Bachelor's degree thesis

Deep Reinforcement Learning for Autonomous Collision Avoidance

Jon Liberal Huarte

Work advised by:

Ming Liu (HKUST)
Vicenç Puig Cayuela (UPC)

In partial fulfillment of the requirements for the

Bachelor's degree in Mathematics

Bachelor's degree in Industrial Engineering Technologies

May 2020

Abstract

Deep Reinforcement Learning for Autonomous Collision Avoidance

by Jon Liberal Huarte

Collision avoidance is a complicated task for autonomous vehicle control. Most traditional methods in this area consist on model-based solutions, where an understanding of vehicle dynamics and an accurate model of vehicle behavior is required, in order to predict the trajectory of the controlled car and the surrounding vehicles. Such solutions struggle to anticipate and explicitly model surrounding car driving behavior.

This work investigates a model-free Deep Reinforcement Learning based method for collision avoidance, where the agent processes the distances to the closest entities and outputs the steering angle and acceleration required to avoid collisions. A traffic simulator is used to generate a wide range of roads and vehicles, which will interact - not always compliantly - with the learning agent allowing it to collect learning experience. After being trained on such conditions, the agent shows intelligent driving behavior, avoiding areas with high traffic density, adapting its speed to avoid rear or front crashes, and steering when necessary to avoid lateral crashes.

Keywords: Autonomous driving, Deep Reinforcement Learning, Neural Networks, Robot Control, Collision Avoidance, Deep Deterministic Policy Gradients.

MSC code: 93C85

Resumen

Prevención de colisiones en coches autónomos mediante aprendizaje por refuerzo

por Jon Liberal Huarte

La prevención de colisiones es una tarea compleja en el control de vehículos autónomos. Los métodos tradicionales utilizan modelos explícitos para predecir la dinámica de los vehículos e intentar anticipar las decisiones de control de los conductores en el entorno. Estos modelos no siempre consiguen predecir con éxito la trayectoria de los objectos dinámicos en el entorno del vehículo controlado.

Esta tesis investiga un método de control basado en el aprendizaje profundo por refuerzo. El agente procesa las distancias detectadas desde el vehículo controlado a los objectos más cercanos, y mediante redes neuronales estima la acción de control óptima para evitar colisiones. Para el aprendizaje, se diseña un simulador de tráfico que genera un amplio rango de carreteras y vehículos, que interactúan - no siempre siguiendo las normas de circulación - con el vehículo de control, lo que permite al agente recabar información diversa y asociar a cada estado una acción de control que minimice el riesgo de colisión. Tras el entrenamiento, el agente demuestra haber aprendido a evitar circular en áreas con alta densidad de tráfico, a adaptar su velocidad para evitar colisiones frontales y traseras, y a realizar giros que eviten choques con vehículos que se aproximan por los laterales.

Palabras clave: Conducción Autónoma, Aprendizaje por Refuerzo Profundo, Redes Neuronales, Control Automático, Prevención de Colisiones, Deep Deterministic Policy Gradients.

Código MSC: 93C85

Acknowledgements

First, I would like to thank my HKUST supervisor Ming Liu for inviting me to the Robotics and Multi-Perception Lab and for the opportunity to research a diverse range of robotics topics. I would also like to thank Jianhao Jiao for his help in the beginning of my stay, for providing me interesting research topics and supporting my research initiatives. I woud like to express my gratitude for Xiaodong Mei, who has advised me in discovering the field of Reinforcement Learning.

Second, I am grateful for the support received from Prof. Vicenç Puig at UPC, who has offered his generous guidance during my stay at HKUST, and his feedback for this thesis.

I would like to direct a heartfelt acknowledgement to CFIS. I sincerely value the trust that Fundació Privada Cellex has placed in me and my fellow students during my studies at CFIS. Special thanks to Miguel Ángel Barja, for his eager willingness to help the students not only during our studies but also in our future endeavors. Furthermore, many thanks to Toni Pascual, who has followed with empathy and caring interest our stay in Hong Kong during these trying times.

Lastly, I would like to thank my family for their cheerful support all these years.

Jon Liberal Huarte
Navarra, May 4, 2020

Preface

The work for this thesis has been conducting at the Robotics and Multi-Perception Lab at Hong Kong University of Science and Technology (HKUST). I have been able to investigate a wide range of Machine Learning and Robotics research areas:

1. **Image stitching.** The Waymo Open Dataset provides images recorded by its fleet of autonomous cars. For each timestep, a set of 5 images provides visual information from different angles, and Image Stitching methods provide a complete panorama image of the car surroundings. This was my work for the first month of my stay.

