms [4], culminating with the formulation of Q-learning [5] will be explored later in this chapter.

2.1.1 Markov Decision Processes

We need to formally describe in a mathematically way a common framework to compare real-world tasks. It is useful to recall the *Markov Property* [6] and the definition of *Markov Decision Process* (MDP) [7].
The standard Reinforcement Learning set-up can be described as an MDP consisting of:

- **A finite set of states** D, comprising all possible representations of the environment.
- **A finite set of actions** A, that contains all possible actions that an agent can perform at any given time.
- **A reward function** $r = \psi(s_t, a_t, s_{t+1})$, that determines the reward obtained by performing action a_t in the state s_t resulting in state s_{t+1}
- **A transition model** $T(s_t, a_t, s_{t+1}) = p(s_{t+1}|s_t, a_t)$, that indicates the probability of a transition between states s_t and s_{t+1} given the action a_t

It has been shown that in every state of the environment the agent can perform an action that will result in a new state, and the agent is expected to progressively collect more reward with each episode of learning so the agent has to solve the problem of choosing an appropriate action for every state of the environment in which it ends up being.

2.1.2 Environment

The environment is the mathematical representation of the world where the agent is placed. In the case of this thesis where the agent is placed in a virtual racing track, the environment can vary from a vector of sensor reading coming from the game or it can be an image of the portion of the track where the agent is placed or even a combination of both.
A state of an environment can be represented by a space of dimension contained within the state space of an environment D, e.g.,

$$s_t \subset \mathbf{D}, \quad s_t \in R^N \tag{2.1}$$

where s_t is some state at time t and **D** the state-space that represent the world of the agent. The dimension of the state R^N is dependent on the problem. This dimension is a key factor in setting up an experiment of Reinforcement Learning. For each problem, there can be multiple ways of representation and identifying the key features that help represent a useful state for the agent can greatly improve the learning aspect.
For example, in the environment considered in this thesis putting in the state representation, the current lap might not be as useful as the speed at which the agent is currently moving. Even when considering a state represented by a 2D image which image to use, the resolution or the number of channels used to represent the image can impact the learning process.

2.1.3 Rewards/Goal

In a Reinforcement Learning framework, an agent learns by reinforcement (as in psychology). If the agent gets a sequence of positive rewards this will lead to a good policy while if it gets a negative reward this will lead to a bad policy. As studied in [8], our agent aims to maximize the accumulated sum of rewards:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{2.2}$$

where R_t is the reward that the agent gets at a particular time step up to T. In general, for stochastic environments, the goal is to maximize the expected value of the return within a task. Eq. 2.2 can be re-written as follows:

$$R_t = \sum_{k=0}^{T} r_t + k + 1 \tag{2.3}$$

However, a future reward may have a different impact on the present. This rate is referred to in the literature as discount factor. A discount factor can be considered as a learning parameter, in the range of $0 \leq \gamma \leq 1$. Every future reward at time t is discounted by γ^{k+1}. The parameter γ changes how the agent considers future or present rewards. If γ is near to zero, the agent considers the rewards near the present state more important and valuable, while if γ is close to one, the agent behaves greedily and considers the future rewards that can get as important as the present ones. The return for a specific policy can then simply written as:

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{2.4}$$

2.1.4 Policy

The behavior of the agent is expressed as a control policy that is a function that maps what action to take at a given state. A policy can be deterministic if the action chosen will be executed for sure, or it can be stochastic if the execution of the action is not guaranteed. Therefore a deterministic policy can be represented as:

$$\pi(s_t) = a_t, \quad s_t \subset D \;\; a_t \subset A \tag{2.5}$$

while if the action are stochastic the policy becomes a probability distribution of a_t given s_t:

$$\pi(a_t|s_t) = p_i, \quad s_t \subset D \;\; a_t \subset A, \;\; 0 \leq p_i \leq 1 \tag{2.6}$$

The policy that generates the most reward in a given environment it is called *optimal policy* and is indicated with π^*, and after applying a proper learning algorithm with a good enough exploration strategy until convergence the policy obtained by the agent can be considered "optimal" or "sub-optimal" as described in *Bellman's Principle of Optimality* [9].

2.1.5 Value Function and Q-values

To evaluate how useful a policy is a value function can be calculated. The value function given a state $s_t \in D$ and following the same policy π gives a value that describes how useful that policy is. The value function can be formulated using the discounted sum of rewards described before as Eq. 2.3

$$V \; : V \; \to \mathbb{R}, \quad V^\pi(s) = \mathbb{E}_\pi \left\{ R_t | s_t = s \right\} = \mathbb{E}_\pi \left\{ \sum_{i=0}^{\infty} \gamma^i r_{t+i+1|s_t=s} \right\} \tag{2.7}$$

Since the value function depends on experience samples V^π can be estimated by "trial-and-error" methods. This property allows the unraveling of the value function to be calculated recursively and thus benefit form dynamic programming properties. If we unravel Eq. 2.7, given a certain state the value function is [8]:

$$V^\pi(s) = \mathbb{E}_\pi \left\{ \sum_{i=0}^{\infty} \gamma^i r_{t+i+1|s_t=s} \right\} = \mathbb{E}_\pi \left\{ r_{t+1} + \sum_{i=0}^{\infty} \gamma^i r_{t+i+2|s_t=s} \right\} \tag{2.8}$$

If instead the policy is stochastic the Eq. 2.8 will unravel into the *Bellman equation* of V^π [8]:

$$\begin{aligned} V^\pi(s) &= \sum_a \pi(a|s) \sum_{s_{t+1}} p(s_{t+1}|s_t, a) \left[r(s, a, s_{t+1}) + \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_{t+1} = s_{t+1} \right] \right] \\ &= \sum_a \pi(a|s) \sum_{s_{t+1}} p(s_{t+1}|s_t, a) \left[r(s, a, s_{t+1}) + \gamma V^\pi(s_{t+1}) \right]. \end{aligned} \tag{2.9}$$

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. [8]:

$$Q : S \times A \to \mathbb{R} \qquad Q(s, a) = \mathbb{E}_\pi \left\{ R_t | s_t = s, a_t = a \right\} = \mathbb{E}_\pi \left\{ \sum_{i=0}^{\infty} \gamma^i r_{t+1+i|s_t=s, a_t=a} \right\} \tag{2.10}$$

If we consider an optimal policy π^*, the value of a state $V^{\pi^*}(s_t)$ is equal to the Q-value $Q^{\pi^*}(s_t, a_t)$ when the optimal action is taken [8]:

$$s_t \subset D \quad a_t \subset A \qquad V^{\pi^*}(s_t) = Q^{\pi^*}(s_t, a_t) = \arg\max_a Q^\pi(s_t, a_t) \tag{2.11}$$

2.1.6 Temporal Difference Learning

Temporal difference learning (TD) [8] algorithms are able to learn from raw experiences without the need for a model of the environment. They combine ideas form 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 Mote 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 form dynamic-programming, but while DP needs a precise model of the environment dynamics TD does not and is so more suitable for unpredictable tasks.

2.1.7 SARSA (On-Policy)

This particular TD algorithm gives its name to the tuple needed to perform an update to the Q-value function (State, Action, Reward, Next State, Next Action) and the update works as follow:

$$Q : S \times A \to \mathbb{R} \tag{2.12}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) \tag{2.13}$$

The α in Eq. 2.13 is the *learning rate* $()0 < \alpha \leq 1)$ and indicates how quickly new information will override old Q-values, while γ is a discount factor $(0 < \gamma \leq 1)$which decreases the estimated Q-values for future states When the agent interacts with the environment and performs a certain action the Q value is gradually updated with a learning rate α, so the SARSA algorithm will continue to move towards an optimal policy as long as the state-action pairs are being updated. The pseudo-code for the SARSA algorithm as found in literature [8] is presented in Algorithm 1.

Algorithm 1 SARSA-learning

```
1. procedure On-Policy
2.     Initialize $Q(s,a)$ arbitrarily for all $a \in A$ and $s \in S$
3.     Repeat until the end of the episode:
4.     $s_t \leftarrow$ Initial State
5.     Select $a_t$ based on a exploration strategy from $s_t$
6.     for each episode step do
7.         Take action $a_t$, observe $r_{t+1}$, $s_{t+1}$
8.         Select $a_{t+1}$ based on a exploration strategy from $s_{t+1}$
9.         $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right)$
10.        $s_t \leftarrow s_{t+1}$
11.        $a_t \leftarrow a_{t+1}$
12.        if $s ==$ terminal then
13.            $Q(s_{t+1},a_{t+1}) = 0$
14.        end if
15.    end for
```

2.1.8 Q-Learning (Off-Policy)

Another TD algorithm is the one-step Q-learning by Watkins [5].
This algorithm is considered off-policy because the optimal Q-value function Q^* is approximated not considering the current policy.
This algorithm is very similar to Sec. 2.1.7 but when we update the Q-value we use the maximum future value instead of the all experience tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. The update is as follows:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right) \tag{2.14}$$

Algorithm 2 Q-learning (Off-Policy)

```
1. procedure Q-Learning
2.     Initialize $Q(s,a) = 0$ $a \in A$ and $s \in S$
3.     Repeat until the end of the episode:
4.     $s_t \leftarrow$ Initial State
5.     for each episode step do
6.         Select $a_t$ based on a exploration strategy from $s_t$
7.         Take action $a_t$, observe $r_{t+1}$, $s_{t+1}$
8.
           $$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right) \tag{2.15}$$
9.         $s_t \leftarrow s_{t+1}$
10.        if $s ==$ terminal then
11.            $Q(s_{t+1},a_{t+1}) = 0$
12.        end if
13.    end for
```

2.1.9 Exploration vs Exploitation

A recurrent dilemma in AI and in Reinforcement Learning especially is the exploration vs exploitation dilemma. When is it better to stop exploring new states and start exploiting the acquired knowledge to get a higher reward?
Balancing this drastically improves our agent performance. In the best case scenario, our agents firstly explore the largest number of possible states and then when it is confident or have deeply explored the environment it starts exploiting gathered knowledge to acquire better results.
This problem is even more complex when we consider uncertain environments, but in the next section, some methods to select an action through a quality function will be presented.

2.1.10 Epsilon-Greed Strategy Selection and Boltzmann exploration

A simple method to choose an action at each time step is given by the ϵ-greedy strategy selection. Given a Q-function $Q(s, a)$, the best action is selected with probability $(1 - \epsilon)$ while we choose a random action from the action set with probability ϵ. Since it represents a probability the parameter ϵ is set to be between zero and one $(0 \leq \epsilon \leq 1)$.
A way yo decrease the value of ϵ is needed because otherwise the agent will be stuck exploring the environment and would never exploit the gained knowledge. Ideally, it would be best a high exploration at the beginning of the learning experiment while a more exploitative strategy later on in the experiment. This strategy is called in literature [8] as a ϵ-greedy decreasing strategy. To show the ϵ decrease a variety of family of function can be used, usually putting ϵ_0 with a value close to one to have almost full exploration at the beginning, and then chosen a discount factor ξ can be computed to obtain dr_ξ. Finally, a new ϵ is calculated after each iteration of the algorithm with the following formula:

$$\epsilon_t = \frac{\epsilon_0}{1 + t dr_\xi} \tag{2.16}$$

One of the main problems with ϵ greedy strategy is that it considers only one action as the best possible one while it labels all the others as bad actions, and when we draw a bad action the corresponding Q-value is discarded possibly losing on potential exploitation gains. This problem intensifies even more if the distance between Q-values is very high and the second-best action is highly penalized.
When using the Boltzmann distribution (thus the name Boltzmann exploration [10]) every $Q(s_t, a_t)$ is used to compute the following formula:

$$p(a_t|s_t, Q_t) = \frac{\exp^{Q(s_t,a_t)/\tau}}{\sum_{b \in A_D} \exp^{Q(s_t,b)/\tau}} \tag{2.17}$$

where τ is the key component to switch between exploration and exploitation. If τ presents high values the numerator is pushed to one giving more importance to exploration, while when τ decreases with time, the exploration becomes more greedy and gives a higher probability to the best actions to be chosen (instead of choosing only the best action as with the ϵ greedy strategy).

2.1.11 On-Policy vs Off-Policy

Both methods have their merit the difference is that On-Policy methods use the action chosen to update the action-value function while Off-Policy methods choose a different policy to update the same action-value function.
The main problem remains how the action is selected and the exploration vs exploitation dilemma discussed before. If we use Off-Policy methods such as Q-learning it will discard the weight of a bad move to converge to the optimal policy, while On-policy methods perform with a ϵ-greedy strategy with the action selected immediately introduced in the update step.

2.1.12 Function Approximation

The main problem that arises when applying Reinforcement Learning algorithms is that to apply these algorithms to real-world tasks we need to consider a set o problems:

1. The world is not discrete and it will often be needed to deal with continuum times and spaces
2. When elaborating a state the designer introduces a personal bias in some parameters of the environment
3. In dynamic environments, the dimensions of the states could become unfeasible to represent due to memory or time limitation

The last point described is also known as *curse of dimensionality* and it became a halting problem for reinforcement learning, because when the dimensionality and the sparsity of the data increase it is harder to find a pattern.
In the next section we will introduce *Deep Learning* and the *Deep Reinforcement Learning* methods that are used to solve the problem of curse of dimensionality introduced here. It is also the method we will use in this thesis to control a car in a racing game.

2.2 Deep Reinforcement Learning

In this section of the thesis, we will give a brief introduction on Deep Reinforcement Learning and its fundamentals. As said before we will cover only on two algorithms in this thesis and we will not discuss the other types of network configuration, but if the reader is interested we suggest reading [11] to have a broader view on the argument.
In particular we will describe two main algorithms: Deep Q-networks (DQN) which was successfully used by Google Deep Mind to train an actor able to play different types of Atari games and the Deep Deterministic Policy Gradient (DDPG) algorithm that is the algorithm that will be used to perform the learning processes of this thesis experiment.
One of the advantages of deep learning and a characteristic that helped boost its popularity is the fact that these algorithms seem to overcame the problem of the curse of dimensionality only by their inherent nature. There are various theories why this is the case but we will not cover them here. The reader can look up these articles if interested [12]. Now we will go into more detail on the structure that makes a deep neural network.

A Deep neural network is characterized by a succession of multiple processing layers. With every layer, there is a different non-linear transformation and each of these produces a different level of abstraction. [13, 14]. First, we will describe a very simple neural network with one fully-connected hidden layer as it can be seen in Figure 2.1.

The input of the first layer is a column vector x of size $n_x(x_n \in \mathbb{N})$. To calculate the values of the next hidden layer a non-linear parametric function transformation is applied to the input. This consists of a matrix multiplication W_1 of size $n_h \times n_x(n_h \in \mathbb{N})$, plus a bias term b_1 and the a non-linear transformation:

$$h = A(W_1 \cdot x + b_1) \tag{2.18}$$

where A is what is called an *activation function*, this is what causes the non-linearity to propagate throughout the network. Each hidden layer h can in turn be transformed to other sets of values up to the last layer which is the output layer y where:

$$y = (W_2 \cdot h + b_2) \tag{2.19}$$

where W_2 is of size $(n_y \times n_h)$ and b_2 is of size $n_y(n_y \in \mathbb{N})$.

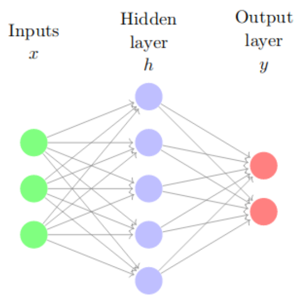


Figure 2.1: Example of a neural network with one hidden layer

All the layers of the network are trained to minimize a specific error $I_S[f]$. The most common way to update the network parameters is to use gradient descent with the backpropagation algorithm [15]. In the simplest case, the network changes it's parameters θ to better fit the desired error function:

$$\theta \leftarrow \theta - \alpha \nabla_\theta I_S[f] \tag{2.20}$$

Nowadays there are many different types of neural network structures beyond the simple feed-forward network described before, with each variation providing different advantages depending on the scope of the network. Moreover, within each network, it is possible to have an arbitrary amount of layers with some application reaching the hundred in number of hidden layers [16]. In this thesis experiment, apart from the already mentioned feed-forward layer, another type of layer able to perform image feature extraction is used for the experiments and will be described in the next section of this chapter. I will not cover however all the other possible types of network configuration but there is plenty bibliography on the subject if the reader is interested [11].

2.2.1 Convolutional Neural Networks

As said before convolution layers [17] are particularly well suited to process images; this is due to their translation invariance property. A convolutional layer takes as input some feature maps and then outputs n feature maps.
The parameters of each layer consists of a set of filters (also called kernels) that have a small receptive field and perform a convolution (hence the name) operation on the input and then they pass the results to the next layer.
There are four main hyperparameters to discuss when talking about convolutional layers:

- the filter height and width: these parameters represent the dimensions of the window that slides on the input map and generate the convolution. As the filter is slid over the width and height of the input volumes it produced 2-dimensional activation maps that give the responses of the filter at every spatial position. The network will then have specific filters that activate only when they see some type of feature in the input volume. In Figure 2.2b we can see the mathematical representation of a filter;
- the depth n of the output volume: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first convolutional layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. A set of neurons that are all looking at the same region of the input is usually referred to as depth column;
- the stride: this refers to how much the filter is moved (or convoluted) on the input image. For example, when the stride is 1 then the filters are moved one pixel at a time, while when the stride is 2 then the filters jump 2 pixels at a time as they are sled around. This will

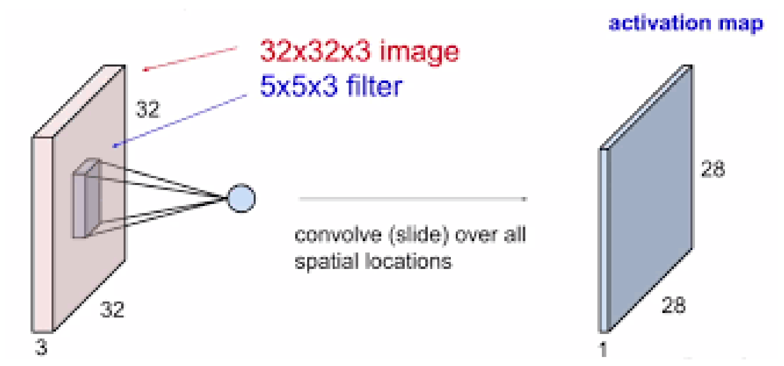
produce smaller output volumes spatially;

- the padding: sometimes it is convenient to pad the input volume with zeros around the border. The nice feature of zero padding is that it allows us to control the spatial size of the output volumes (most commonly it is used to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

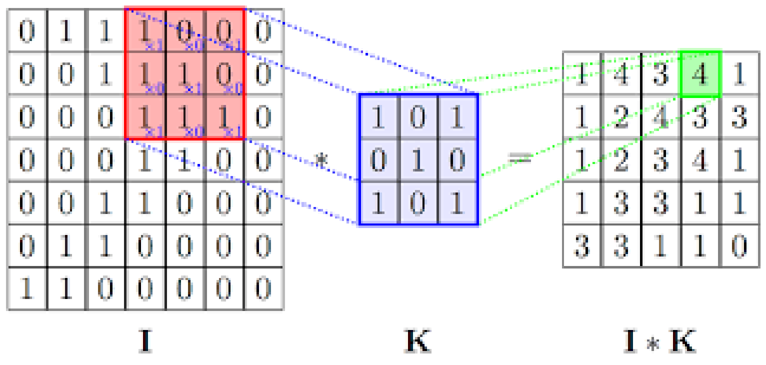
Having said so we can now compute the spatial size of the output volume a as a function of the input volume size **(W)**, the receptive field size of the convolutional layer neurons **(F)**, the stride with which they are applied **(S)**, and the amount of zero padding used **(P)** on the border:

$$O = (W - F + 2P)/S + 1 \tag{2.21}$$

In Figure 2.2a we can see the graphical representation of a simple convolutional layer.



(a) Application of one convolutional layer with n filters of size $5 \times 5 \times 3$ on a $32 \times 32 \times 3$ image



(b) Mathematical representation of a 3×3 filter

2.2.2 Deep Q Network

The DQN algorithm has a great history when solving AI problems in games. It has been used by Google Deep Mind to train an agent to play various ATARI titles learning directly from pixels. Since it is a Q-learning algorithm it follows the principles described in Section 2.1.8
The simple settings described before are inapplicable when considering a high-dimensional state-action space so the value function needs to be parametrized: $Q(s, a; \theta)$. This algorithm relies on two concepts that are used to perform the learning paradigm: **experience replay** and **target**

network.
To perform experience replay the agent experience made by the tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ is stored in a buffer B, during the training Q-learning updates are performed on samples (also called mini-batches) of experience drawn uniformly at random by the buffer of stored experiences. This is done to remove correlation in the observation sequence that can cause Q-learning algorithm to diverge.
Target networks are also introduced, it is called target because when updating the loss function we are trying to change the Q-values to be close to these targets. The problem is that the target depends on the same parameters that the "real" network has thus made the minimization of the loss unstable. To avoid this the parameters of the target network are updated using a time delay so that they are similar but not exactly the same. In DQN these target values are updated by copying the parameters of the real network every some-fixed number of steps. The reason to update the target parameters only every some-fixed number of step is to avoid to propagate instabilities to the target networks thus reducing the divergence of the target values:

$$Y^{Q}_{target} = r + \gamma \max_{a^{'} \in A} Q(s^{'}, a^{'}; \theta_{target}) \tag{2.22}$$

$$L_{DQN} = (Q(s, a; \theta_k) - Y^{Q}_{taget})^2 \tag{2.23}$$

2.2.3 Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG for short) algorithm is the main technique used to perform the learning in our environment. This algorithm extends DQN in the case that it can be used to overcome the restriction of discrete action. It uses the same methods of DQN such as experience replay and target network and also relies on the use of an actor and critic network that will be discussed hereafter. [18]
At its core, DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy that is easier to learn. These types of policy gradient algorithms use a form of policy iteration, firstly they evaluate the policy following a policy gradient to maximize performance. Since DDPG is an off-policy algorithm and it uses a deterministic target it allows the use of the Deterministic Policy Gradient theorem.
It also uses the method introduced before called actor-critic: it uses two neural networks, one built for the actor and one for the critic. These networks compute the action for the current state and generate a temporal difference error-value at each time step. The input for the actor network is the current state and the output is a single real value chosen from a range of continuous values. The critic instead uses both the state and the action provided by the actor and produces an estimate of the current Q-value for the state-action pair. The weights of the actor are then updated by the Deterministic policy gradient theorem, while the critic weights are updated with the gradients provided by TD error signal.
In Figure 2.3 we provide a basic scheme of the actor-critic structure.

The algorithm then uses the two concepts already introduced and discussed for the DQN algorithm: experience replay and target networks with the only difference being the update rule for the target network parameters that are not only copied at every some-fixed time step but are updated following this rule:

$$\phi_{target} \leftarrow \rho\phi + (1 - \rho)\phi_{target} \tag{2.24}$$

where ρ is a hyperparameter between 0 and 1 (usually close to 1) and ϕ_{target} are the weights of the target network and ϕ are the weights of the network.
We can then update the formulas of the DQN with these new techniques introduced:

$$y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\phi^{\mu}_{target})|\phi^{Q}_{target}) \tag{2.25}$$

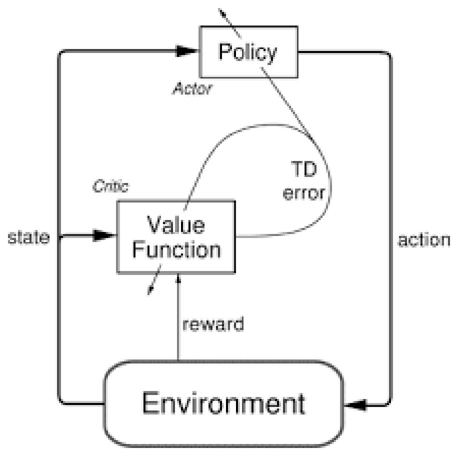


Figure 2.3: Example of a neural network with one hidden layer

$$L_{DPPG} = \frac{1}{N}\sum_i((y_i - Q(s_i, a_i|\phi^Q))^2) \tag{2.26}$$

With these formulas we indicates that a minibatch of size N has been pulled from the experience replay buffer with the **i** index referencing the i-th sample. The target for the temporal difference error computation, y_i is computed from the sum of the immediate reward, r_i and the outputs of the target actor (μ') and critic network (Q') having respectively weights ϕ^{μ}_{target} and ϕ^{Q}_{target}. Then the critic loss can be computed with regard to the output of the critic network for the i-th sample $Q(s_i, a_i|\phi^Q)$
To see more details of target network usage see [1]. As mentioned before the weights of the critic network can be updated with the gradients from the loss function, while the actor network weights are updated by the Deterministic Policy Gradient. It has been proved in [19] that the stochastic policy gradient $\nabla_{\phi^\mu}\mu \approx$, which is the gradient of the policy performance is equivalent to the deterministic policy gradient given by the equation:

$$\mathbb{E}_{\mu'}[\nabla_a Q(s, a|\phi^Q)|_{s=s_t, a=\mu(s_t)}\nabla_{\phi^\mu}\mu(s|\phi^\mu)|_{s=s_t}] \tag{2.27}$$

This shows that the policy term in the expectation is not a distribution over the actions but it turns out that all that is needed is the gradient of the critic network w.r.t the action multiplied by the gradient of the output of the actor network w.r.t its parameters. The derivation of the theorem will not be presented in this work but can be found in [19] if the reader wants.
To also help the reader having a better understanding of this algorithm, which is the main focus of the thesis a pseudo-algorithm of the DDPG process is shown below.

Algorithm 3 3 Deep Deterministic Policy Gradient

```
1. Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer **D**
2. set target parameters equal to main parameters $\theta_{target} \leftarrow \theta, \phi_{target} \leftarrow \phi$
3. repeat:
4.        Observe state s and select action a = clip($\mu_\theta$(s) + $\epsilon$, $a_{low}$, $a_{high}$), where $\epsilon \sim \mathbb{N}$
5.        Execute a in the environment
6.        Observe next state s', reward r, and done signal d to indicate whether s' is terminal
7.        Store (s,a,r,s',d) in replay buffer **D**
8.        If s' is terminal, reset environment state.
9.        If it's time to update then
10.             for however many updates do
11.                   Randomly samples a batch of transitions, $B = (s, a, r, s', d)$ from **D**
12.                   Compute targets using Eq. 2.25
                      $$y(r, s', d) = r + \gamma(1-d)Q'(s', \mu'(s')|\theta^{\mu}_{target})|\phi^{Q}_{target})$$
13.                   Update Q-function by one step of gradient descent using loss function of Eq.
2.26
                      $$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (y(r, s', d) - Q_\phi(s, a)^2)$$
14.                   Update policy by one step of gradient ascent using:
                      $$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$$
15.                   Update target network
                      $$\phi_{target} \leftarrow \rho\phi + (1-\rho)\phi_{target}$$
                      $$\theta_{target} \leftarrow \rho\theta + (1-\rho)\theta_{target}$$
16.             end for
17.        end if
18. until convergence
```

2.3 Related Work

2.3.1 Deep Reinforcement Learning in Games

Applying deep reinforcement learning to video games proved to be a promising research direction and in this section we provide a brief overview of some of the most notable works in the literature.

The popularity of these methods increased in the past few years, following two major successes in this field of research. The first one was achieved by Google DeepMind group which developed a single reinforcement learning agent able to play several Atari 2600 games on a human level [1]. They used an algorithm called Deep Q-Networks(DQN), which learns to estimate the Q-values (state-action value functions) of selecting each action from the current game state. Since the state-action value function is a sufficient representation of the agents policy, a game can be played by selecting the action with the maximum Q-value at each time-step. Despite learning policies from raw screen pixels to actions, these networks have been shown to achieve state-of-the-art performance on several Atari 2600 games. Even more interestingly the same network can be used in several tasks without any change and that the learning is end-to-end from

raw pixel values to Q-values without any need for handcrafted features or human intervention. We used the convolutional layer used to process the images to develop one of the networks of this thesis. Another achievement that sparked the interest towards deep learning was **AlphaGo** [20]. In this work, the authors developed a hybrid agent that managed to beat the world champion in the Chinese board game of Go. This method is considered hybrid because at first the agent was trained using supervised learning by studying recorded games, then, using deep learning methods, it strengthens its knowledge of the game by playing against himself.
In the next years, DQNs have been extended to improve their performance in complicated games. Nowadays recurrent Deep Q-Networks use a Long Short Term Memory (LSTM) together with deep Q-network [21]. This method allows the networks to handle partial observability of the environment, and using the inherent recurrence in LSTM allows the network to perform even when the quality of the observation changes during the evaluation time.
The main advantage of these recurrent DQNs is that they don't require multiple game frames as input to operate but they are able to use a single input frame and integrate information across multiple time-steps by using the LSTM layer.
More recently, Wang et. al. [22] proposed a new architecture called the Dueling Architecture DQN which explicitly separates the representation of state values and state-dependent action advantages. It consists of two streams that represent state-value function and action-advantage functions while sharing a common convolutional feature learning module. The two streams are combined using a special aggregating layer to produce an estimate of the Q-value. This architecture is the current state-of-the-art in playing Atari games and achieves better than human-level performance in most games.

On the basis of the above early works, in the recent years several works proved that deep reinforcement learning can be successfully applied to different types of games including First Person Shooters, as shown in the works of [23], [24], [25], Platformers as shown in [26], Real Time Strategy as [27] and [28] and racing games [29], [30], [31], [2].

2.3.2 Deep Reinforcement Learning in Racing Games

In this thesis we focus on deep reinforcement learning applied to racing games, thus we provide below some of the most relevant applications in this field.
Many different algorithms have been used and tested to develop an agent capable of driving around a track.
In [39] the authors develop and trained, using double Q-learning, an autonomous agent able to control the simulated car of the simple racing environment JavaScript Racer [32], creating an agent able to detect and take turns during the training process. A more complex environment is presented by the authors of [29] where a Mario Kart 64 autopilot able to recognize road features and correct over and under-steering using a convolutional neural network is presented.

The authors of [30] implement an A3C to develop an agent for end-to-end driving in World Rally Championship 6, producing an agent capable of learning significant features and exhibits good driving behavior while training on different and various tasks.
Similar to our work, since it uses the same game, is the work presented in [31]. The authors use recurrent neural networks with LSTM to develop an agent able to drive in TORCS.
In [2] the authors develop an agent capable of racing on TORCS using a DDPG performed by a fully connected neural network. The DDPG algorithm was chosen for the final implementation since it was proven that generally actor-critic algorithms manage to outperform value-based algorithms, both in terms of the time and the resources required for training the algorithm.
Following these results, we chose to implement the same algorithm to apply deep reinforcement learning as the starting point for the development of our work.

Chapter 3

Software Architecture

In this chapter of the thesis, we will describe the software used to perform the experiment.
In particular, we will firstly introduce the open-source game TORCS, a racing simulator on which we focus for our work since it is widely used for game research. We will then describe the OPENAI gym framework that gives us the reinforcement learning concepts used to describe our environment. We will also briefly mention the open source TensorFlow library that gives us the modules for building the learning algorithm we will use, and also the Tensorboard visualization tool that we will use to have an easier understanding of our experiment results.
These two parts are placed inside separated docker container and exchange information using a client-server architecture; in the TORCS container we have the server built based on the work done for the Simulated Car Racing Championship and in the other container we used an already built Python client called SnakeOil that will be better described later in the chapter. In Figure 3.1 we can see an overview of our software structure.
We will now describe more in detail the software components mentioned before.

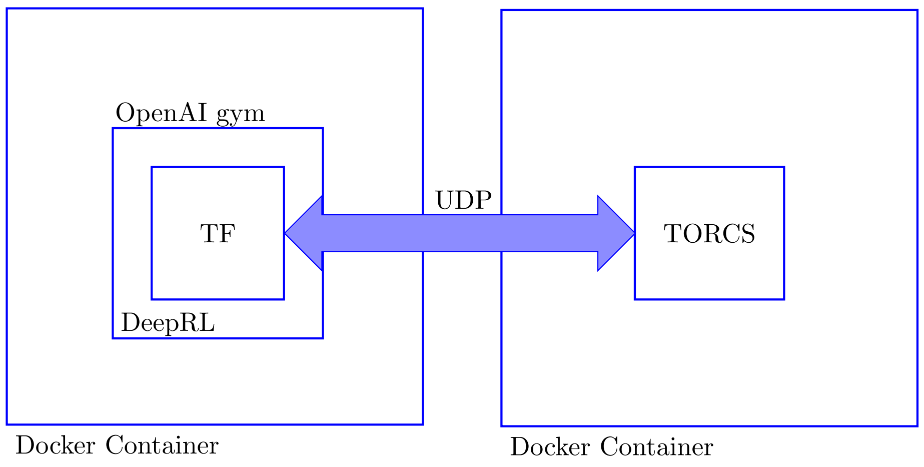


Figure 3.1: Schematic view of the software architecture used to perform our experiments

3.1 TORCS and the SnakeOil client

The Open Race Car Simulator (TORCS) [33], is an open-source 3D racing game based on OpenGL technologies. We chose to use TORCS because it is become widely used in academic research. This racing simulator, born from the idea of Eric Espi and Christophe Guionneau in 1997, is written in C++ and has allowed programmers during the years to develop robots able to drive on the track of the game. TORCS implements a very detailed physics engine that takes

into account many aspects of real car racing i.e., aerodynamics, wheel rotation, damage of the car, etc. Thus allowing the possibility to create very sophisticated agents. TORCS has been further developed through the years by a very wide community of users that have extended the game including the addition of some custom tracks that we will use in the experiments of this thesis. For this thesis, we will use TORCS version 1.3.1. This will be the game that our agent will interact with, and to do so it will use a client-server architecture better described in Section 3.2.
We used an already existing client interface called SnakeOil. This client's task is to set-up the UDP connection with the game server and to extrapolate the information coming from the server and put them in a comprehensible string for our agent to use. More details about the information received from the client will be provided in Section 3.2.
The SnakeOil client has also the aim of controlling some aspects of the car, i.e the clutch and the gear; these parameters are out of our scopes, so we decided to have them directly controlled by the client following a simple rule: based on the speed of the agent the client will automatically decide what is the right gear to shift to, obviously having higher gear corresponding to higher speed values.

In Figure 3.2 we can see a screenshot of the game taken at the start of the Forza map, while in Figure 3.3 we can see the structure of the client-server connection.

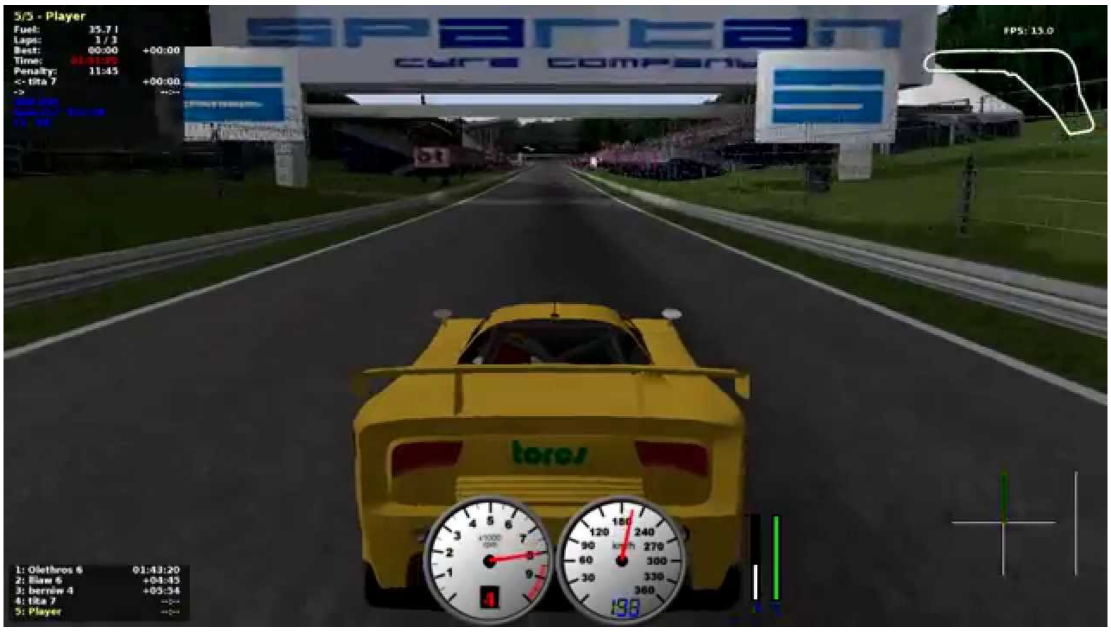


Figure 3.2: An example from a screenshot of TORCS.

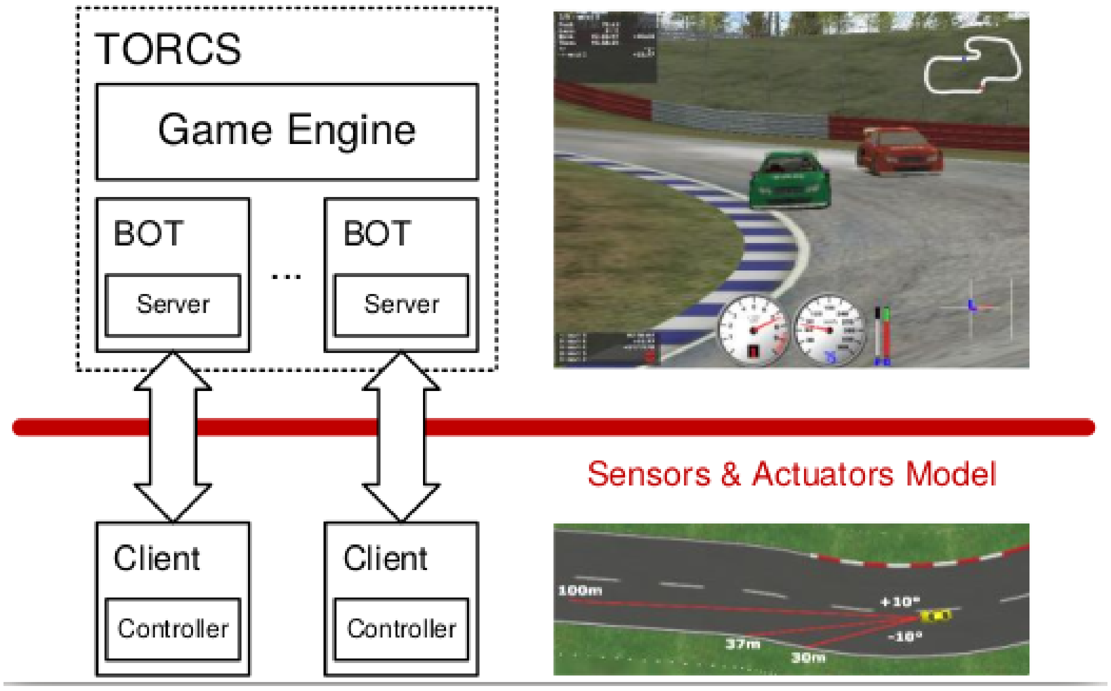


Figure 3.3: Client-server architecture.

3.2 Simulated Car Racing Competition

The Simulated Car Racing Competition (SCR) [34] is a scientific event where the aim is to develop an intelligent agent to compete in racing games. The competition had many editions, and for the work of this thesis, we will focus on the one concerning the use of TORCS. In this section we will focus on the API used to create a client-server connection between the agent and the game; however, we will not focus on how the connection is physically made (as said in Section 3.1 we use an already existing client to establish the UDP connection). Since the inputs the server needs to receive and the output the agent gives are the key components of our work, we will focus on them. The agent will get a list of sensor-readings from the server that reflect both the environment state surrounding the agent and the game state; on the other hand, the agent will operate on some actuators that will control basic commands to drive the car. The complete list of sensors that the server sends to the agent is provided in Table 3.1 and includes the current speed, the current gear, the fuel levels, etc. If the reader is interested in having a more in-depth comprehension of the SCR server we suggest a direct reading of the manual [35]. In Table 3.2 we present the actuators that the agent can modify: besides the typical racing commands (i.e, pushing the brake or the accelerator, etc.) there is also a meta-command that is useful to reset the race from the client-side.