2. **Reinforcement Learning for Robot Control**. A novel *learning by trial and error* approach to the Collision Avoidance problem. It is the work presented in this thesis. For this project a new traffic simulator was created, which was the open sourced at [17].

My studies at UPC have proved to be essential to tackle the challenges that arised during the project. Specifically, the Mechanics, System Dynamics, Automatic Control, Machine Mechanisms Theory and Informatics courses in the Degree in Industrial Technology Engineering; and also the Statistics, Probability, Algorithmics, Calculus, Differential Geometry, Ordinary Differential Equations courses in the Degree in Mathematics.

Contents

List of Figures

List of Acronyms

APF	Artificial Potential Field
CLP	Conformal Lattice Planner
DDPG	Deep Deterministic Policy Gradients
DNN	Deep Neural Network
DRL	Deep Reinforcement Learning
DL	Deep Learning
KBM	Kinematic Bicycle Model
MDP	Markov Decision Process
NN	Neural Network
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SAC	Soft Actor Critic
TRPO	Trust Region Policy Optimization
VFF	Virtual Force Field
VFH	Vector Field Histogram

Chapter 1

Introduction

1.1 Motivation for Deep Reinforcement Learning

Autonomous control systems must perform well even when dealing with significant uncertainties in the system and environment. They must be able to robustly compensate corner cases and system failures, without external intervention. This requires intelligence, understood as the ability to solve a wide variety of complex problems. As described in [23], autonomous control systems require:

(a) **Conventional control methods:** Algorithmic-numeric methods, designed to model and analyze the dynamical behaviour of the autonomous systems and its environment.

(b) **Decision making symbolic methods:** AI methods, able to manage high-level mission planning and decision making, similar to traditional automata.

Neural Networks

The accute differences in the nature of (a) and (b) type methods suggest that they should be approached using different techniques. However, in recent years Neural Network (NN) architectures have shown unparalleled ability at function approximation, regardless of the nature of such functions. Deep Neural Network (DNN)s have shown capacity for learning logical reasoning [24], and for learning the dynamics of complex physical processes, such as object tossing [2] or fluid simulation [6].Therefore, it is natural to assume that DNNs might be able to incorporate both (a) and (b) type functions.

The surge in computing power

Yet training such networks requires data. Collecting experience for any physical task has been traditionally expensive. For most robot control tasks, real world experience collection is extremely inefficient. Reinforcement Learning (RL) methods require millions of samples, and obtaining such information from the real world takes time, and implies risks, such as hardware damage.

For the last decades, computing power has increased roughly according to Moore's Law. This has enabled time-efficient and accurate real world environment simulation. Simulations that are often several orders of magnitude faster than real world experience collection, while being also more cost effective in terms of material components and technology.

Time-efficient simulation allows efficient experience collection, providing RL methods with the training data that they have lacked since they were theoretically developed in the 80s.