Name	**Range (unit)**	**Description**
angle	[-π,+π] (rad)	Angle between the car direction and the direction of the track axis.
curLapTime	[0,+∞) (s)	Time elapsed during current lap.
damage	[0,+∞) (point)	Current damage of the car (the higher is the value the higher is the damage).
distFromStart	[0,+∞) (m)	Distance of the car from the start line along the track line.
distRaced	[0,+∞) (m)	Distance covered by the car from the beginning of the race

focus	[0,200] (m)	Vector of 5 range finder sensors: each sensor returns the distance between the track edge and the car within a range of 200 meters. When `noisy` option is enabled sensors are affected by i.i.d. normal noises with a standard deviation equal to the 1% of sensors range. The sensors sample, with a resolution of one degree, a five degree space along a specific direction provided by the client (the direction is defined with the *focus* command and must be in the range [-90,+90] degrees w.r.t. the car axis). Focus sensors are not always available: they can be used only once per second of simulated time. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), the focus direction is outside the allowed range ([-90,+90] degrees) or the sensors has been already used once in the last second, the returned values are not reliable (typically -1 is returned).
fuel	$[0,+\infty)$ (l)	Current fuel level.
gear	$\{-1,0,1,\cdots 6\}$	Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 6.
lastLapTime	$[0,+\infty)$ (s)	Time to complete the last lap
opponents	[0,200] (m)	Vector of 36 opponent sensors: each sensor covers a span of 10 degrees within a range of 200 meters and returns the distance of the closest opponent in the covered area. When `noisy` option is enabled, sensors are affected by i.i.d. normal noises with a standard deviation equal to the 2% of sensors range. The 36 sensors cover all the space around the car, spanning clockwise from -180 degrees up to +180 degrees with respect to the car axis.
racePos	$\{1,2,\cdots,N\}$	Position in the race with respect to other cars.
rpm	$[0,+\infty)$ (rpm)	Number of rotation per minute of the car engine.
speedX	$(-\infty,+\infty)$ (km/h)	Speed of the car along the longitudinal axis of the car.
speedY	$(-\infty,+\infty)$ (km/h)	Speed of the car along the transverse axis of the car.
speedZ	$(-\infty,+\infty)$ (km/h)	Speed of the car along the Z axis of the car.
track	[0,200] (m)	Vector of 19 range finder sensors: each sensors returns the distance between the track edge and the car within a range of 200 meters. Sensors are affected by i.i.d. normal noises with a standard deviation equal to the 5% of sensors range. The sensors sample the space in front of the car every 10 degrees, spanning clockwise from $+\pi/2$ up to $-\pi/2$ with respect to the car axis. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), the returned values are not reliable.
trackPos	$(-\infty,+\infty)$	Distance between the car and the track axis. The value is normalized w.r.t to the track width: it is 0 when the car is on the axis, -1 when the car is on the right edge of the track and +1 when it is on the left edge of the car. Values greater than 1 or smaller than -1 means that the car is outside of the track.

wheelSpinVel	[0,+∞] (rad/s)	Vector of 4 sensors representing the rotation speed of wheels.
z	[−∞,+∞] (m)	Distance of the car mass center from the surface of the track along the Z axis.

Table 3.1: Description of the available sensors (part I). Ranges are reported with their unit of measure (where defined).

Name	**Range**	**Description**
accel	[0,1]	Virtual gas pedal (0 means no gas, 1 full gas).
brake	[0,1]	Virtual brake pedal (0 means no brake, 1 full brake).
clutch	[0,1]	Virtual clutch pedal (0 means no clutch, 1 full clutch).
gear	-1,0,1,$\cdots$,6	Gear value.
steering	[-1,1]	Steering value: -1 and +1 means respectively full right and left, that corresponds to an angle of 0.366519 rad.
focus	[-90,90]	Focus direction (see the *focus* sensors in Table 3.1) in degrees.
meta	0,1	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race.

Table 3.2: Description of the available effectors.

3.3 OpenAI gym

This section provides an overview of the OpenAI gym toolkit for a better understanding, and it shows the advantages that it brings.
Through the years the OpenAI gym has been established as a great method to provide a standardization of reinforcement learning algorithms: it helps researchers have a similar framework for all their reinforcement learning algorithms.
This framework focuses on the episodic aspect of reinforcement learning experiments, that is to maximize a total reward fo each of the learning episodes and reach a good level of performance in the fastest way possible; to do so it focuses around the two main concepts of the Markov Decision Process discussed in Section 2.1.1, which are the agent and the environment.
OpenAI gym already provides different types of environments and various amount of reinforcement learning algorithms ready to be used to solve machine learning problems; however, it also provides the user with the possibility of defining new environments and new agents suited for their specific needs. It is the agent that describes the method of running the algorithm to solve the environment's task: this is what will be described later in this chapter together with the solution that has been chosen to reach the scope of this thesis.
There are two main methods that the unified interface class ENV needs to run:

- reset: This method resets the environment state to its initial state after the agent has reached a terminal state, and returns the initial observation to the agent.
- step: Performs the agent chosen time-step action in the environment and returns observation, reward, done, info.

It is import to better describe the returns of the step method because it gives the agent the substantial information to perform the reinforcement learning algorithm.

- observation: an object that represents the state in which the environment is after applying the time-step action chosen by the agent;
- reward: the reward the agent achieved by performing the action chosen;
- done: this value indicates to the agent whether it has reached or not a terminal state;
- info: a list of information that might be useful for debugging.

The step function is the fundamental method in every reinforcement learning environment: it is performed at each time step and has the task of applying the action chosen by the agent and calculating the corresponding reward. The DDPG algorithm used in this thesis is built using a custom OpenAI gym environment and a custom neural network.

3.3.1 TensorFlow

TensorFlow is an open-source software library that is used for performing numerical calculations using data flow graphs. It was developed by researchers from the Google Brain Group for machine learning and especially deep neural networks, but thanks to its versatility it is widely used in many other computational fields. Thanks to a flexible architecture, TensorFlow allows the user to perform the computing operation on multiple platforms using one or more CPUs (or GPUs), servers and more.
TensorFlow works by building a computational graph of which the reader can see an example in Figure 3.4: in this graph, the nodes represent mathematical operations performed in the network while edges indicate the core aspect of this framework, that are the tensors.
A tensor is a general representation of multidimensional data arrays, and as the name indicates, Tensorflow is a framework that bases its operation on running computation involving these tensors. Tensors can have multiple dimensions to indicate different types of data (i.e. a 1D-tensor represent a vector, while a 2D-tensor indicates a matrix).
Nodes represent math operations, and edges represent multidimensional data arrays, i.e., tensors, that relate to each other. Its flexible architecture allows you to expand computing on a variety of platforms, such as one or more CPUs (or GPUs), servers, mobile devices, and more in desktop computers. TensorFlow was originally developed by researchers and engineers from the Google Brain Group (part of the Google Institute for Machine Intelligence) for machine learning and deep neural networks, but its versatility makes it widely used in other computations field.

TensorBoard

Since the mathematical operations involving tensors done in a machine learning project are usually complicated and involve a large number of neurons that make them incomprehensible to be read by a human user, it is useful to use a visualization tool called Tesnorboard. This tool, that we will also use to provide graph and explain the training and testing result of our experiments in a later chapter, allows us to visualize graph of various variable of our training (i.e. the loss function or the reward progression during the training) to even more complex quantities (i.e. histograms of the activation of a single layer of the network) together with the visualization of the full data flow graph.
This process is made possible using a very useful Tensorflow utility called the Summary Writer, by which it is possible to save on disk serialized objects that contain the variable we want to track. These variables are written in an event file that Tensorboard can read even during a run of the experiment, and the user can visualize them by connecting to the Tensorboard host through a web browser.
In Figure 3.4 the reader can see an example of a full data flow graph generated by a convolutional neural network.

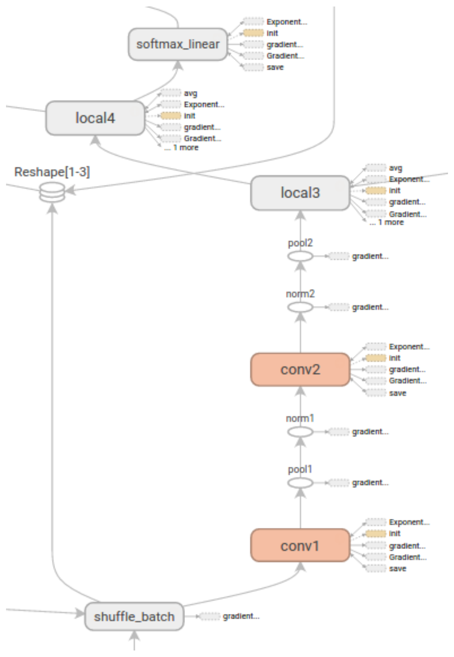


Figure 3.4: Visualizing the operation of a CNN in TensorBoard.

Chapter 4

Deep Reinforcment Learning in TORCS

In this chapter of the thesis, we will describe our approach to apply deep reinforcement learning to TORCS, going into details about how our experiment is set-up.
We will provide the input/output design decisions, explaining how the input of each network is created and what will be controlled through the output. We will present all our design decision concerning the formulation of the MDP, explaining the input/output design decisions, focusing on how the input of each network is created and what will be controlled through the output.
We will then focus on the exploration strategy selection and the reward function used, two other critical points of our design process the evaluation metric and the exploration strategy together with early-stopping criteria and the gradient inverting technique used to bound the networks output.
At the end we will present the three different kinds of neural networks used to perform this experiment, going into detail on the structure of each network.

4.1 Reinforcement Learning problem design

To begin this chapter we will give an outline of the experimental choices we made to fulfill the MDP elements described in Section 2.1.1. At first, we had to choose the input/output model of our network, giving a representation of the environment state (the input) and what the agent will influence with his prediction (the output). We decided to have the agent controlling directly the acceleration and the steering angle of the car, the reason why is explained later in this chapter. The networks are then trained in two different scenarios, the first one having only one track in which the agent races during the learning episode, and the other with five tracks alternating at each episode. Another critical point in the design was the creation of the reward function and the evaluation metric used when testing the agent. For the reward, we decided to favor longitudinal speed as said in a later section, while when testing the agent we also use the total distance raced that gives an easier understanding of the agent performance during the race.

4.1.1 Input/Output

As we said before, the design of the input/output of our experiment was a critical part of our early design stages. We decided to train and test three different types of neural network each of them having a unique input that represents the environment's state. We will now describe in detail how we built the input for each of the three networks.

Numeric input

The first network uses only numeric data as the inputs: these represent telemetry data sent from TORCS at each time-step. We already described the data that TORCS sends to the agent in Table 3.1; it is important to note that not all data coming from the game is used as the state representation: some of them that we thought were useless to the agent (i.e. the opponent sensors, since the agent runs on a track with no opponents) and were so discarded.
These data are then placed in an array and fed to the actor and critic networks. This ended up creating an array of size 29 as the input of our Numeric Network. In Figure 4.1 we provided a graphical representation of the input array, also providing the names of the telemetry data used to build the input.

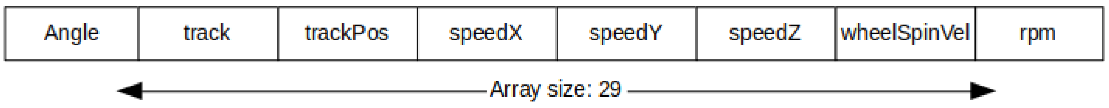


Figure 4.1: Array representing the numeric input for the first network

Image input

For the second network, we design we chose to use an image of the current track state as the input of the network. This image doesn't come directly from TORCS but it is built at each time-step with a method explained later. This has allowed us to run TORCS in text-only mode which significantly speeds up the experiment since it eliminates all the rendering and the visual effects of the game.
We will now explain the process to construct the images described before.

We decided to use a top-down black and white 2D image as the input for the network: we chose this because it is reasonable to think that with this type of input the agent can extrapolate all the crucial information that he needs to drive in the track, with an emphasis on the distinction between the track (in white) and the outside of the track (in black).
One more thing we had to determine was the image size and resolution. We used the same 88x88 image resolution as the author of [1]. This decision was later supported by preliminary experiments where we compared the results obtained with this image size with a network that used a more standard 64x64 image. We noticed that the network using the image suggested by the former gives better results than the latter.
As the final image, we didn't want a classic first-person view of the scene but rather a top-down visualization of the environment, and we will have the agent at the bottom center of the frame facing directly straight.
We start to create our image from a model of the track, that helps us using the Python Pillow library to generate the first top-down image of the entire track. This process is done once at the start of each learning episode. The image is saved and then modified in later steps.
You can see the result of this first step in Figure 4.2.
The final image that will be fed to the neural network will have the agent position to correspond with the bottom center of the image. This gives the agent a good overview of the state of the environment representing what he sees ahead of him in real-time. To do so we will need to extrapolate the direction in which the agent is facing, and then perform a rotation on the previously built image: in this way the entire track is ”facing” the agent direction. This process is done by calculating the relative angle between the agent direction and the direction of the track and then using this angle and a method from the Pillow library to perform a counter-clockwise rotation on the image. The angle of the agent is derived using its speed along the x and y coordinates, while the z direction of the agent is not considered since we are building a 2D image.

$$\theta = \tanh(speed_y/speed_x) \tag{4.1}$$

The result of this rotation can be seen in Figure 4.3.
Once the image is rotated we just need to center it around the agent position. To do so, we consider the agent x and y coordinates coming from the TORCS server and perform on the agent's coordinates the same rotation we had applied on the image in the previous step. This is done by applying the 2D point rotation formula listed below and normalizing w.r.t the width and height of the image. After that, the agent coordinates are translated in pixel coordinates and using a method from the Pillow library we crop a rectangular section from the image having the agent pixel coordinates in the bottom center.

$$x_{rot} = x * \cos\theta - y * \sin\theta \tag{4.2}$$

$$y_{rot} = y * \cos\theta + x * \sin\theta \tag{4.3}$$

After these steps we have the final image representing the state of the environment that will be used as the input for some of our network. In Figure 4.4 you can see two possible final images: the first one taken when the agent was in a straight, while in the second one it is approaching a curve.

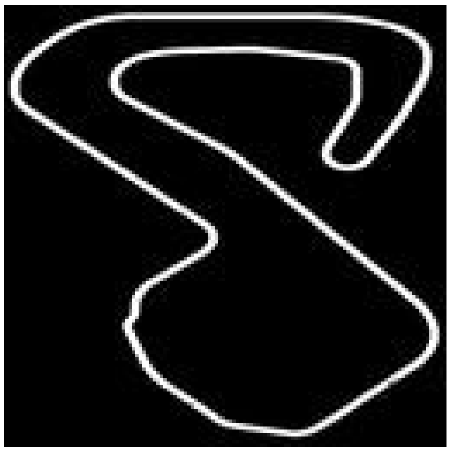


Figure 4.2: The first 3000x3000 image build containing the overview of the Brondehach track.

Mixed input

For the third network, we decided to use a mixture of the two previous inputs to create a network that uses both an image, as explained in Section 4.1.1 , and some telemetry data as the input. Differently from the telemetry used in 4.1.1, in this case, we didn't include the track range sensors and the track position, because the agent should be able to infer those values from the image we also provide as input to the network. This resulted in having the same 88x88 2D black and white image and a 9 element numeric array as the two inputs of the network.
The telemetry data we used in this case is provided in Figure 4.5

As far as the output is concerned, we chose to let the agent act on a low level, directly influencing the actuators of the car. The agent controls both the acceleration and the steering angle of the vehicle, and the output is then sent to the game at each time-step using the SnakeOil client described in Section 3.1. This decision was made because it allows the agent to drive around the track without the need of external tools and so we can study the performance of the agent in a scenario where it isn't influenced by other factors when choosing an action.

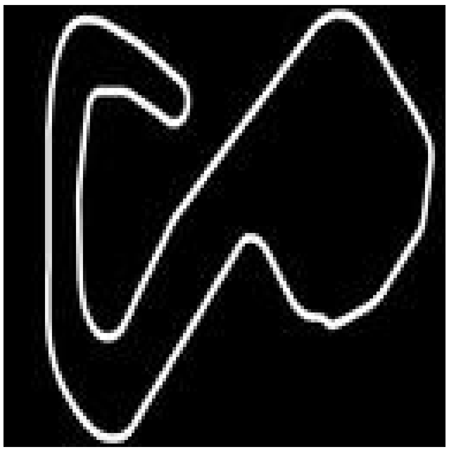


Figure 4.3: The overview of the track after applying the rotation according to the agent direction.

(a) Final image of a straight.

(b) Final image of a curve.

Figure 4.4: Final images obtained when the agent is on a straight or when the agent is approaching a curve

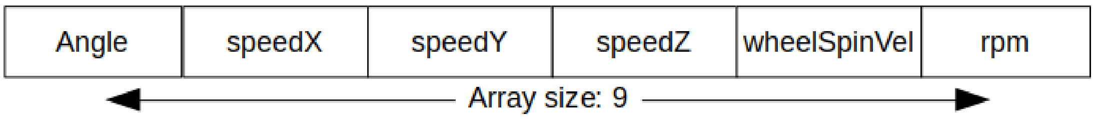


Figure 4.5: Array representing the numeric input for the third network

This, of course, is not the only possible solution, in fact, we explored the possibility to have the agent's control on a higher level of abstraction. In this case, the agent would output a single value that represents the desired track position at the next time-step, and then uses a "follower bot" to determine the right acceleration and steering angle to apply. This situation, however, is not explored in this thesis and can be used as a starting point for future work on the subject.

Once the experiment is started the agent will at first receive the raw telemetry data from the game; this will be either passed directly to the network or used to build the image based on which neural network the agent is using, or a mixture of both the previous. When the state of the environment is built and ready to be used, it is passed to the actor network that will predict the first action.

4.1.2 Exploration Strategy

After the actor has chosen the action that the agent will perform, we need to choose an exploration strategy to apply to said action, as explained in Section 2.1.9.
For example, if we used a ϵ-greedy strategy discussed in Section 2.1.10 to try some randomly

chosen action from a uniform random distribution, we could choose actions that significantly slow down the experiment: for example we can get stuck in a local minima where the car immediately brakes and doesn't move on the track.
To avoid this situation an Ornstien-Uhlenbeck noise has been chosen. This process, better discussed in [36], works with the following formula:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \tag{4.4}$$

where θ is a factor that controls how fast the process is reversed towards its mean μ, while σ is the variance of the process. This exploration noise is applied for the first 100000 time-step and this allows the agent to explore a lot of meaningful action possibilities. In Table 4.1 with the values of these parameters used in the experiment.

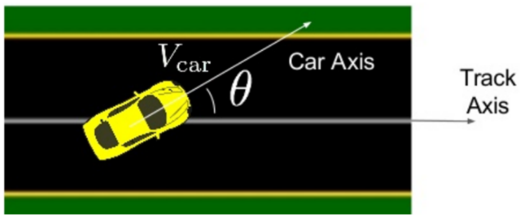
x	μ	θ	σ
Steering	0.0	0.6	0.3
Acceleration	0.2	1.0	0.1

Table 4.1: Parameters chosen according to [2] for the Ornstien-Uhlenbeck noise

After this step, the modified action is sent to Torcs using the standard OpenAI gym method described in 3.3, and a new observation of the environment is sent to the agent as well as the reward obtained from the selected action.

4.1.3 Reward Function

The design of the reward function is perhaps another critical step in designing our experiment. This can be considered the most important choice to make as it will heavily influence the agent behavior during the learning process.
Since our objective is that the agent should be able to learn to drive around the track as fast as possible, we decided to use the longitudinal velocity penalized by the transverse velocity.
In Figure 4.6 the basic idea behind the reward function, with the formula used to calculate its value, is pictured. Note that v_x and θ represent respectively the car speed along the x axis and the angle between the car and the track orientation at time t.



Reward at time step t:
$R_t = v_x \cos\theta - v_x \sin\theta$

Figure 4.6: Reward Function

After calculating the reward and filling the replay buffer as described in Section 2.2.2, we extract a batch from it and use it to calculate the target q-values. The q-values together with the reward are then used to train the critic network and to calculate the gradients of the actions that are needed to train the actor.

4.1.4 Gradient Inverting

After the critic calculates the gradients, we perform what is called *gradient inverting* as proposed in [37]. This is done to bind the action values between the range we need in our experiment.
In this work, the authors explain the method to bound an action between certain values without using specific activation functions that can cause the output to saturate at the limits of the activation. In our case using a tanh-shaped activation function seemed the most reasonable choice (knowing that it has limits at -1 and 1 exactly like our actions, as explained in a later section of this chapter); but during our experiment we noticed that using a tanh-shaped activation caused the network to saturate during the first few episodes of the experiment, whilst using gradient inverting prevented the saturation from happening drastically improving the results of our experiments.
Following [37] we built a custom function that takes as inputs the gradient, its relative action and the maximum and minimum bounds of the action, and returns the inverted gradient.

4.1.5 Episode stopping criteria

After both the target and the normal network are trained and before starting the next step we perform an early evaluation of the training to see whether is worth to continue with this episode. This is done to speed up the learning process and to penalize the agent for performing wrong behaviors. An episode is stopped early in three different cases:

- If the agent is driving backward (with an angle less then zero) on the track. This is quite obviously wrong and if the car is found in this situation the episode is terminated and a negative reward is assigned to the agent;
- If the car is driving for more than three seconds with half is width outside the track bounds. We chose to stop early the learning episode and penalizing the agent with a negative reward in this scenario because we found out that in certain scenarios the agent was driving around the track following the walls outside the road. Baring from this situation it is also obvious that we want our agent to drive inside the track bounds without exceeding them. We also, however, decided not to immediately stop the episode when the car was found out of track because in certain situation it is feasible to take a turn going slightly outside the track bounds especially when considering taking a corner at very high speeds;
- If the car is stuck getting a low reward for several steps. This situation happens rarely but can be avoided by stopping the episode early; we found out that sometimes the car was stuck against an outside barrier getting close to zero reward at each time step and unable to reverse and get back on the track, so rather than wasting a lot of steps waiting for the episode to automatically terminate we decided to detect when the agent is stuck getting close to zero reward and to stop the iteration early and proceed to the next one.

After doing all the steps previously described, the process is repeated until a stopping condition is met or until the max number of steps for an episode is reached.
In Figure 4.7 we provided a simplified scheme of the two different experiments we made highlighting all the steps described above while in Table 4.2 we provide all the hyperparameters commonly used during our experiments by the three networks.

In the next section, we will go into more detail about the core and the structure of the three types of networks we used to perform the learning process of our experiment.

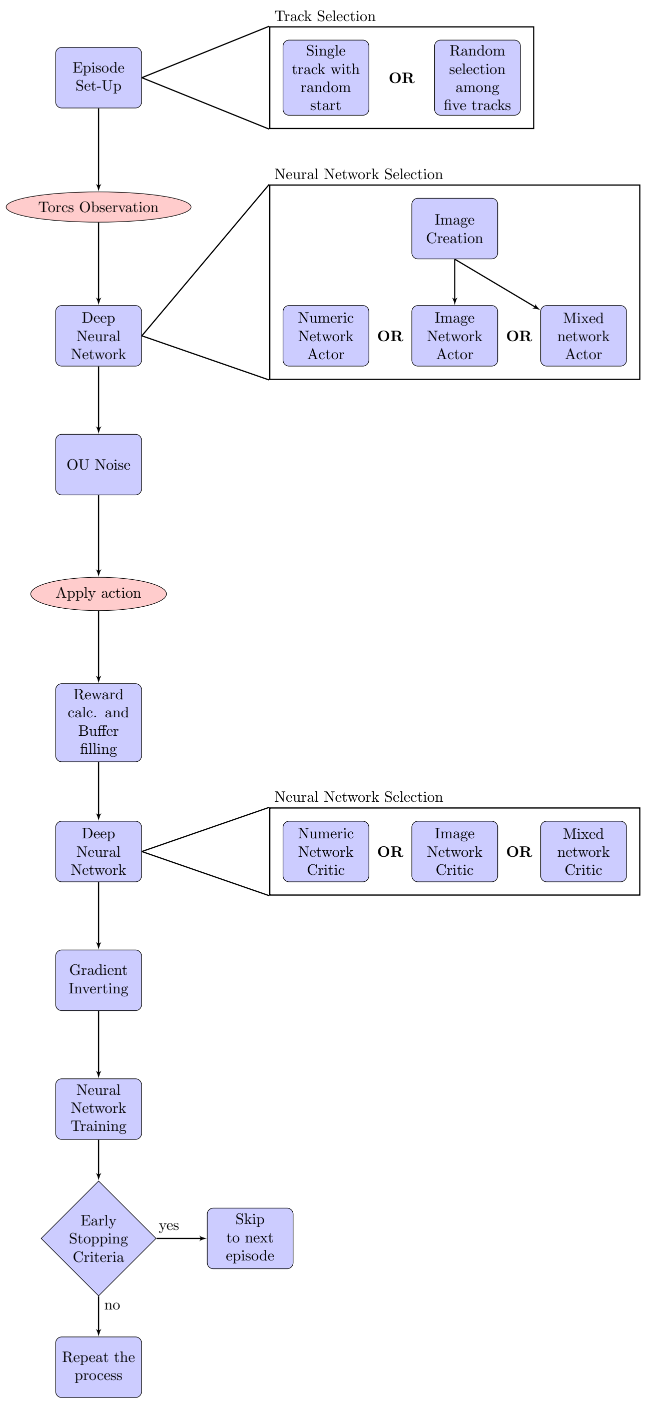


Figure 4.7: The experiment design scheme with the major operation performed at each step of each episode. Ellipsis are used when there is a direct communication with the TORCS server.

Name	**Value**	**Description**
buffer size	100000	Size of the experience replay buffer
batch size	32	Size of the batch randomly extracted from the buffer
gamma	0.99	Discount factor described in Section 2.1.3
rho	0.001	Hyperparameter used for target network weights updated as explained in Section 2.2.3
lra	0.0001	Actor learning rate
lrc	0.001	Critic learning rate
explore	100000	Number of steps dedicated to the exploration
max episode	2000	Maximum number of episodes
max step	10000	Maximum number of steps per episode

Table 4.2: Hyperparameters used by the three networks

4.2 Neural Networks Structure

In this section, we will present in detail the structure of the networks that we will use to perform our Deep Reinforcement Learning experiment. The algorithm used by these networks is the standard DDPG with actor-critic method, experience replay and target networks described in Section 2.2.3.
It is important to note that for this work we will use three different types of networks that are described in the following sections.

- The first network (Numeric Network) uses only numeric data as the representation of the state;
- The second network (Image Network) uses an image representing the position of the agent on the track, extracted using the methods described in Section 4.1.1;
- The third network (Mixed Network) uses both the image and some numeric values as the representation of the environment state.

We will now go into more details about the structure of these three networks.

4.2.1 Numeric network

The first network is a fully-connected network that will only use numeric inputs; we already described the structure of the input in Section 4.1.1 and will now focus on the structure of the network itself.
We will begin describing the network by analyzing the actor and the critic.
The actor network is composed of two hidden layers with 300 and 600 hidden units respectively, both with a rectified linear unit (ReLU) activation function.
The output of the actor consists of 2 continuous actions, both with 1 unit and linear activation function that represent the Steering and the Acceleration of the car. As seen in Table 3.2, these actions cannot have any type of value, but they need to be bounded. Both the outputs will be bounded between the value -1 and 1 where for the steering output -1 represent a max right turn and 1 represent a max left turn, while for the acceleration output -1 means full brake and 1 full throttle.
To perform this bounding action we will use gradient inverting as explained in Section 4.1.4. This method will be in common to all the experiments performed and the three networks used, so we will describe it here once for all.

On the other hand, the critic network takes both the actions provided by the actor and the environment state as inputs. It is composed of one hidden layer with ReLU activation function and 300 hidden units, a second layer with linear activation and 600 hidden units and then a third hidden layer with 600 hidden units and ReLU activation. According to [38] the actions were not included until the second hidden layer, where they are merged with the input layer. The output is a single unit linear activation layer that represents the q-value. The model is then optimized using the Adam Optimizer.
In Figure 4.8 we provide a schematic structure of the Numeric Network for both the actor and the critic.

4.2.2 Image Network

The second type of network is built on the already created numeric network to use an image as the input of the network as explained in Section 4.1.1.
Once the image is created, it is then converted to an array and fed to the network as the input. Both the actor and the critic, in this case, need a method to process images, so on top of the fully-connected layers used in the numeric example some convolutional layers are added, because convolutional layers are well suited to process images as explained in Section 2.2.1. Following

the same approach of Google Deep Mind in [1], where the authors were able to develop agents able to play various Atari games from image input, three convolutional layers were added to the network. The first convolution consists of 32 8x8 filters with 4x4 stride and ReLU activation, the second convolution consists of 64 4x4 filters with 2x2 stride and ReLU activation, while the last convolution consists of 64 3x3 filters with 1 stride. These layers have the task of processing the image before passing to the same feed-forward network described in Section 4.2.1.
In Figure 4.9 we provide a schematic structure of the Image Network.

4.2.3 Mixed network

For the last of the three networks created in this thesis, we have one that uses both images and some of the telemetry data used by the Numeric Network input as explained in Section 4.1.1. The network is based on the two previously described having three initial convolutional layers to process the image, but then, before feeding the result to the feed-forward network, the results of the convolution are added to the numerical data. To do so a new layer is introduced for the actor network, while for the critic network we use the already existing adding layer where the actions are introduced into the network, to also add-in the numerical input.
In Figure 4.10 we provide a schematic structure of the Mixed Network.

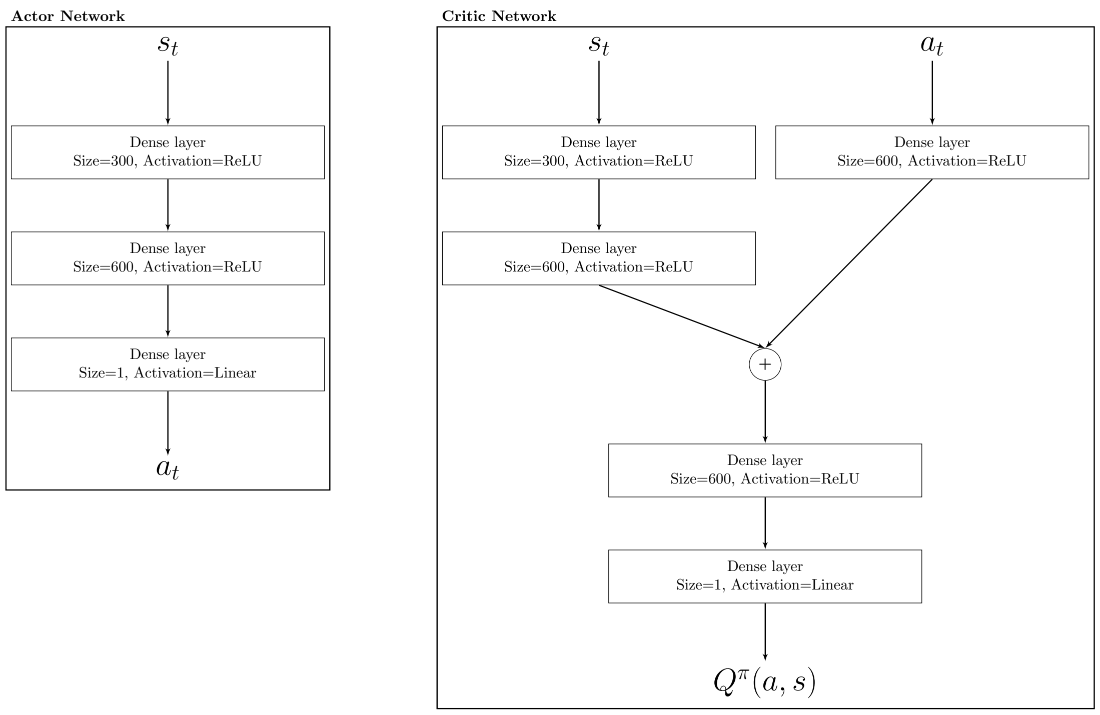

Actor Network
s_t
Dense layer
Size=300, Activation=ReLU
Dense layer
Size=600, Activation=ReLU
Dense layer
Size=1, Activation=Linear
a_t
Critic Network
s_t
a_t
Dense layer
Size=300, Activation=ReLU
Dense layer
Size=600, Activation=ReLU
Dense layer
Size=600, Activation=ReLU
+
Dense layer
Size=600, Activation=ReLU
Dense layer
Size=1, Activation=Linear
$Q^{\pi}(a, s)$

Figure 4.8: The Actor and Critic network scheme of the Numeric Network

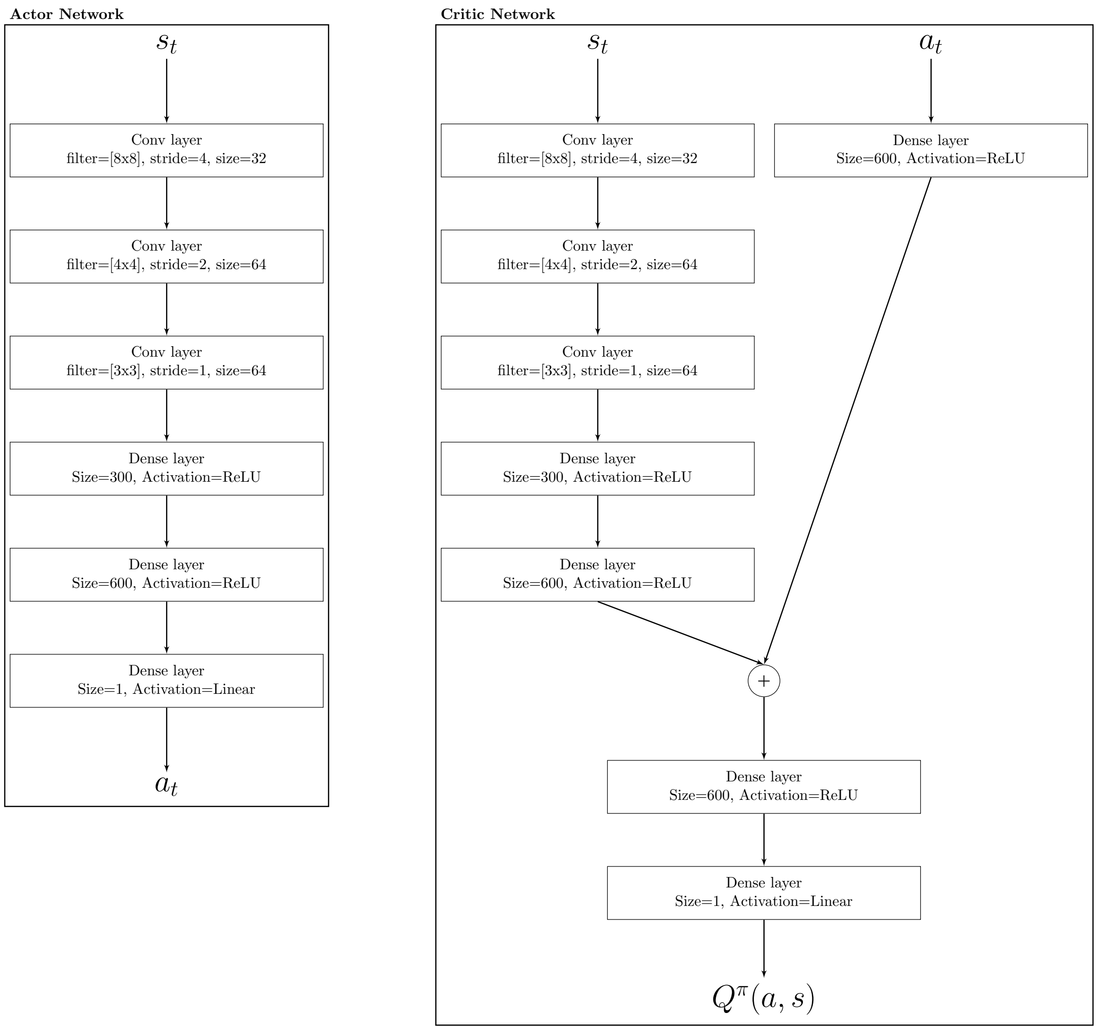

Actor Network
s_t
Conv layer filter=[8x8], stride=4, size=32
Conv layer filter=[4x4], stride=2, size=64
Conv layer filter=[3x3], stride=1, size=64
Dense layer Size=300, Activation=ReLU
Dense layer Size=600, Activation=ReLU
Dense layer Size=1, Activation=Linear
a_t
Critic Network
s_t
a_t
Conv layer filter=[8x8], stride=4, size=32
Dense layer Size=600, Activation=ReLU
Conv layer filter=[4x4], stride=2, size=64
Conv layer filter=[3x3], stride=1, size=64
Dense layer Size=300, Activation=ReLU
Dense layer Size=600, Activation=ReLU
+
Dense layer Size=600, Activation=ReLU
Dense layer Size=1, Activation=Linear
$Q^\pi(a, s)$

Figure 4.9: The Actor and Critic network scheme of the Image Network

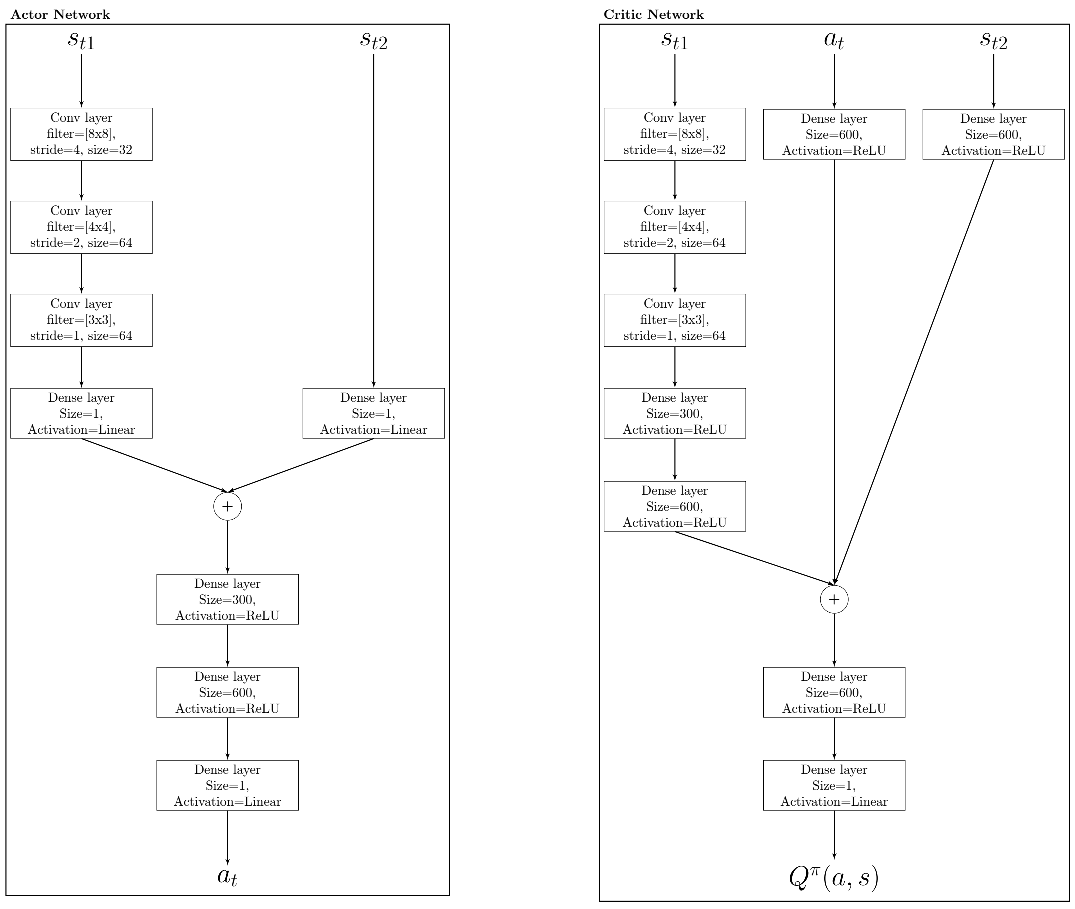

Actor Network
s_{t1}
s_{t2}
Conv layer filter=[8x8], stride=4, size=32
Conv layer filter=[4x4], stride=2, size=64
Conv layer filter=[3x3], stride=1, size=64
Dense layer Size=1, Activation=Linear
Dense layer Size=1, Activation=Linear
+
Dense layer Size=300, Activation=ReLU
Dense layer Size=600, Activation=ReLU
Dense layer Size=1, Activation=Linear
a_t
Critic Network
s_{t1}
a_t
s_{t2}
Conv layer filter=[8x8], stride=4, size=32
Dense layer Size=600, Activation=ReLU
Dense layer Size=600, Activation=ReLU
Conv layer filter=[4x4], stride=2, size=64
Conv layer filter=[3x3], stride=1, size=64
Dense layer Size=300, Activation=ReLU
Dense layer Size=600, Activation=ReLU
+
Dense layer Size=600, Activation=ReLU
Dense layer Size=1, Activation=Linear
$Q^{\pi}(a, s)$

Figure 4.10: The Actor and Critic network scheme of the Mixed Network

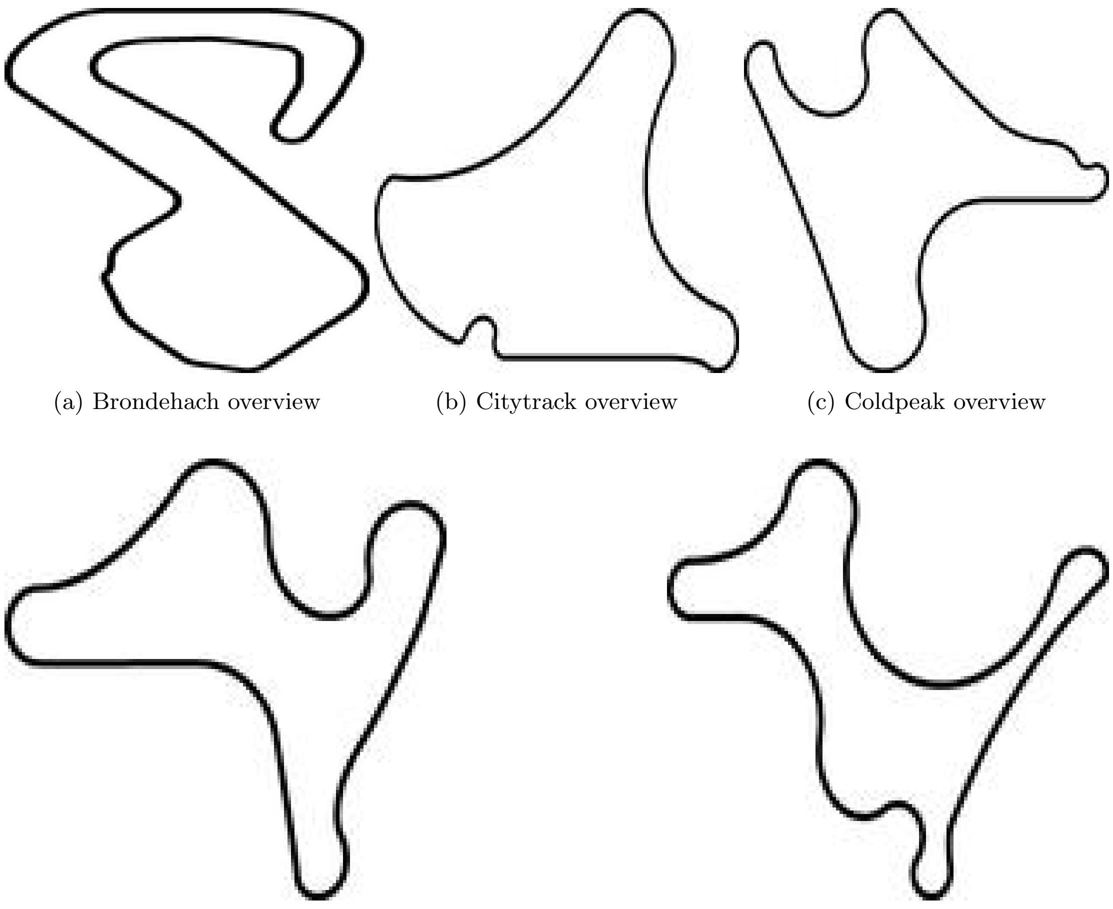
Chapter 5

Experiment Results

In this chapter, we present the results of both the training and testing of our experiment. At first, we focus on the training results dividing them for each of the three networks we used as described in Section 4.2 and also for the two different training scenarios we will describe later. For the training we analyze the graphs that show the progression of the reward function and the distance raced; this allows us to evaluate the performance of the networks during all the training processes. We also use graphs taken at specific episodes that we think have significant importance to show the evolution of the behavior of the agent. These graphs include the steering, acceleration and the track position of the car.
For the testing part, we provide the cumulative reward gained and the distance raced on each of the testing track that will be described later, while also tracing the image of the track and the trajectory the car kept during the testing lap.