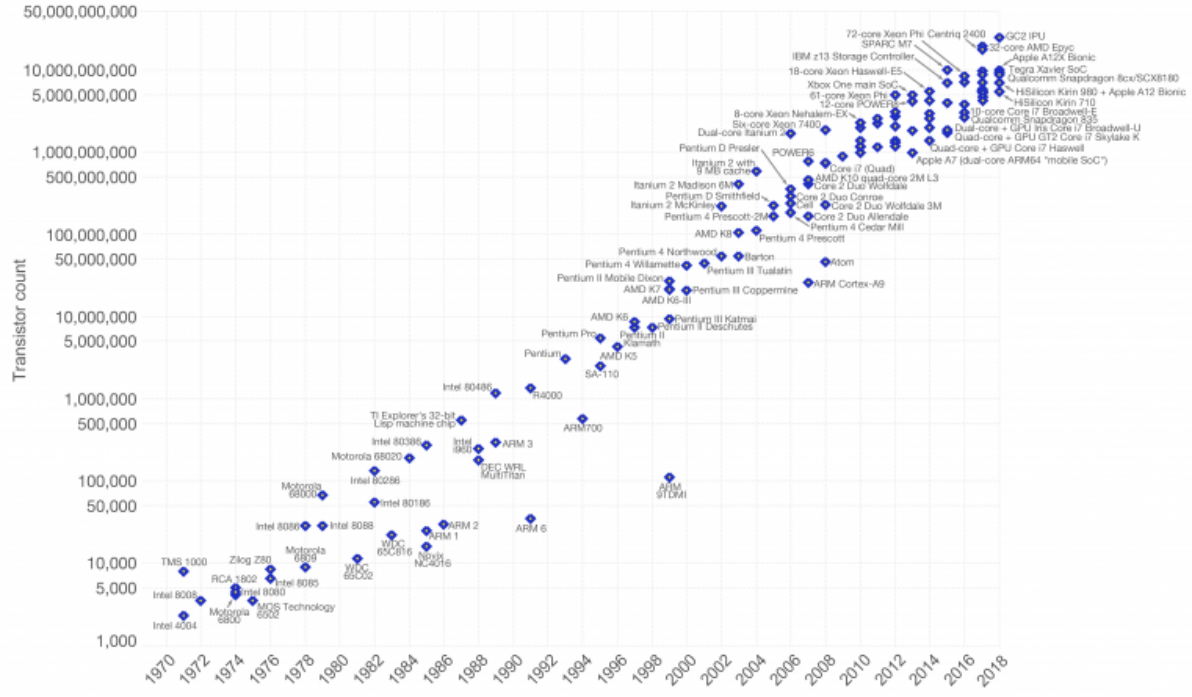


Figure 1.1: Moore's Law - The number of transistors on integrated circuit chips, reprinted from [26].

Learning by trial and error

The idea that an autonomous system can master a complex task just by trial-and-error sounds intuitive to a human. The human mind is designed to learn that way, it is therefore only logical to us that an intelligent entity is able to master tasks provided with enough experience.

Technically, any interaction with an environment provides information about the underlying processes. If the subject is able to partially perceive such information, it is safe to assume that it can attain some level of understanding from it. It can be concluded that;if the subject is able to broadly gather a diverse subset of the total information, and; if it is able to aggregate each sampled information gain, the subject should be able to form a solid capacity to interact with such environment.

1.2 Application to Autonomous Driving

Autonomous Driving methods rely heavily on hierarchical planning techniques; separate methods, each carefully designed for the specific task assigned. For such a complex task, it would be naive to claim that such specific implementation can be substituted by a simple RL technique.

RL as an assistant

Yet a RL approach can be helpful as an assistant method. Blindly trusting solely Deep Learning (DL) techniques is dangerous, as interpretability of NNs is extremely limited. Interpretability research is becoming an important area of DL research, as explained in [12]. However for the moment, the output of DNNs for a each specific input depends on millions of parameters and it cannot be expected for any human to infer such output.

This work can therefore be used as an auxiliary helper method that can partially influence decisions when principal techniques output unclear commands, or when sensor failure brings down a part of the autonomous driving system.

1.3 Document overview

This work is structured in the following chapters:

- Chapter 2 explains the research relevant to this work scope, explaining both the challenges to overcome and the state-of-the-art techniques to solve them.
- Chapter 3 provides some introductory level concepts of RL that are required to understand this work.
- Chapter 4 describes the simulator that has been designed to train and test the RL agent responsible for collision avoidance presented in this thesis.
- Chapter 5 goes through the implementation of the RL agent, as well as the baselines for performance comparison.
- Chapter 6 analyses the performance of the agent, and reflects on future possibilities that could spin off this work.

Chapter 2

Related work

This chapter refers to previous research related to the contributions of this work, which could be considered the state-of-the-art in the fields of collision avoidance and RL.

2.1 Collision Avoidance for Mobile Robots

Motion planning for mobile robots has gained interest in recent times. Navigating an uncertain environment has been a major challenge in mobile robotics, and diverse effective techniques have been proposed. This section presents a compendium of methods for autonomous collision avoidance.

2.1.1 Virtual Force Field

This approach [5, 8, 1] consists of considering the robot as a particle subject to forces that repel it from obstacles, and that attract it to the navigation target.

Variants of Virtual Force Field (VFF) have been proposed for unmanned aerial navigation, ship navigation, and autonomous driving.

Artificial Potential Field

First proposed by Khatib [18], Artificial Potential Field (APF) is the most basic form of Virtual Force Field robot control. It consists of considering the potential field resulting from the superposition of repelling virtual forces generated by the obstacles. The desired navigation direction is incentivised by considering a potential field that pulls the robot in

such direction. After computing the total potential field around the robot, the gradient of the field determines the desired direction of the robot.

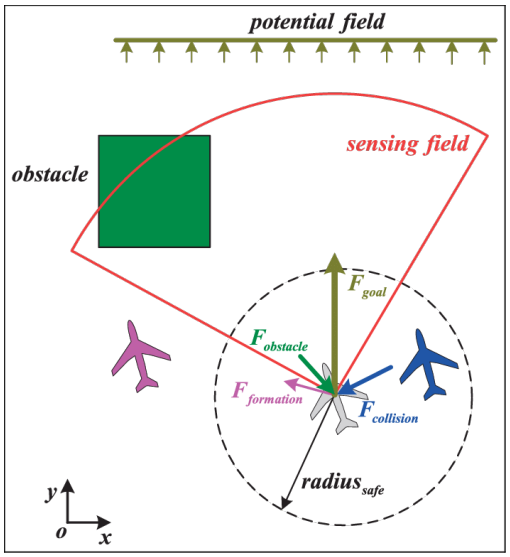


Figure 2.1: Artificial Potential Field for air navigation, reprinted from [13].

2.1.2 Vector Field Histogram

This approach [5] was presented as an improvement on VFF, and mainly aims to tackle the unstable motion that characterizes VFF methods.

Vector Field Histogram (VFH) uses certainty grids [21] and occupancy grids [7] to compute a probabilistic occupancy map, and then generates a polar histogram to determine the obstacle density for each direction. The valley in the histogram that lays closest to the target direction is then selected as the optimal direction.

VFH fails to consider dynamic obstacle motion, and is designed for holonomic robots. It is however more stable than VFF methods and the robot tends to get stuck less.

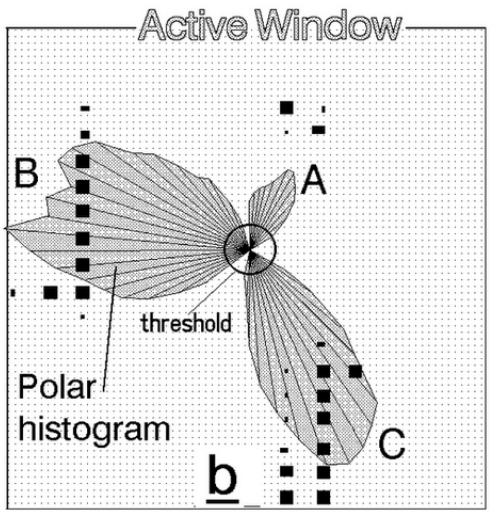


Figure 2.2: Visual Representation of Vector Field Histogram, reprinted from [5]
Visual Representation of Vector Field Histogram, reprinted from [5]. Directions facing obstacles A, B and C represent higher obstacle density in the histogram.

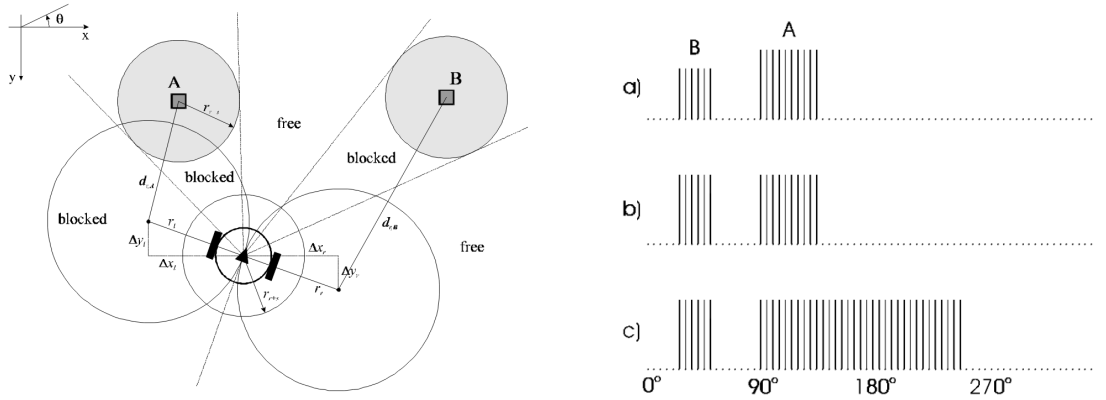
VFH+

Presented in [30] as an improvement on VFH [5]. VFH+ is different from VFH mainly in the sense that it considers robot dynamics. This is important for our case as cars are non-holonomic vehicles.

The histogram generated in VFH is modified to represent the obstacle density of each possible trajectory arc that the robot can perform. This modified histogram is restricted therefore by robot motion kinematics.

The optimal steering direction is then selected by choosing the valley in the modified histogram (see histogram (c) in 2.3) that lays closest to the goal angle.

This alternative fails to predict the motion of dynamic obstacles.