5.1 Training results

In this section, we will focus on the results that our three networks obtained during the training process in two different environmental setups.
First Scenario: the track is randomly selected every episode from a selection of five different tracks that we chose a priori. We selected these five tracks because they form a good mixture of easier and harder tracks, with some of them having long straights in which the agent can reach higher speeds and some having sequences of turns so that the agent needs to learn to control its speed to stay on the track.
Second Scenario: the track on which the agent performs the training is fixed for the entirety of the learning experiment, while the agent is forced to start the episode from a random point in the track that changes at every new episode; this decision is made to avoid a possible scenario where an agent that always starts in the straight at the beginning of the track can possibly overfit and have a worse performance than expected. The track we chose is called brondehach and the reader can see an overview in Figure 5.1a. We chose this track because it is the most difficult out of the five selected, having lots of narrow turns followed by long straights, and also because it is the only track that has been created by a human designer, while the other four are procedurally generated by an algorithm.
In Figure 5.1 we present the overview of the five tracks we used for our experiment.



(a) Brondehach overview (b) Citytrack overview (c) Coldpeak overview

(d) Emero-city overview (e) Alsoujlak-hill overview

Figure 5.1: Overview of the five tracks used during the training process

We will discuss the results focusing on each of the networks separately.

5.1.1 Numeric Network

At first, we will focus on the Numeric Network performance during training (see Section 4.2.1 for the structure of the network). In Figure 5.2 we show the reward and the distance raced by the agent during the training process. The reward is shown at each time-step of the experiment while the distance is shown per episode of training, since the distance that the agent travels in one frame is very small and not significant enough to be shown. As can be seen in both Figure 5.2a and 5.2c the reward the agent gets is similar whether it raced on one or five tracks, getting a high reward in the early stages of the training and maintaining a good performance for the entirety of the training process. Since the reward is directly correlated with longitudinal speed (see Section 4.1.3) we can deduce that the agent kept a high speed around 100km/h. As far as the distance is concerned we can see from Figure 5.2 that the agent after about 250 episodes started slowly increasing the distance performed getting to peaks of 7000 meters; even though this result was obtained when the track selected was the one called citytrack (see Figure 5.1b),which is one of the easier track of our selection it is still a good result considering that all the five tracks are about 3000 meters long, this means that the agent successfully performed more than one lap. The distance raced is not that high in the scenario where the agent is trained only on the brondehach track (see Figure 5.1a), this however was expected since the agent was forced to start in a different position on the track which we consider to be the most difficult of the selection. Still, the agent manages to get good results after around the 250 episode mark peaking with the completion of one lap.

We will compare the behavior of the agents by analyzing two sets of graphs.
We record the agent output (both the acceleration and the steering angle) together with the track position in two different moments of the training the process. The first record was taken around the 50000 steps of the training; as explained in Section 4.1.2, this represents the middle of the exploration procedure using the Ornstien-Uhelback noise. The second record was taken when the agent performed its best episode in terms of distance raced and reward gained.
In Figure 5.3 we will show this comparison where side by side graphs represent the same output in the two different scenarios. The first thing to notice is that when the agent was still exploring the episode lasts very few steps and is then terminated because the agent is out of the track bounds as the value of the track position is greater than one; this is however expected since the agent is still exploring various situation and is not always choosing the better policy. It is notable however that even though it is the early stages of learning the agent is already preferring higher values for the acceleration output while the steering angle is not stable and causes the agent to exceed the track bounds.
It is immediately clear that, as expected, the episode lasts for far more step when the agent achieved its best performance. Moreover, in the scenario with five tracks, following the track position in Figure 5.3f we can see that the car never went out of the track so we can infer that the episode was terminated because the maximum number of steps was reached during the episode, while, when training on the more difficult track only, the agent at one point went off track and the episode was ended. Also in both scenarios, the agent kept the accelerator always at the maximum value (Figure 5.3b and 5.4b), meaning that it managed to follow the track without using the brake. This could be a problem because we expected the agent to learn to brake when taking difficult turns and this might become a problem on tracks with many difficult turns. Another unexpected result was the steering angle evolution. We expected that the agent would learn to keep the steering wheel straight (with a value of zero) during most of the track and modify the value when approaching a turn. Instead as seen in Figure 5.3d and 5.4d the agent stays on track but it always tends to alternated form value close to 1 and value close to -1, leading the car to proceed in a strange pattern along the tracks.

5.1.2 Image Network

In this section we will analyze the training results obtained by the Image Network (Section 4.2.2 for the network structure). We will follow the same approach used for the Numeric Network, firstly analyzing the reward and the distance raced in both training scenarios and then focusing on the evolution of the outputs of the network. The first graphs immediately show that the network performance is worse than the one previously analyzed. The reward functions of both the experiments reach values not comparable to the one obtained by the Numeric Network. Even the maximum distance raced barely reaches the 1000 meters, showing that the network struggled to keep the car on track.

When analyzing the acceleration and the steering angle graphs we can see when looking at Figure 5.7a and Figure 5.6a that even using this network the agent quickly learns to keep the car on full throttle to maximize the reward function; when looking at the episode where the agent performed the best in terms of distance raced we can see that in both cases (Figure 5.7b and 5.6b) the acceleration is kept at maximum as observed in the Numeric Network. When observing the steering angle both during exploration and at peak performance we can see that it has a less sporadic spread, keeping it closer to zero than in the previous case, but this made the agent perform worse than the previous network.
To explain this situation we hypothesize that the agent that uses images as the state representation could not properly extrapolate the current speed at which he was going. This is a fair assumption because when looking at an example of a state image as in Figure 4.4 it is difficult to understand at which speed the car is moving at that precise moment. This could lead the agent to take a turn at a speed and angle that are not possible causing it to go out of the track bounds.

To test this possibility we performed another episode using a modified Image Network.

5.1.3 Image Network steer only

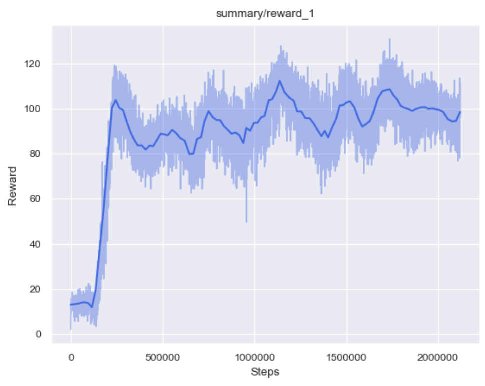
To explore whether our hypothesis was correct we set-up another experiment. We used the previously tested Image Network but since we thought that the network couldn't extrapolate the speed of the car from the image, we modified the output of the network having the agent only controlling the steering angle. The acceleration was then delegated to a modified version of the SnakeOil client describe in Section 3.1. The client aims to reach a fixed speed of 50km/h that lets the agent to adjust in time to any turn on the tracks. We only tested this situation on the brondheach track.
Looking at Figure 5.8 we can see that the reward function is more consistent and the car reaches a higher distance raced than the Image Network controlling the speed when racing only on the brondehach track.
Also by looking at the steering angle at peak performance in Figure 5.9, we can see that the distribution is similar to the one of the Numeric Network where the steer rapidly jumps between high and low values. This, even though it seems to confirm our hypothesis, is not enough to determine it, because there could have been other factors (i.e. the fact that the network needed to control only one actuator) that helped the modified Image Network achieved better performance.
To even further test this hypothesis we will now show the result obtained by the Mixed Network that uses both images and telemetry data to represent the environment state.

5.1.4 Mixed Network

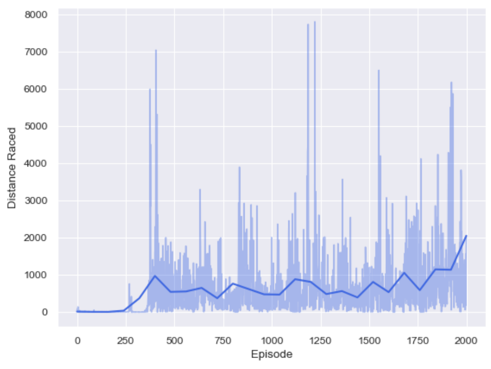
In this section, we will analyze the results obtained during training by the Mixed Network (see Section 4.2.3 for the network structure). This Network should in part resolve the problem encountered by the Image Network since the value of the speed is directly used to create the state representation together with the images.
As we can see in Figure 5.10 the reward function in both scenarios is higher than the one reached by the Image Network peaking around 120. The function is similar to the one obtained by the Numeric Network but this network seems slower to obtain these results. This is maybe because the network has more layers to perform image convolution and to process the telemetry data. The network performs well in regards to the distance raced, reaching a peak of 3000 meters when racing on five tracks. This, as explained before, means that the agent manages to keep the car on track for a full lap.

By analyzing the performance of the network in the early stages of exploration we can also see that the network is slower in learning. As we can see in Figure 5.11a and 5.12a the network tends to lower the acceleration when not necessary if compared to the relative track position that seems to indicate that the car was on a straight line (this is even more evident when we think that in the scenario with five tracks the agent always starts on the beginning straight).
When analyzing the evolution of the steering instead we can see from both the performance during the exploration process and the one when the agent reached the maximum distance that it is similar to the one of the Numeric Network with a high variance between high and low value; we can then assume that the car will have a sporadic movement while keeping on the track.

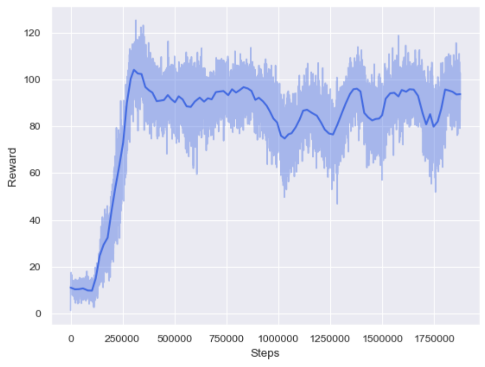
In this section, we analyzed the performance that our three networks reached during the training process. In the next section, the model of the agents will be tested by evaluating them on three unseen tracks that we selected.



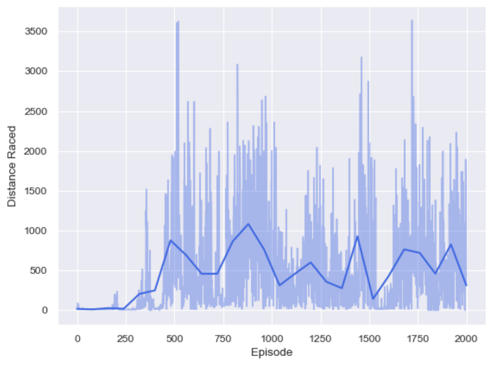
(a) Reward collected at each step



(b) Distance raced at each episode

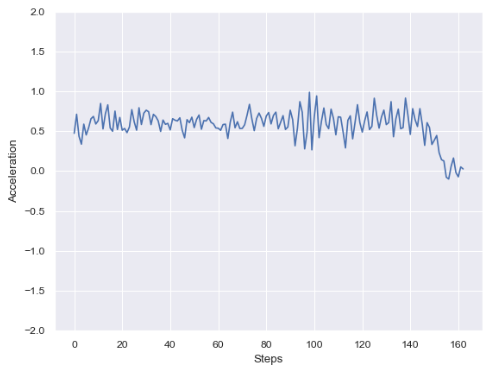


(c) Reward collected at each step

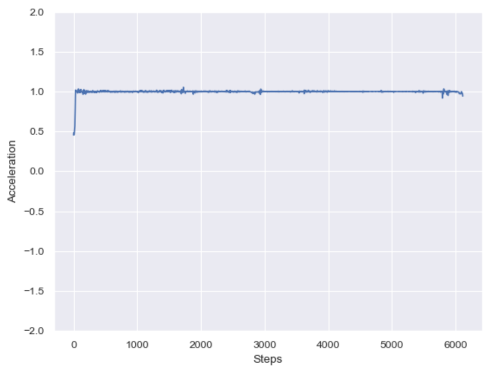


(d) Distance raced at each episode

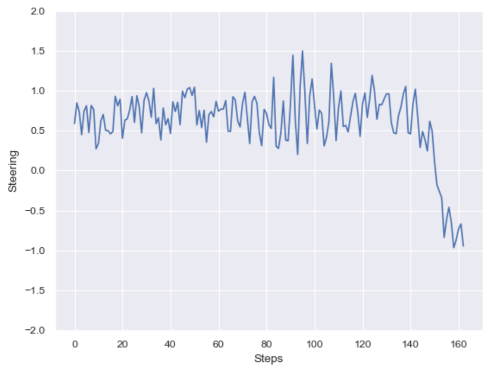
Figure 5.2: Reward and distance obtained by the Numeric Network; above when racing on five tracks, below when racing on a single track with random start



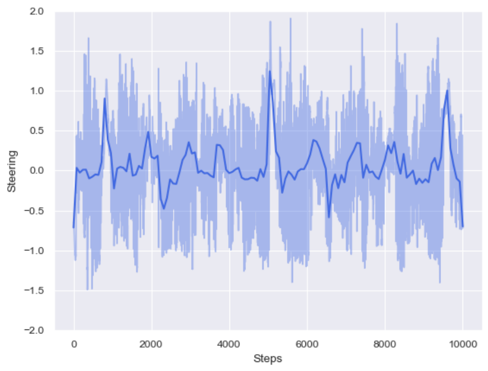
(a) Acceleration mid exploration



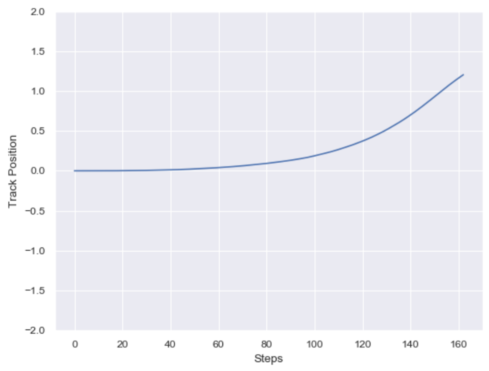
(b) Acceleration best episode



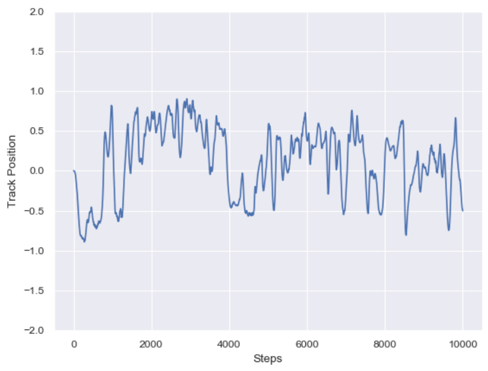
(c) Steering mid exploration



(d) Steering best episode

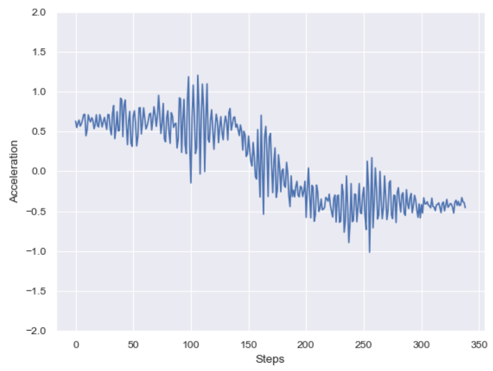


(e) Track position mid exploration

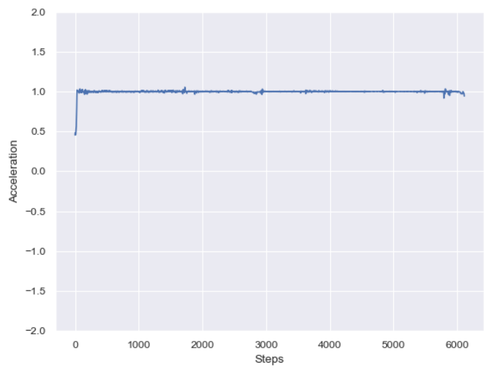


(f) Track position best episode

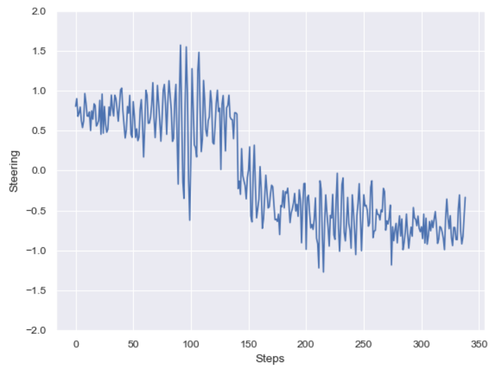
Figure 5.3: Set of graphs representing the acceleration, steering angle and track position recorded by the Numeric Network when racing on five tracks