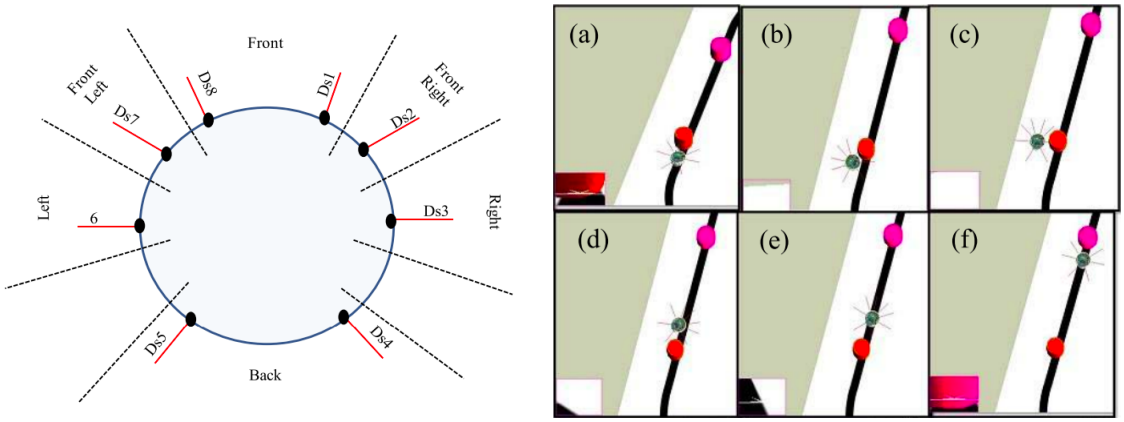
(a) Example robot layout.

(b) The 3 stages of polar histograms for VFH+ steering angle selection.

Figure 2.3: Polar histogram generation from an example obstacle layout, reprinted from [30].

2.1.3 Line Following

These solutions [9, 19, 31] are more recent and more specific for autonomous driving vehicles. Reliable solutions have been proposed for road detection and line following in autonomous cars. Collision avoidance is then performed by identifying obstacles that obstruct the desired trajectory.



(a) Laser sensor distribution.

(b) Sequence of line following based obstacle avoidance

Figure 2.4: Line following obstacle avoidance presented in Almasri et al, reprinted from [19].

In Almasri et a [19], a line following technique is presented for holonomic robots; line following

is performed until front laser sensors detect an obstacle. In that case, the robot turns until laser sensor readings reach a safe threshold, and navigates around the object until returning to the reference line is possible. A careful tuning of each sensor threshold is required to achieve satisfactory obstacle avoidance.

These line following based methods are design for static environments and also fail to anticipate dynamic obstacle motion.

2.1.4 Conformal Lattice Planner

Perhaps a more driving specific method is the Conformal Lattice Planner [20]. Applied to the autonomous driving problem, this approach consists first on picking goal points that are laterally offset further down the road.

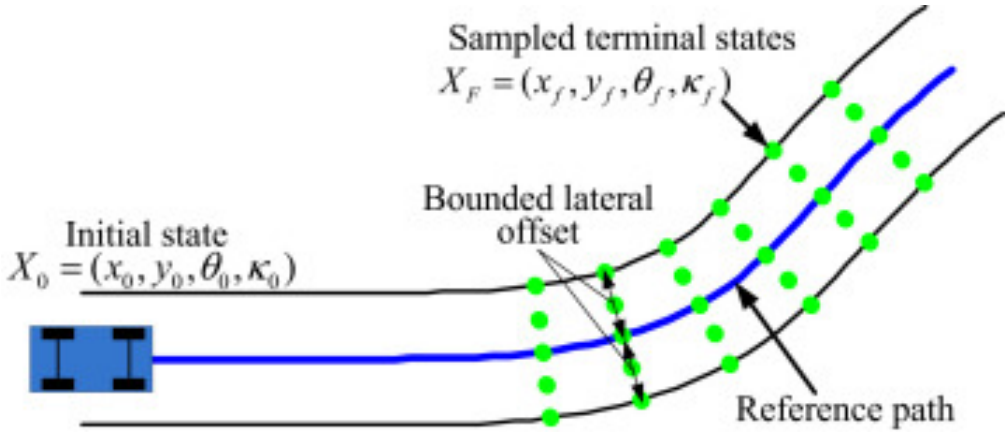


Figure 2.5: Conformal Lattice Planner, reprinted from [32].

Then, a spline trajectory is computed from the car to each of the points (green points in Figure 2.5). A cost function evaluates the *goodness* of each of these trajectories, according to criteria such as collision with obstacles, minimizing curvature, reducing drastic speed changes and distance to the reference line. Once the optimal trajectory is selected, the control action that corresponds to that trajectory is considered the optimal control input.