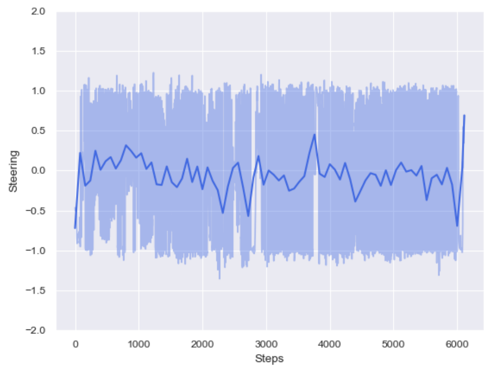
(a) Acceleration mid exploration



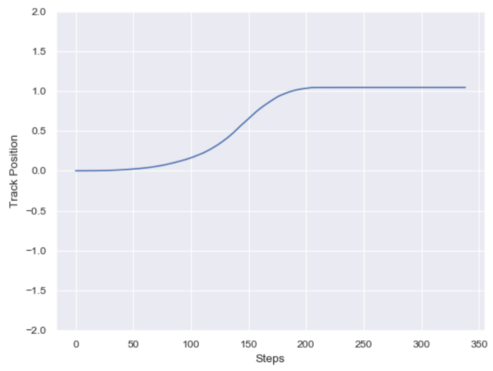
(b) Acceleration best episode



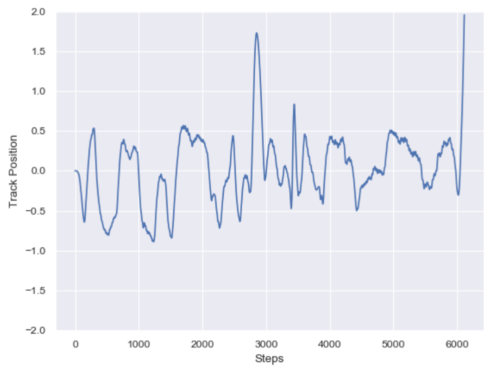
(c) Steering mid exploration



(d) Steering best episode

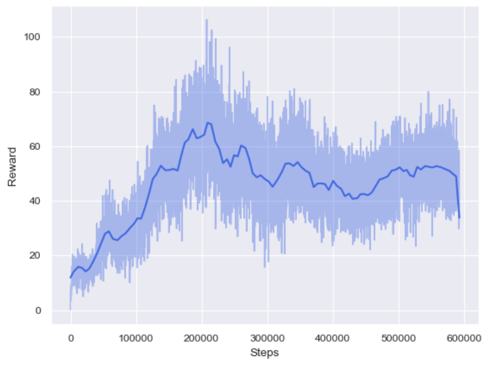


(e) Track position mid exploration

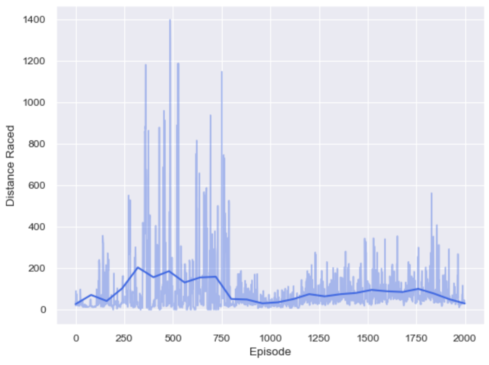


(f) Track position best episode

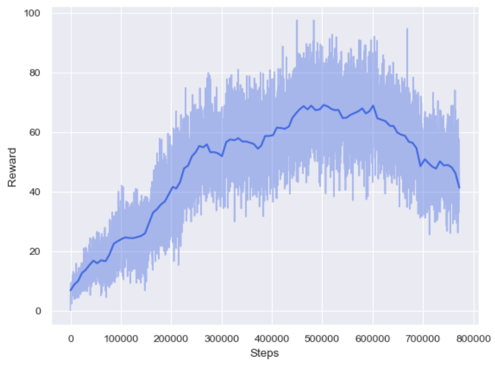
Figure 5.4: Set of graphs representing the acceleration, steering angle and track position recorded by the Numeric Network when racing on one track



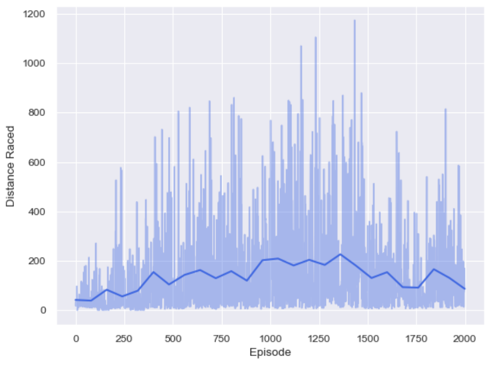
(a) Reward collected at each step



(b) Distance raced at each episode

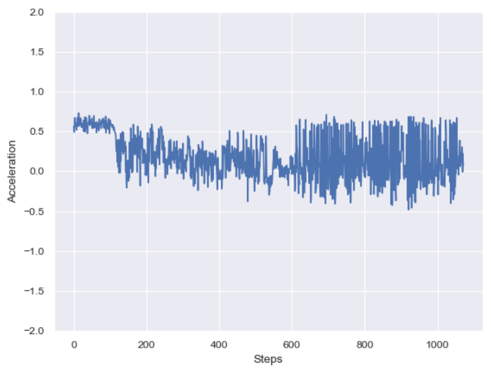


(c) Reward collected at each step

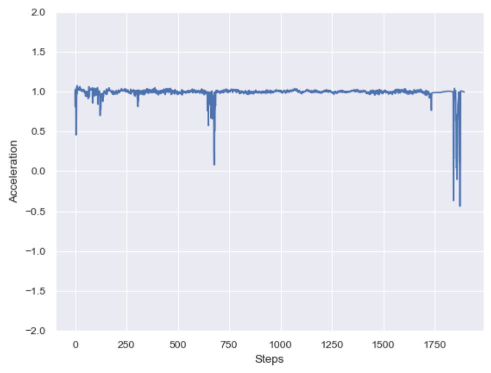


(d) Distance raced at each episode

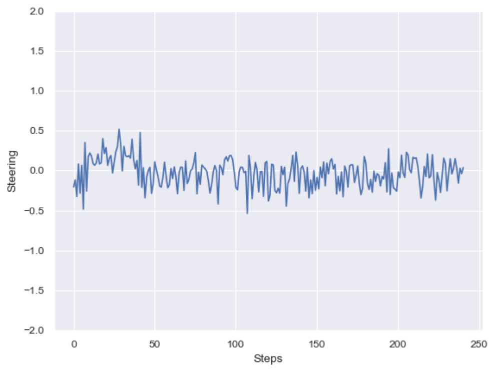
Figure 5.5: Reward and distance obtained by the Image Network; above when racing on five tracks, below when racing on a single track with random start



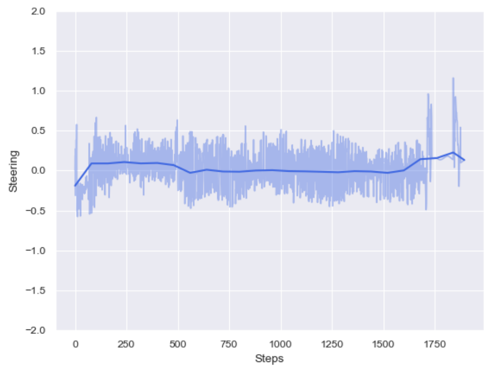
(a) Acceleration mid exploration



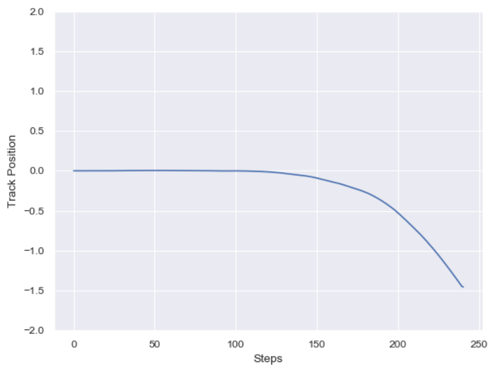
(b) Acceleration best episode



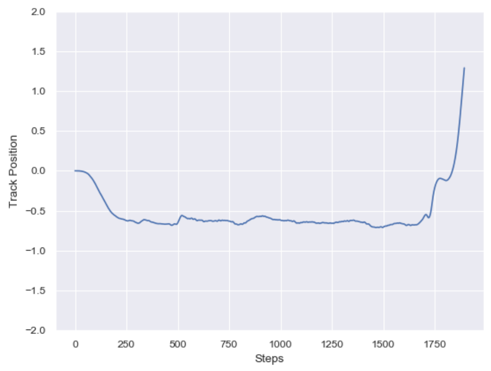
(c) Steering mid exploration



(d) Steering best episode

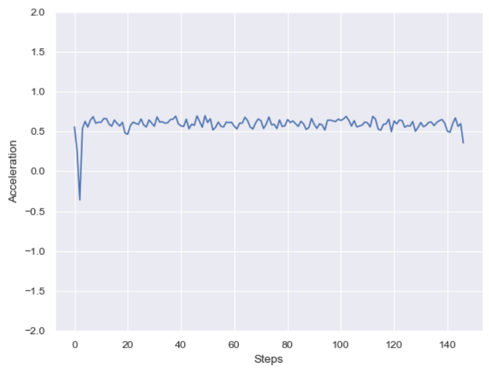


(e) Track position mid exploration

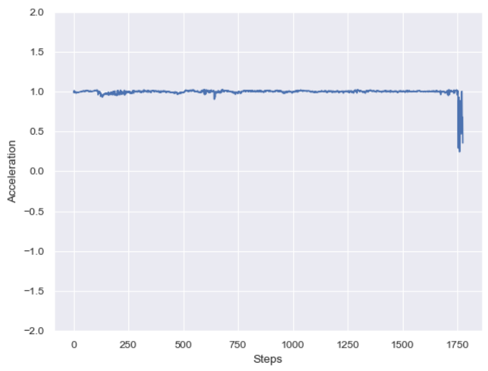


(f) Track position best episode

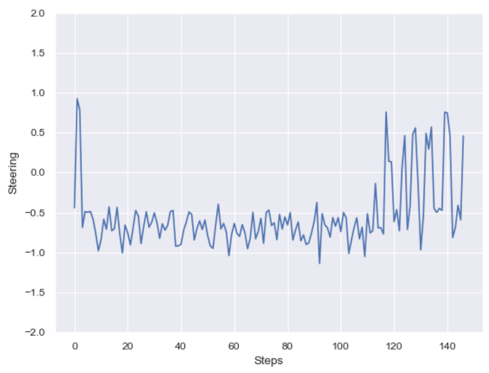
Figure 5.6: Set of graphs representing the acceleration, steering angle and track position recorded by the Image Network when racing on five tracks.



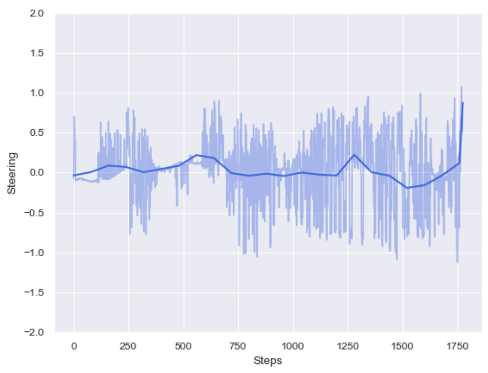
(a) Acceleration mid exploration



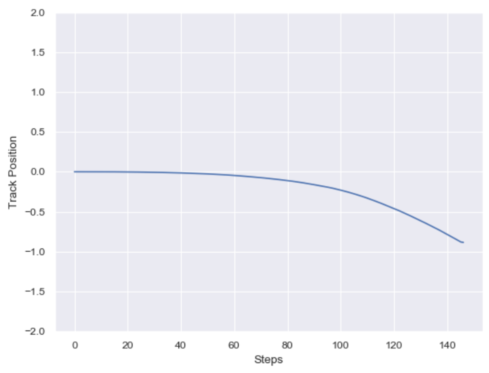
(b) Acceleration best episode



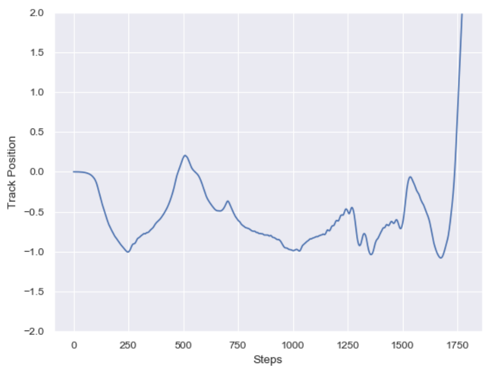
(c) Steering mid exploration



(d) Steering best episode

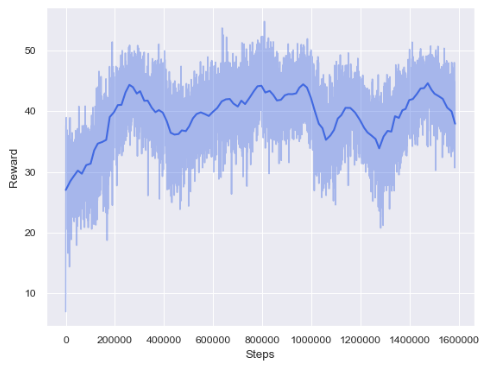


(e) Track position mid exploration

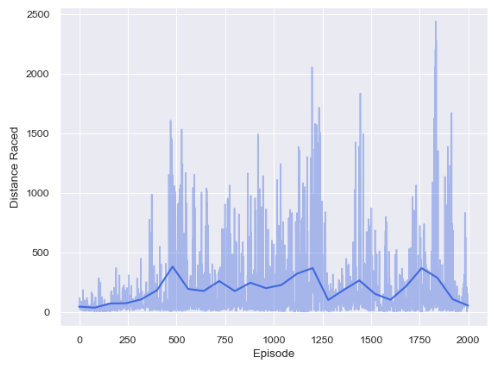


(f) Track position best episode

Figure 5.7: Set of graphs representing the acceleration, steering angle and track position recorded by the Image Network when racing on one track.

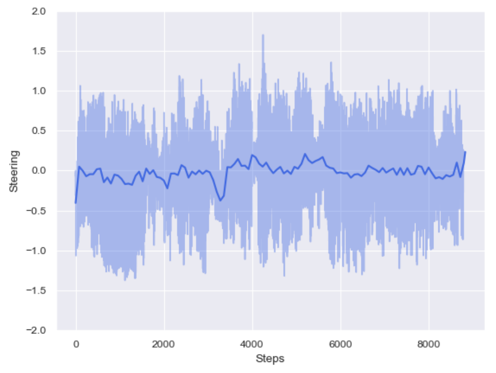


(a) Reward collected at each step

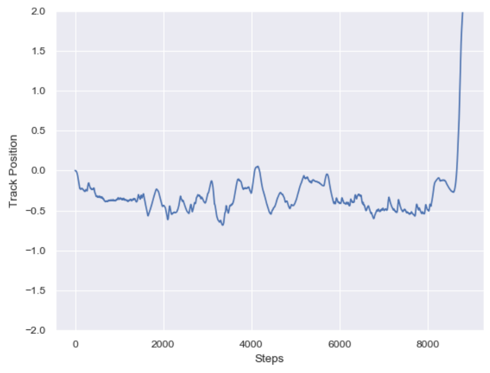


(b) Distance raced at each episode

Figure 5.8: Reward and distance raced obtained by the Image Network controlling only the steering angle when racing on one track

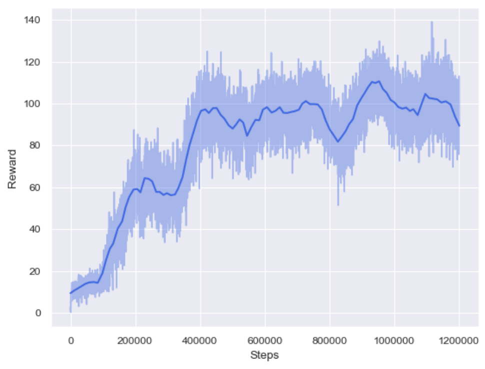


(a) Steering best episode

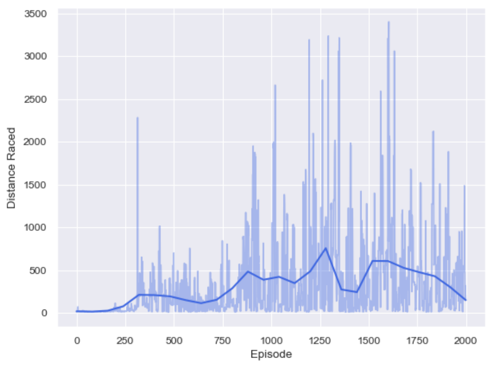


(b) Track position best episode

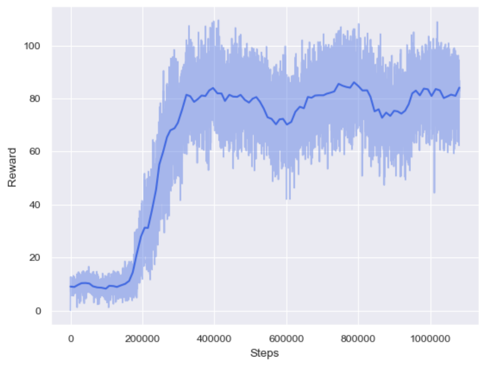
Figure 5.9: Steering and track position of the Image Network controlling only the steering angle when racing on one track



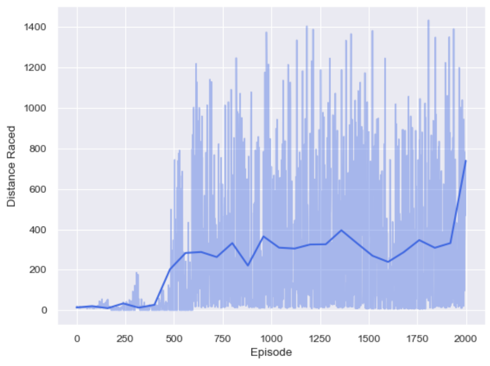
(a) Reward collected at each step



(b) Distance raced at each episode

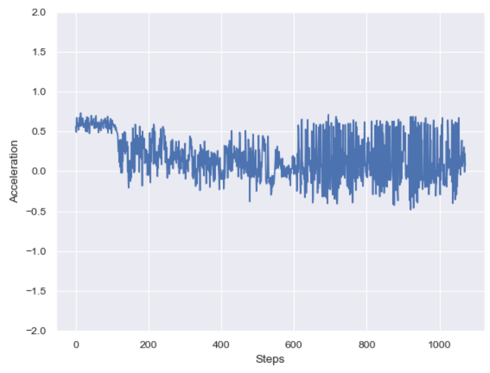


(c) Reward collected at each step

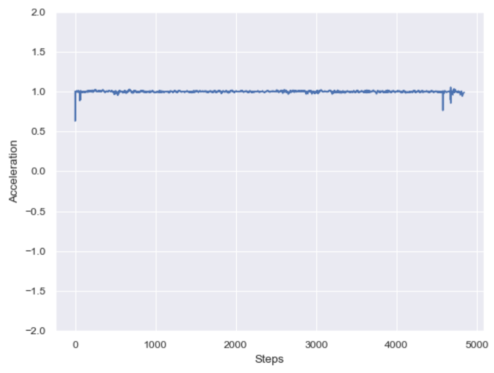


(d) Distance raced at each episode

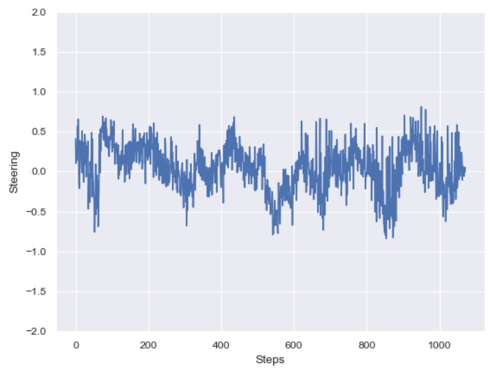
Figure 5.10: Reward and distance obtained by the Mixed Network; above when racing on five tracks, below when racing on a single track with random start



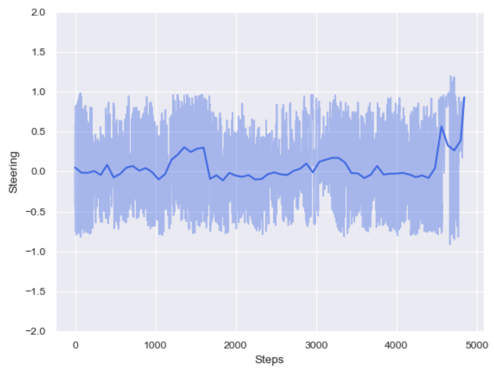
(a) Acceleration mid exploration



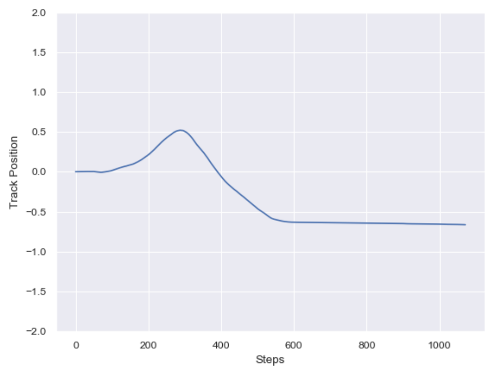
(b) Acceleration best episode



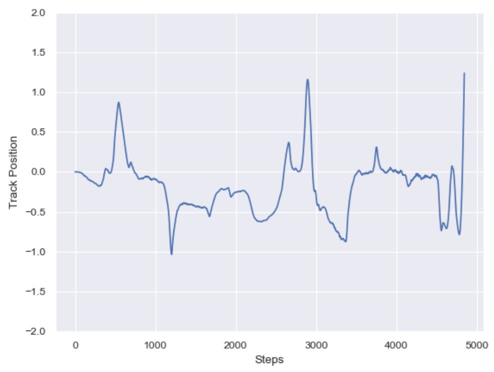
(c) Steering mid exploration



(d) Steering best episode

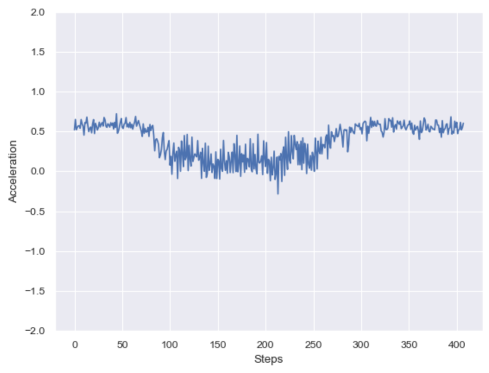


(e) Track position mid exploration

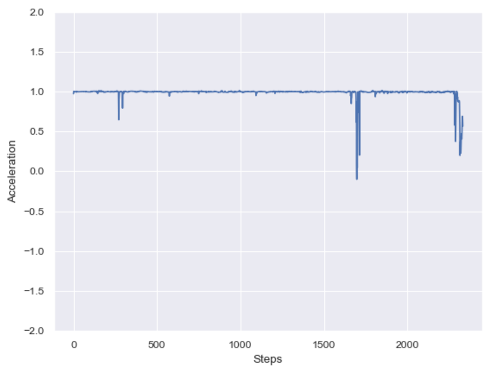


(f) Track position best episode

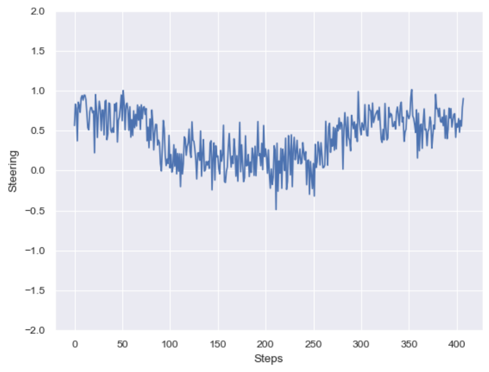
Figure 5.11: Set of graphs representing the acceleration, steering angle and track position recorded by the Mixed Network when racing on five tracks.