Conformal Lattice Planner has been the main technique for planning these recent years, and can be considered state-of-the-art. However, it still requires dynamic object behavior modelling to anticipate obstacle motion in order to avoid collisions. As this is the approach that best relates to the task tackled in this work, Conformal Lattice Planner is the baseline used for performance comparison.

2.2 Deep Reinforcement Learning

The field of Deep Reinforcement Learning has become a considerable branch of Deep Learning research in recent years. The progress in supervised learning (image classification, object detection, medical diagnosis, Natural Language Processing, pose estimation, etc.) has outpaced the progress in Deep Reinforcement Learning. The idea that an initially clueless entity can learn the structure of a complex process just by interacting with such process is promising, yet challenging to implement, as many practical difficulties arise.

2.2.1 Challenges in modern RL

The credit assignment problem

It is not straightforward to attribute merit to the actions executed by the agent. Some actions might yield a reward long after they are executed. After performing a sequence of actions, the reward received cannot be certainly attributed to any action in particular. Standard Deep Reinforcement Learning (DRL) techniques tend to be error prone in environments with delayed rewards (like Go, or chess). Estimating which actions to reinforce therefore requires more sophisticated methods.

The exploration-exploitation trade off

This trade-off is not only specific to RL tasks. The exploration-exploitation trade-off is studied in human cognitive psychology [27, 4], and arises whenever a subject needs to choose between acquiring new information, or settling for the best-so-far option. As a relatable example, the exploration-exploitation trade-off arises whenever a person is at a restaurant and doubts between the dish that she likes best so far, and a dish she has never tried before.

In RL, if the agent is only driven by maximizing environment rewards, it might settle for suboptimal actions early in the training process, renouncing to explore better actions that are incorrectly estimated to be less profitable. On the other hand, strongly motivating it to explore might lead to failed training convergence, as the actions executed might be too random to extract useful information about the environment. For complex environments, the state space is just too broad. It is not computationally feasible to explore the whole state and action space, the agent must restrict itself to a relatively small set of spaces. Balancing how much the agent should explore is key to satisfactory training.

Reward shaping

Each DRL application consists of a human goal. Transferring this goal from words to a consistent reward is challenging. If the reward is not aligned with the original human goal, the agent might find alternative ways to maximize the reward without actually fulfilling its original task. For example, it might find inconsistencies in the training simulators (such as in OpenAI Hide and Seek [3]), or glitches in the reward systems.

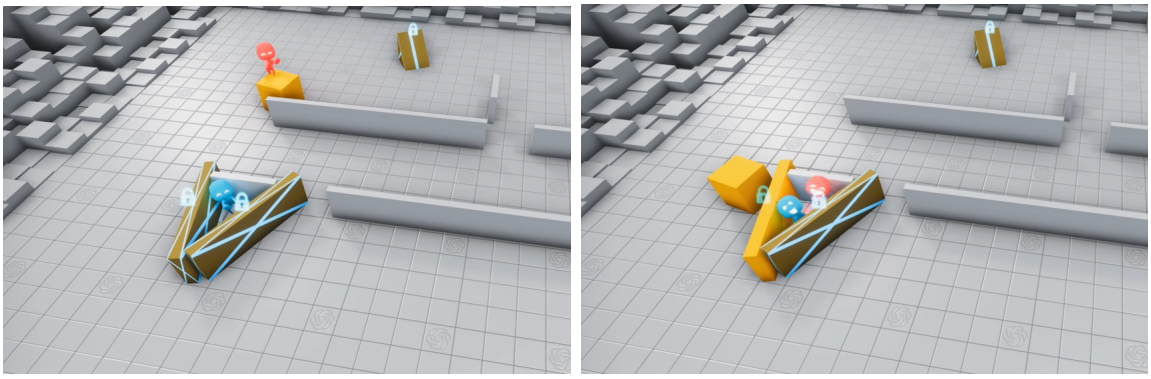


Figure 2.6: In OpenAI Hide-and-Seek [3], the seekers exploit a bug in the physics engine that enables them to surf boxes to jump into the hiders' shelter.

Automatic reward shaping has been studied [11], but so far it is not considered to be applicable for most approaches. For most DRL problems, rewards are specifically designed by researchers. For this thesis, this is also the case.

Sample inefficiency

Training experience is needed for any Reinforcement Learning training task. Such experience is usually obtained by simulation. Some off-policy algorithms theoretically allow to reuse training samples, but even so, DRL algorithms reuse samples dramatically less than other Deep Learning methods, where the training runs through the complete dataset at each epoch. Improving sample efficiency has been a major goal for recent research, some DRL algorithms (such as Soft Actor Critic (SAC)) show outstanding efficiency improvements.

2.2.2 State-of-the-art overview

A variety of techniques have succeeded to solve a diverse range of problems. The following methods are considered the current state-of-the-art DRL techniques:

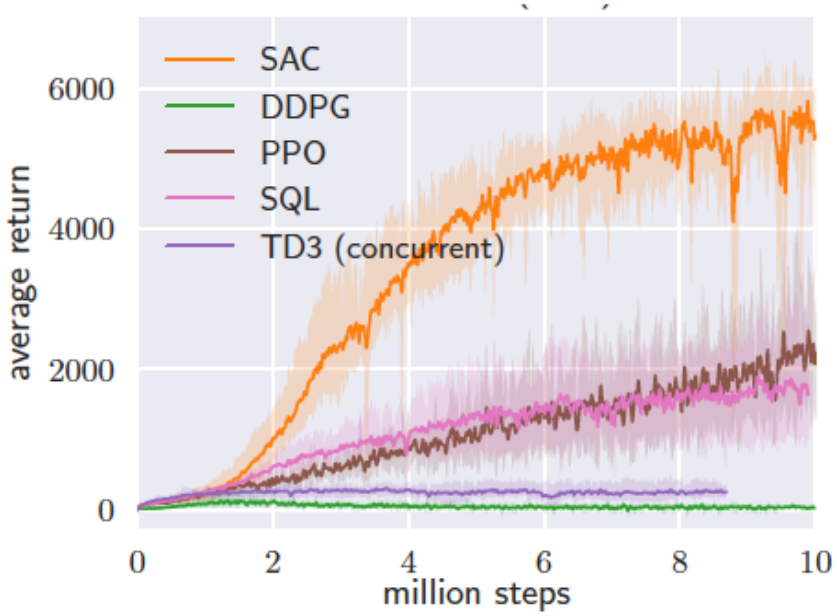


Figure 2.7: Sample efficiency of several DRL algorithms for Humanoid Locomotion Learning, reprinted from [28].

Soft Actor Critic

Actor-Critic methods consist of two separate models; the Actor, that decides on the optimal action, and the Critic, which evaluates how good that action is . Haarnoja et al [28] published a variant of Actor-Critic methods that motivate action space exploration. First, SAC is similar to Deep Deterministic Policy Gradients (DDPG) in the sense that both are designed for continous action spaces (which is our case). Therefore, the set of possible actions is no longer finite and discrete.

The *soft* term in Soft Actor Critic refers to a tweak in the objective reward function: instead of greedily maximizing the expected reward, an *entropy* term is added to motivate action randomness when possible. The optimal action is no longer the one with the highest traditional Q-value, and actions that yield similar rewards but allow for greater variance in the action command are prioritised over actions that yield good results only in a narrow action range.

This method is considered state-of-the-art for Reinforcement Learning application, specially when the action space is large, as in the Humanoid Locomotion problem, where there are 21 action dimensions.

Proximal Policy Optimization

Presented in Schulman et al[15], Proximal Policy Optimization (PPO) is based on Trust Region Policy Optimization (TRPO) theory [16].

The motivation is the following. When training Policy Gradients methods, it is common to see the performance of the agent collapse suddenly. This is attributed to the fact that weights in the policy network are repeatedly pushed out of the safe region where numerical stability is guaranteed, until network outputs reach a singularity and start outputting nonsense.

TRPO is a Policy Gradients method that solves this stability issue by restricting weight updates to a *trust region* that can be computed for each sample.

PPO is an approximation of TRPO that shares the idea of restricting weight update magnitude, while simplifying the implementation and keeping performance.

Deep Deterministic Policy Gradients

DDPG was presented in [29], as an adaptation of Q-learning methods to continous action spaces.

The idea behind this approach is based on gradient ascent. As the action space is continuous, the Q function can be assumed to be differentiable with respect to the action space. Therefore, the gradient of the Q function with respect to the action a_t; $\nabla_{a_t} Q$, can be used to estimate the direction from a_t towards the optimal action a_t^*.

The policy network is therefore updated to progressively predict actions that lay closer to the optimal action a_t^* at each sample.

Chapter 3

Reinforcement Learning: essential concepts

This chapter presents the most important elements of the RL framework. We first present a mathematical model of the general RL problem, then we go through the main elements in the RL framework. The last section explains the two main approaches for training policies towards optimal performance.

3.1 Mathematical foundations

For the sake of mathematical tractability, it is useful to model the RL problem as a mathematically idealized form of such problem, for which precise theoretical statements can be made. We need a mathematical object that captures the set of states that can take place, the nature of the transitions between such states, and the rewards associated with such transitions.