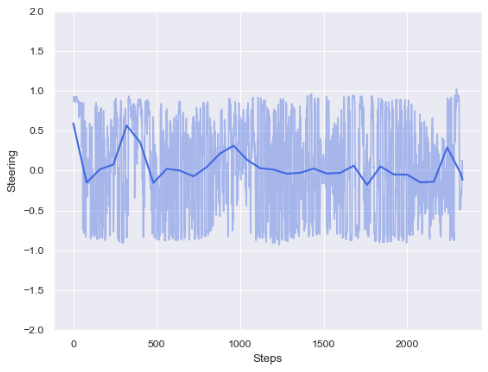
(a) Acceleration mid exploration



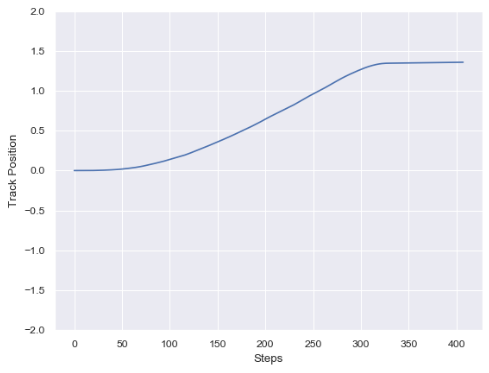
(b) Acceleration best episode



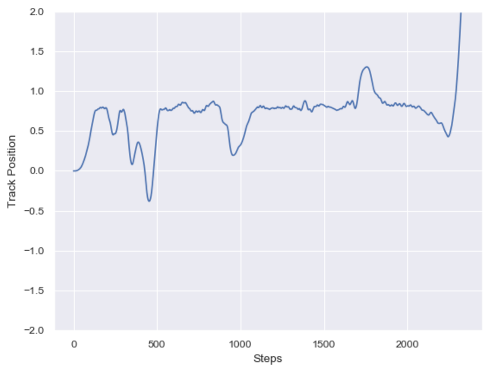
(c) Steering mid exploration



(d) Steering best episode



(e) Track position mid exploration



(f) Track position best episode

Figure 5.12: Set of graphs representing the acceleration, steering angle and track position recorded by the Mixed Network when racing on one track.

5.2 Testing results

After the experiment is completed, we need to test the model that has been generated to observe its capacity of generalization. During the training, every hundred episodes, we saved checkpoints of the model created while learning; these checkpoints save the entire weights configuration of the network so that the model can be later restored with the same structure. After the learning is complete, we select the checkpoint that is closer to the episode where the agent reached the higher distance raced, and we test it by making it run on three unseen tracks for 10000 steps. The overview of these tracks can be seen in Figure 5.13. To select these three tracks we chose the same method we used while selecting the training tracks.
During this test we keep track of the reward that the agent is getting as well as the distance raced and the trajectory it kept during its lap; then we use this result to evaluate the agent performance. This process allows us to detect possible over-fitting problems that could arise during training.

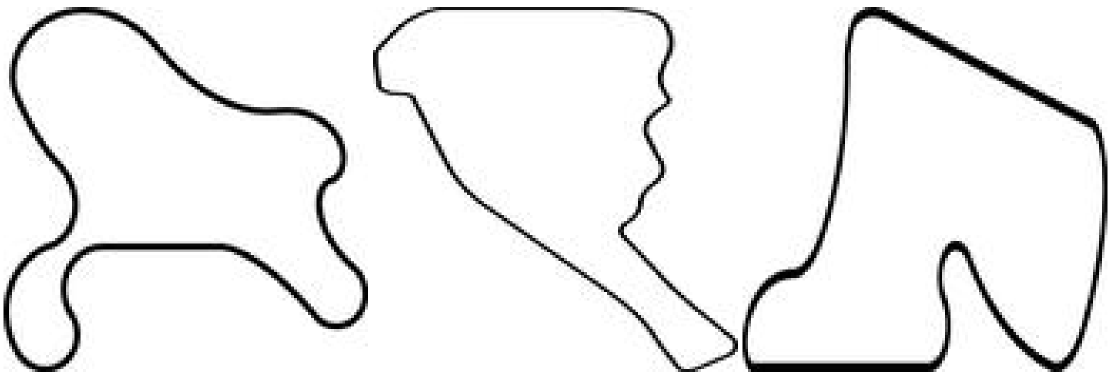

(a) Mueda-city overview (b) Petit overview (c) Ustka-city overview

Figure 5.13: Overview of the three tracks used during the testing process

To examine the performance obtained by the three networks we will first focus our evaluation focus on each network separately and then we will compare the results obtained. To do so we provide a table for each of the networks, containing the total reward obtained when performing the testing on a specific track and also the total distance raced reached by the agents.
To have a baseline to which compare the results obtained by our network we used the SnakeOil client to drive on the same three tracks. This simple controller races around the track for 10000 steps like the models of our agents; this gives us the best situation to compare the results. When racing it tries to match a desired top speed that we set to 200km/h (we want the client to try to go as fast as possible like our three agents) modifying the steering angle, acceleration and brake accordingly. In this case, we recorded only the total distance raced by the client on each of the track because SnakeOil doesn't have a reward and the distance raced is the easier and immediate value for comparison.
In Table 5.1g we can see the results obtained by the SnakeOil client.
We will also provide an image of the entire track with the trajectory kept by the agent, this will allow the reader to visualize the performance of the agent by examining the trajectory it kept during its lap.

5.2.1 Numeric Network

We can see in Table 5.1a and 5.1b that the Numeric Network performed well even when placed on unseen tracks. The model that was trained on five different tracks was able to complete more than one lap when place on the Mueda-city track while it did worse on the other two tracks

stopping very early when tested on the ustka-city track; it is interesting to notice that this agent generally performed better than the one trained on only one track, given this result we can hypothesize that the agent trained on five different tracks has a better generalization than the model trained on only one track.
It is interesting to note that the Numeric Network manages to outperform the SnakeOil client in all of the testing tracks almost doubling the distance raced on the Mueda-city track.
We will now present the trajectory the two models kept during their best performance; it is interesting to notice in Figure 5.14 where the agent completed more than one lap, that the trajectory overlaps almost perfectly so the agent manages to keep a stable trajectory even with the erratic steering wheel movement detected when training (see Section 5.1.1), while this is more visible in Figure 5.15 where we can see an erratic movement during straight lines, this may be the cause of why the agent exited the track bounds at the end.

5.2.2 Image Network

We can see the results of the Image Network testing in Table 5.1d and 5.1c. It is immediately clear that the network performed worse than the Numeric Network as we expected from the training results. We can see that in this case, the performance is similar between the network trained on one track and five tracks. When comparing the trajectory of the best performance when trained on five tracks the network performed a little bit better but we can see from Figure 5.17 and 5.16 that the agent exited the track when facing the first few turns of the race. This can be seen as a confirmation of the hypothesis we made in Section 5.1.2, the agent could struggle to infer its current speed from only images and so it finds strong difficulties when approaching never seen turns.
Unsurprisingly we can notice that the SnakeOil client outperforms this network in all of the three testing track, meaning that the agent doesn't reach the performance baseline we set.

5.2.3 Mixed Network

We will now analyze the performance during testing of the Mixed Network. As explained in Section 4.2.3 this network uses a combination of both images and telemetry data to represent the environment state. We can see from Table 5.1f and 5.1e that the agent improves with the respect of the Image Network in both distance raced and reward obtained. This can be due to the fact that the network now has a numerical representation of the speed at which it is racing so it is able to better adjust the steering when approaching a turn. Even though it is not able to fully complete a lap we can see in Figure 5.19 and 5.18 that the agent is capable of taking a series of turns before exiting the track limits. The results, however, are not quite as good as the ones obtained by the Numeric Network but as oppose to Figure 5.15 we can see that in this case the agent manages to keep a steady trajectory without making erratic movement or sudden changes of direction.
The Mixed Network manages to outperform the SnakeOil client on the petit track while performing worse on the other two tracks, suggesting that this network managed to resolve the issues of the Image Network even if not completely.

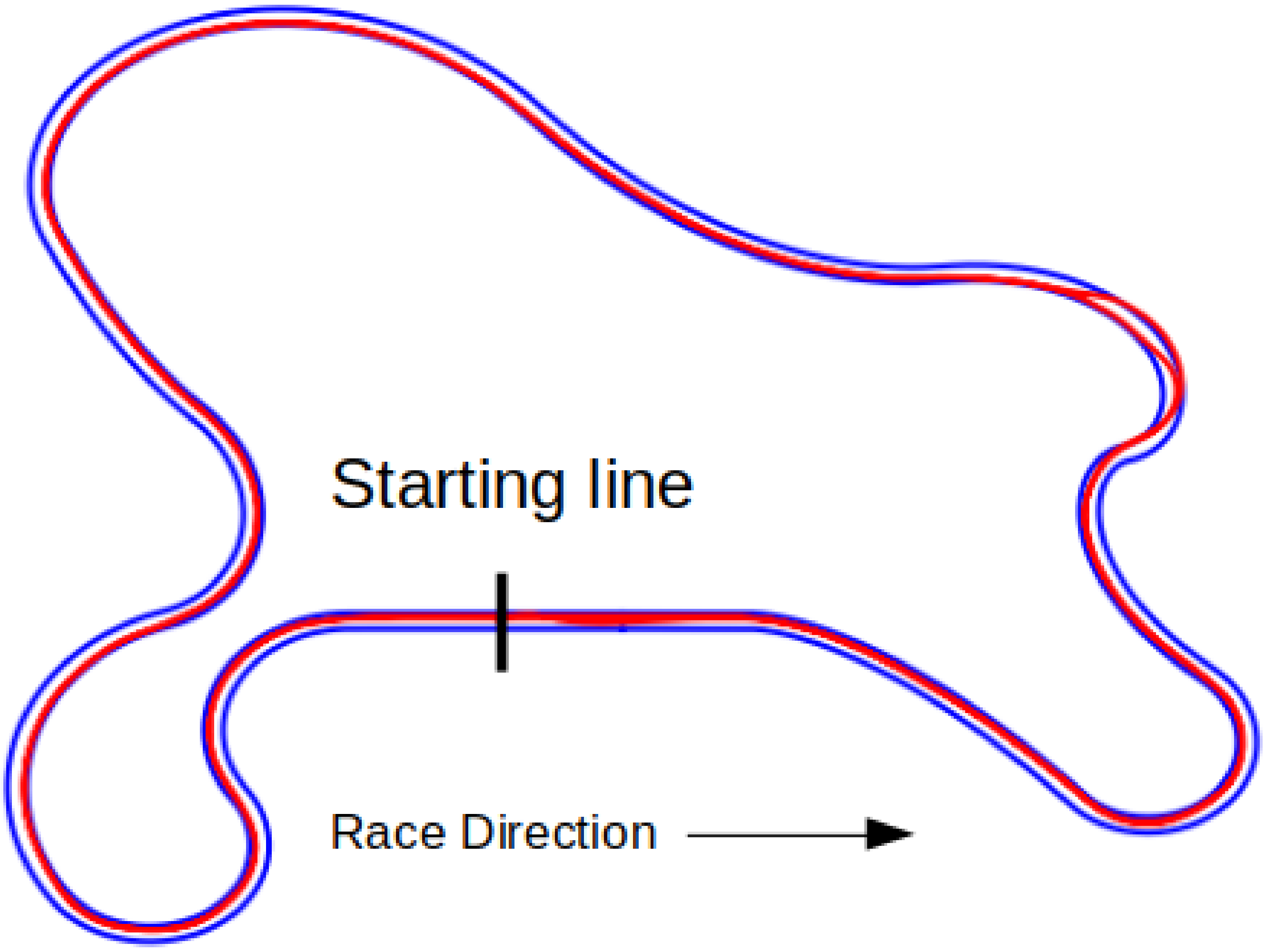


Figure 5.14: Trajectory of the Numeric Network agent trained on five tracks on the Mueda-city track

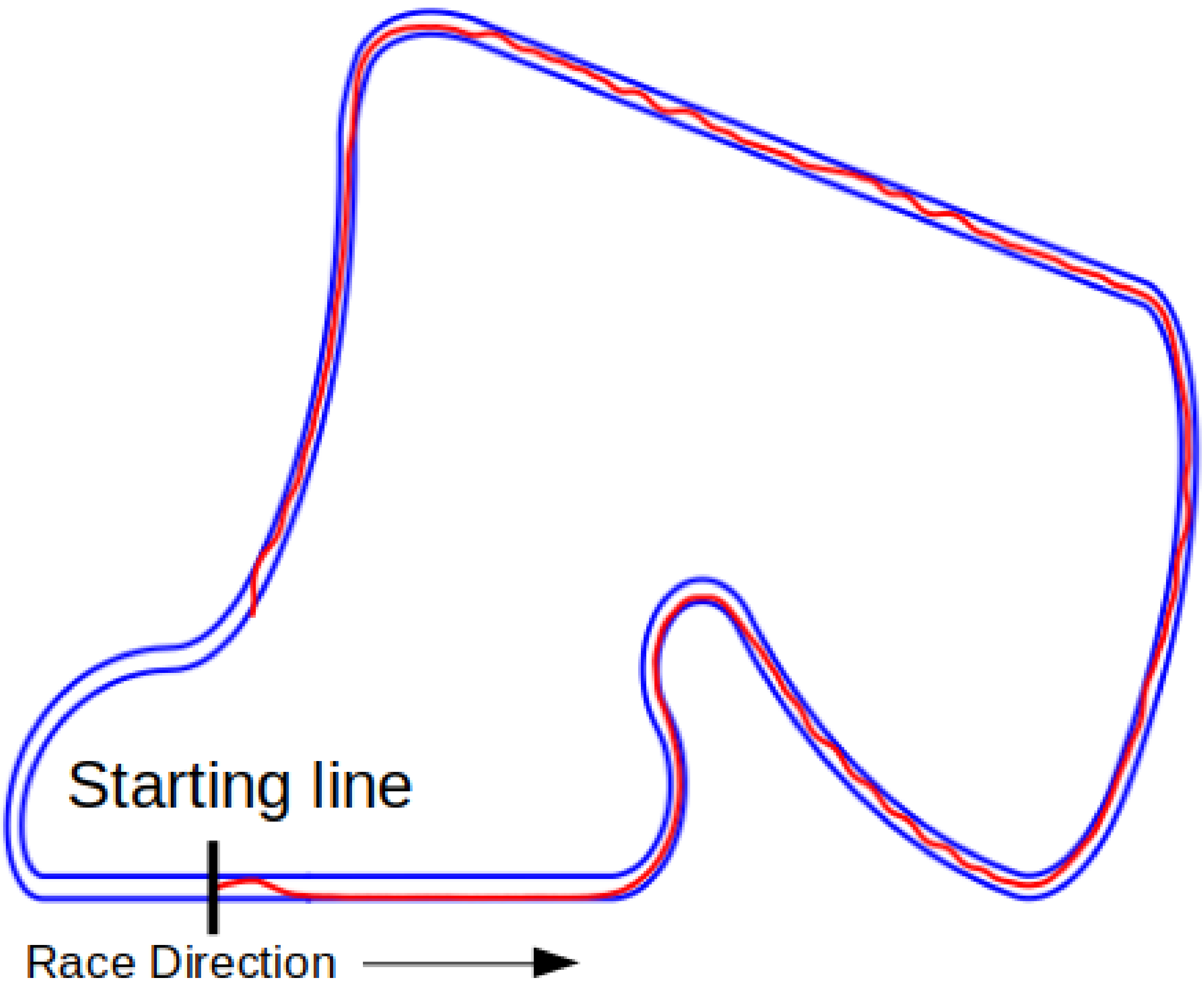


Figure 5.15: Trajectory of the Numeric Network agent trained on one track on the ustka-city track

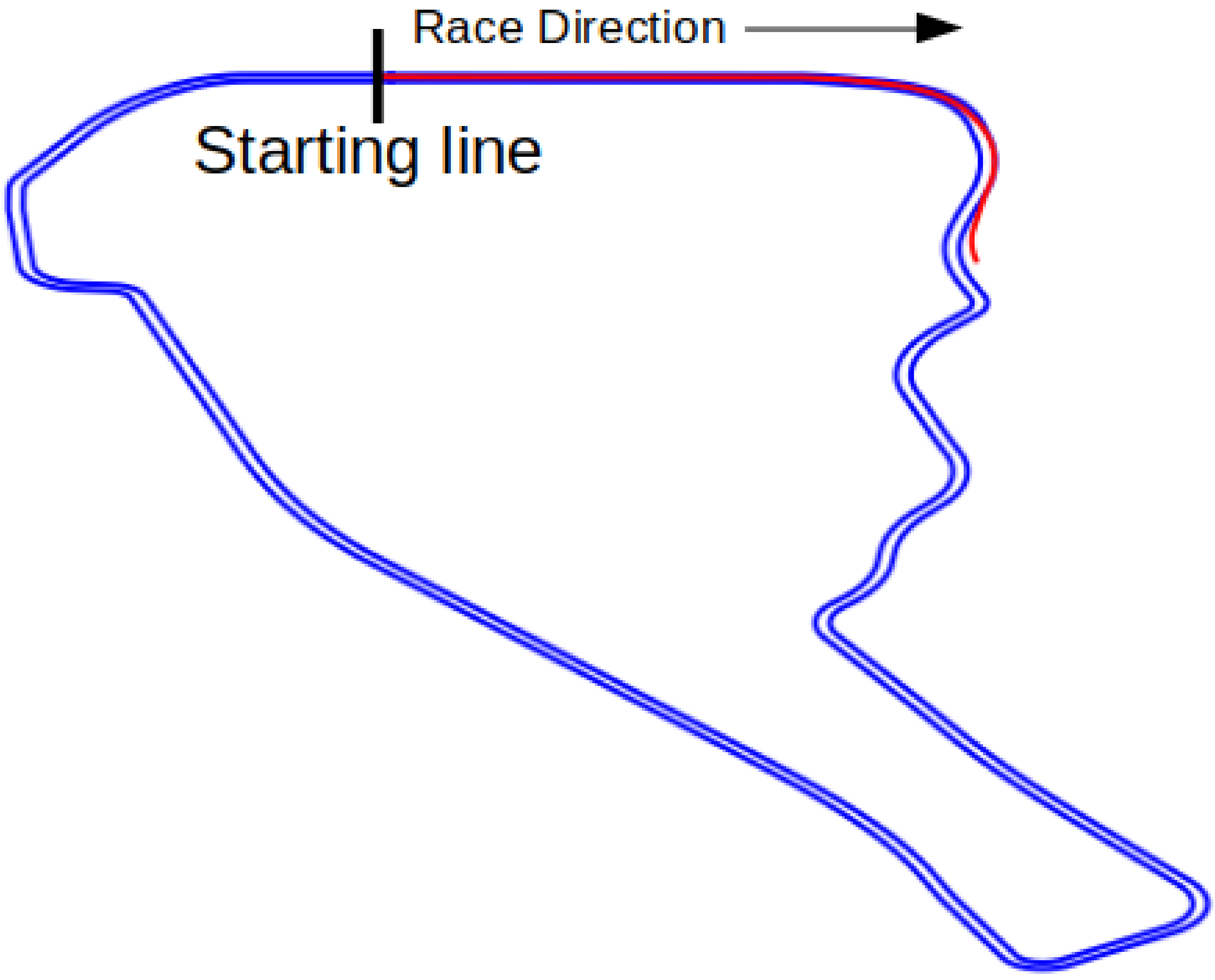


Figure 5.16: Trajectory of the Image Network agent trained on five tracks on the petit track

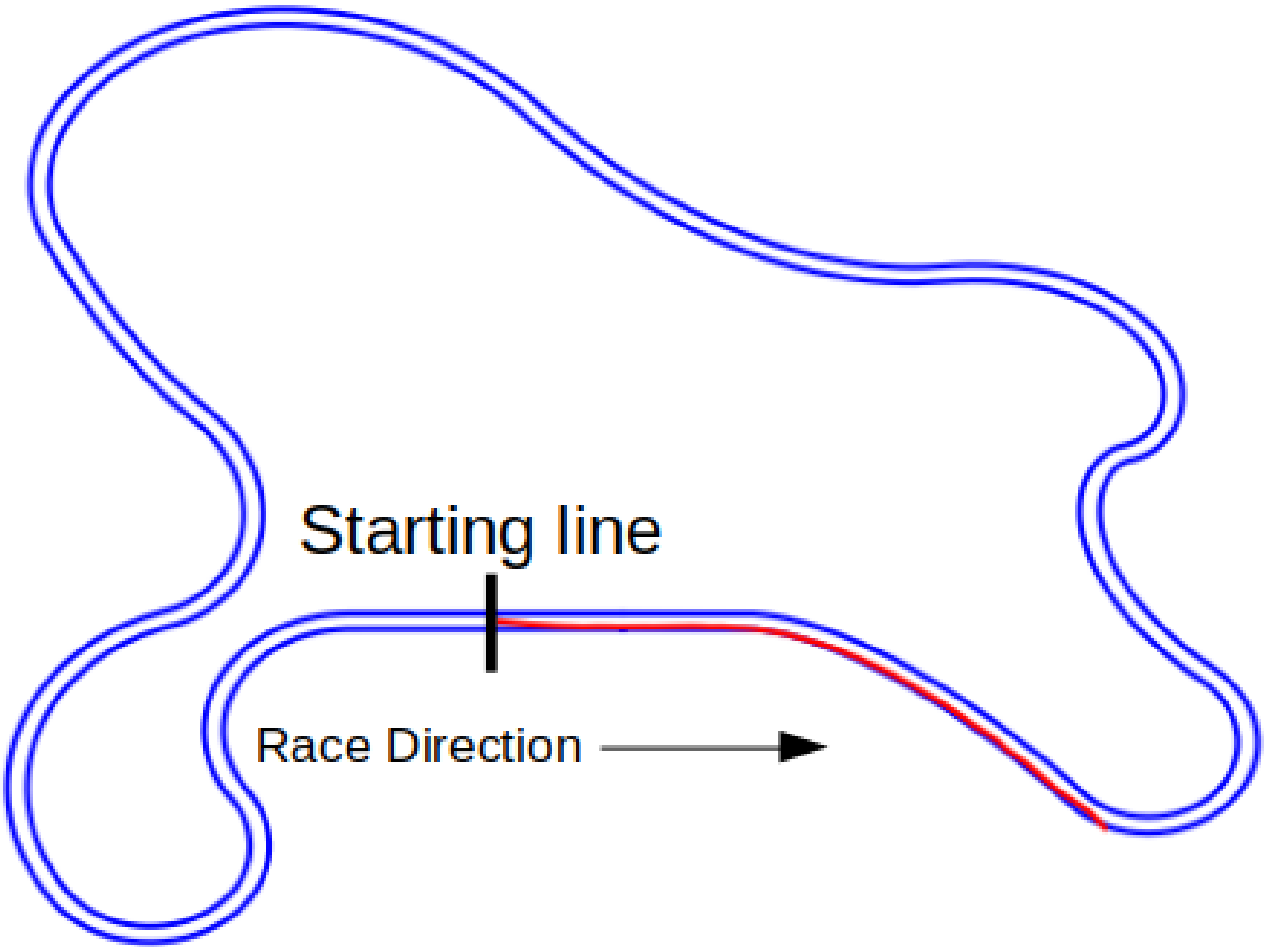


Figure 5.17: Trajectory of the Image Network agent trained on one track on the Mueda-city track

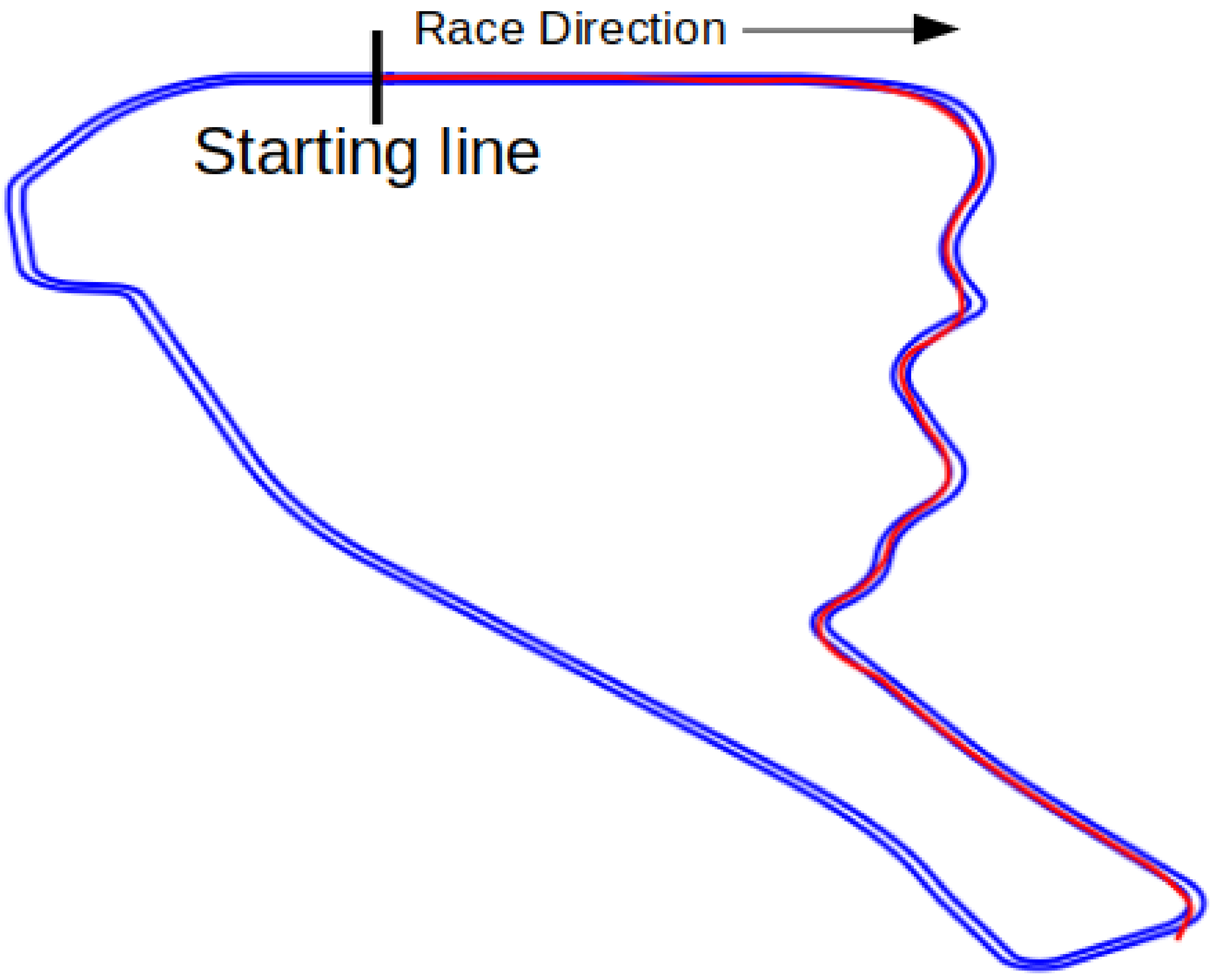


Figure 5.18: Trajectory of the Mixed Network agent trained on five tracks on the petit track

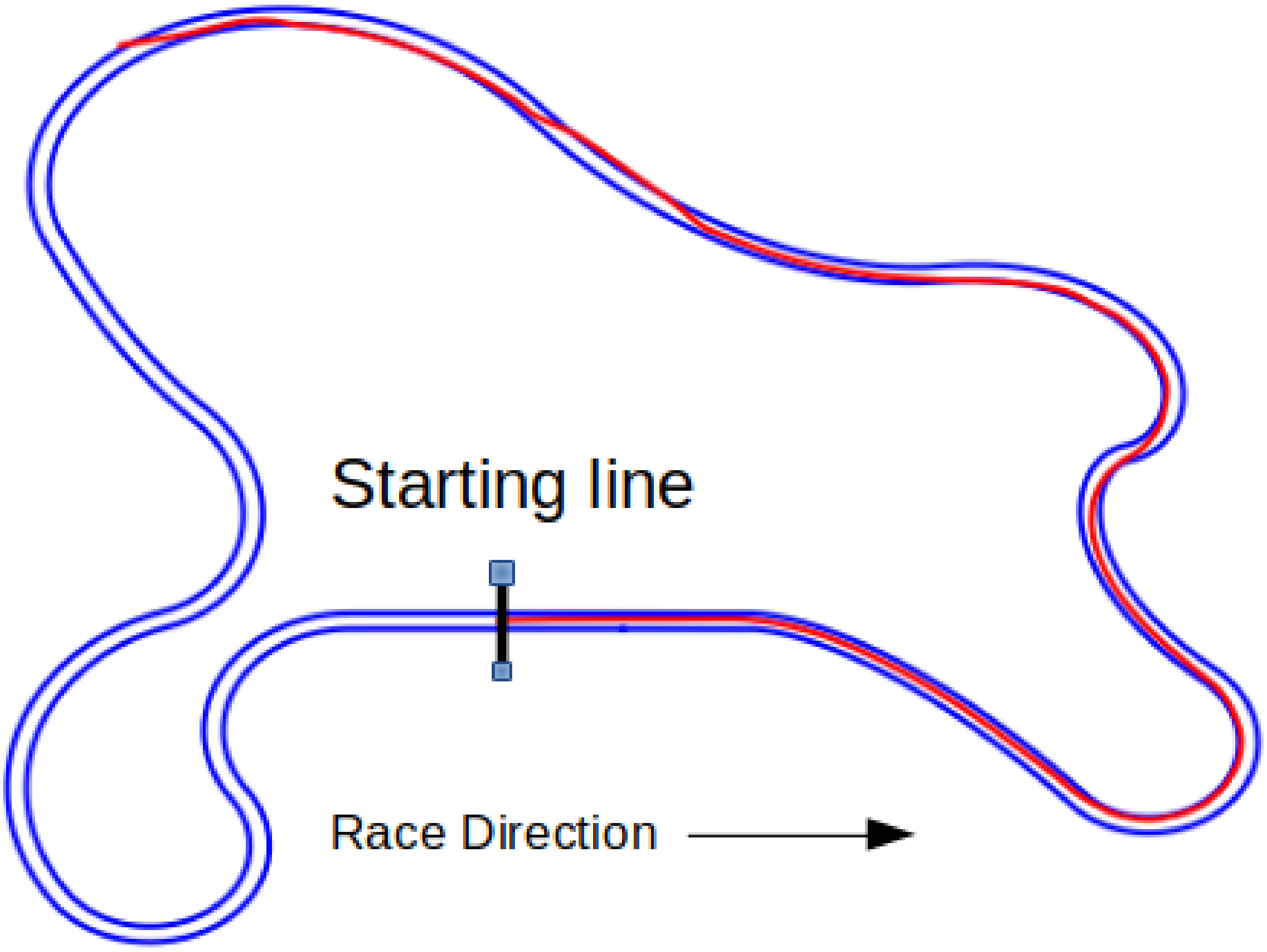


Figure 5.19: Trajectory of the Mixed Network agent trained on one track on the Mueda-city track

Tracks	**Distance**	**Reward**
Mueda-city	2504.6	405007.4
petit	272.3	45618.1
ustka-city	**2761.5**	444307.5

(a) Numeric Network trained on one track

Tracks	**Distance**	**Reward**
Mueda-city	**7853.3**	1339276
petit	2135.5	370208.1
ustka-city	667.6	114208.9

(b) Numeric Network trained on five tracks

Tracks	**Distance**	**Reward**
Mueda-city	535.6	87478.6
petit	414.8	64434
ustka-city	447.2	76410

(c) Image Network trained on one track

Tracks	**Distance**	**Reward**
Mueda-city	497.4	87182.9
petit	716.4	104134.6
ustka-city	293.8	46419

(d) Image Network trained on five tracks

Tracks	**Distance**	**Reward**
Mueda-city	2035.8	319936.6
petit	1317.5	221927.6
ustka-city	553.9	89702

(e) Mixed Network trained on one track

Tracks	**Distance**	**Reward**
Mueda-city	967.5	160949.4
petit	**2005.2**	291754.4
ustka-city	635.6	103987

(f) Mixed Network trained on five tracks

Tracks	**Distance**
Mueda-city	3325.7
petit	1486
ustka-city	2486

(g) SnakeOil client results

Table 5.1: Testing results obtained by our three network on the testing tracks together with the results obtained by the SnakeOil client. In bold the best distances achieved for each track

Chapter 6

Conclusions and Future Work

In this thesis, we studied the applicability of deep reinforcement learning techniques to develop an autonomous agent for the racing simulator TORCS. Starting from the work done in [2], we selected the DDPG algorithm and produced three networks each of them using a different input. The first network uses only numerical data to represent the environment state, this data contains the telemetry at each time-step of the experiment; for the second network we used a custom-built method to generate at each time-step a top-down image of the track to be the input of the neural network; finally the third network combines the inputs described before to have both the image and the numerical data as the state representation.
All three networks were then trained in two different scenarios and then also tested on unseen tracks.
In Chapter 5 we analyzed the results obtained by our three networks in both the training and testing scenarios. We saw from that the Numeric Network managed to produce an agent capable of driving fast around the track while the Image Network struggled to reach the desired outcome. Our analysis suggests that this was due to a lack of information that the network couldn't extrapolate from images only. The lack of knowing its speed caused the agent to fail to take very fast turn effectively, making the agent leave the tracks bounds very early in the learning episode thus causing the agent to never encounter different racing situations to improve its behavior. To test this hypothesis we design an experiment where the agent had to only control the steering angle of the car while the control of the speed was delegated to the SnakeOil client. The results of this network showed an improvement when compared to the Image Network with two outputs, so this inspired the creation of the third and final network developed for this thesis: the Mixed Network. This seems to confirm our hypothesis even though it may be possible that the network improved its performances for other reasons (i.e. it had to control only one output instead of two).
When analyzing the results obtained by the third network we saw that it outperformed the Image Network while at the same time not quite reaching the performance of the Numeric Network. This seems to confirm our hypothesis that adding the speed parameter to the image input helps to increase the network performances; the fact that the Mixed Network doesn't reach the results obtained by the Numeric Network suggests that the information regarding the agent position extracted from the images are not as accurate as the numeric information provided by TORCS telemetry values. This is just an empirical observation and not an experimental result.
Although the models we designed were far from being able to perfectly race along the track, we believe our work is a good starting point towards the development of an autonomous agent able to outperform the current AI bots used in racing games. Even more, we proved that it is possible to create agents able to understand the racing scenarios starting form images without the need of human designers to embed their knowledge into the agent behavior.

6.1 Future Work

In this section, we will present the future work that can be done starting from what we produced for this thesis.
The first possibility is working to improve the Mixed Network performance. This can be done in two different ways, the first one is to improve the image generation process to produce images that contain more information needed by the agent. To do so we think it is feasible to integrate the agent speed into the image of the network, giving the agent more "look-ahead space" according to the speed of the car. This would need the design of a convolutional network able to accept images with different resolutions as the input, in this way we would improve also the performance of the Image network. The second way to improve the Mixed Network is to produce images at a higher resolution. This would allow the network to extract more precise information about its position and could improve the performances.
As we anticipated in Section 4.1.1 it could be interesting to modify the output of the networks, giving the agents a higher level of control by making them predict a future track position and having a dedicated "follower bot" to adjust the actuators of the car itself. It would be interesting to see if the network would produce different results at each time step or it would simply saturate at zero making the car to always stay in the center of the track.
Another possibility is to use supervised learning techniques to develop a first model and then optimize it using reinforcement learning techniques. By combining the two methods we think it is possible to improve the performances of the networks.
We could also try to insert some domain knowledge into the inputs created for our networks. This can be done by creating images containing the best racing line calculated a priori when selecting the track. While it could be interesting to examine the behavior of the agents in this situation it would also mean having the need of previous domain knowledge to develop a working agent.
Finally having developed an agent able to race on an empty track the next step would be to introduce opponents in the racing environment. This is the logical next step to take when creating an artificial intelligence for a racing game. This will introduce new problems to be explored such as agent behavior on start and when overtaking but represents the final goal to be reached to have a competitive autonomous agent that can be implemented in racing games environments.

Bibliography

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

[2] Ben Lau. Using keras and deep deterministic policy gradient to play torcs. `https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html`.

[3] Richard Stuart Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, 1984. AAI8410337.

[4] Watkins CJK. *Learning from delayed rewards.* PhD thesis, 1989. AAI8410337.

[5] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[6] RM Blumenthal. An extended markov property. *Transactions of the American MathematicalSociety*, 85(1):52–72, 1957.

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[8] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[9] R. Bellman and R. Kalaba. *Dynamic Programming and Modern Control Theory.* Elsevier Science, 1966.

[10] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[11] Sergey Ivanov and Alexander D'yakonov. Modern deep reinforcement learning algorithms. *CoRR*, abs/1906.10025, 2019.

[12] Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *J. Mach. Learn. Res.*, 7:2399–2434, December 2006.

[13] Dumitru Erhan, Y. Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *Technical Report, Univerist de Montral*, 01 2009.

[14] Christopher Olah, Ludwig Schubert, and Alexander Mordvintsev. Feature visualization. *Distill*, 2017. https://distill.pub/2017/feature-visualization.

[15] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[16] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[17] Yann Lecun and Yoshua Bengio. *Convolutional networks for images, speech, and time-series.* MIT Press, 1995.

[18] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1008–1014. MIT Press, 2000.

[19] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395. JMLR.org, 2014.

[20] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[21] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[22] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.

[23] P. Serafim, Y. Nogueira, C. Vidal, and J. Cavalcante-Neto. On the development of an autonomous agent for a 3d first-person shooter game using deep reinforcement learning. In *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 155–163, Nov 2017.

[24] K. Shao, D. Zhao, N. Li, and Y. Zhu. Learning battles in vizdoom via deep reinforcement learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–4, Aug 2018.

[25] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. *CoRR*, abs/1609.05521, 2016.

[26] Sebastian Heinz. Using reinforcement learning to play super mario bros on nes using tensorflow. `https://github.com/sebastianheinz/super-mario-reinforcement-learning`.

[27] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[28] Per-Arne Andersen, Morten Goodwin, and Ole-Christoffer Granmo. Deep RTS: A game environment for deep reinforcement learning in real-time strategy games. *CoRR*, abs/1808.05032, 2018.

[29] Eduardo Torres Montano Harrison Ho, Varun Ramesh. Neuralkart: A real-time mario kart 64 ai. `https://github.com/rameshvarun/NeuralKart`.

[30] E. Perot, M. Jaritz, M. Toromanoff, and R. d. Charette. End-to-end driving in a realistic racing game with deep reinforcement learning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 474–475, July 2017.

[31] Ahmad Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017:70–76, 01 2017.

[32] J. Gordon. Javascript racer. `https://github.com/jakesgordon/javascript-racer`.

[33] Eric Espié, Christophe Guionneau, Bernhard Wymann, Christos Dimitrakakis, Rémi Coulom, and Andrew D. Sumner. Torcs, the open racing car simulator. `http://torcs.sourceforge.net/`, 2005.

[34] D. Loiacono, P. L. Lanzi, J. Togelius, E. Onieva, D. A. Pelta, M. V. Butz, T. D. Lonneker, L. Cardamone, D. Perez, Y. Saez, M. Preuss, and J. Quadflieg. The 2009 simulated car racing championship. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):131–147, June 2010.

[35] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship: Competition software manual. *CoRR*, abs/1304.1672, 2013.

[36] Guy Blanc, Neha Gupta, Gregory Valiant, and Paul Valiant. Implicit regularization for deep neural networks driven by an ornstein-uhlenbeck like process. *CoRR*, abs/1904.09080, 2019.

[37] Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. In *Proceedings of the International Conference on Learning Representations (ICLR)*, May 2016.

[38] 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. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016.

[39] Rishi Bedi April Yu, Raphael Palefsky-Smith. Deep reinforcement learning for simulated autonomous vehicle control. Technical report, Stanford University, 2016. `http://cs231n.stanford.edu/reports/2016/pdfs/112_Report.pdf`.