3.1.1 Markov Decision Process

A Markov Decision Process (Markov Decision Process (MDP)) is a discrete time stochastic control process, and provides a mathematical framework for modelling decision making, in situations where outcomes depend partially on the decision taken.

First, an MDP consists of an Agent - Environment interface:

- Agent: the learner and decision maker.

- Environment: Everything outside the agent, everything that the agent interacts with.

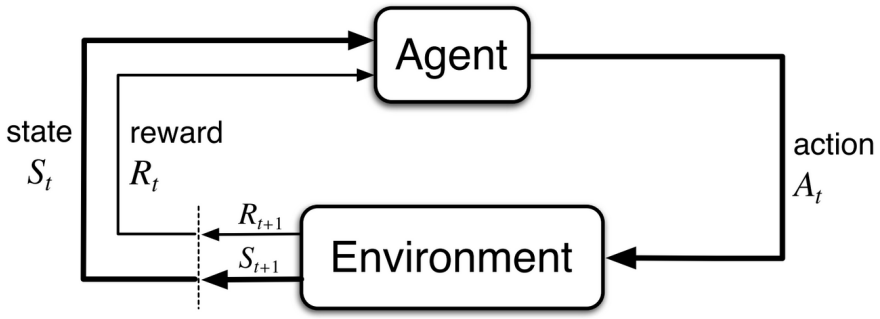


Figure 3.1: Agent-Environment Interaction in a Markov decision process.

The interaction between the agent and its environment can be defined more specifically. At each time step, the agent receives a representation of its environment's state, $S_t \in \mathcal{S}$. Based on that state, the agent selects an action $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ is the set of feasible actions at state S_t. As a result of the action selected, one time step later, the agent receives a numerical reward $R_{t+1} \in \mathcal{R}$, and finds itself in a new state S_{t+1}.

Concatenating such interactions gives rise to a *trajectory* comprised of states, actions and rewards:

$$S_0, A_0, R_1, S_1, A_1, R_2, ..., S_T, A_T, R_{T+1}$$

This trajectory is not completely random. Performing a particular action at a particular state usually has an influence on the reward and new state that arise. R_t and S_t are therefore random variables whose probability distributions depend on the preceding state and action. For any preceding state $s \in \mathcal{S}$ and action $a \in \mathcal{A}(s)$, the probability of reaching state $s' \in \mathcal{S}$ and obtaining a reward $r \in \mathcal{R}$ at the next time step is given by the probability distribution function p:

$$\begin{aligned} p\colon \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} &\longrightarrow \mathbb{R} \\ (s', r, s, a) &\mapsto Pr(S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a) \end{aligned}$$

It is worth noting that this function p **completely defines the dynamics of the MDP**. The RL training task therefore consists on estimating p by interacting with the environment

and observing the transitions that take place.

From p, we can also compute useful functions for our RL task. For example, we can compute an estimate of *how good* an action a is at a specific state s, by computing the expected immediate reward $r(s,a)\colon \mathcal{S}\times\mathcal{A}\longrightarrow\mathbb{R}$,

$$r(s,a) \coloneqq \mathbb{E}[R_t|S_{t-1}=s, A_{t-1}=a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s',r|s,a)$$

3.1.2 Policies and rewards

Now that we have defined our MDP, it makes sense to define the decision taking function: the policy. A *policy* π is a function that evaluates the current state of the environment, and outputs the action that the agent should take. More formally, $\pi\colon \mathcal{S}\longrightarrow\mathcal{A}$.

The RL agent needs to learn the optimal policy, the policy that, broadly speaking, maximizes some measure of the expected total reward. There are several options to define this measure. For example, we might aim to maximize only the immediate reward R_{t+1}, but this objective would result in a short-sighted greedy policy that prioritizes short term rewards at the cost of long term losses. A more sensible choice would be to aim to maximize the sum of rewards G_t,

$$G_t \coloneqq R_{t+1} + R_{t+2} + R_{t+3} + ...$$

However, G_t still has some inconvenients. Firstly, in the case of infinite episodes, G_t might fail to converge. Furthermore, it does not seem logical to consider that all rewards are equally important, regardless of how far down the road they are. The farther in the future a reward is, the more uncertain it is. Discounting the sum of rewards solves both issues, and gives place to a new *discounted cumulative sum of rewards* G_t

$$G_t \coloneqq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where γ is the *discount factor* and is usually $0.9 < \gamma < 0.999$.

The optimal policy is therefore

$$\pi^* \coloneqq \underset{\pi\in\Pi}{\operatorname{argmax}}\, \mathbb{E}_\pi[G_t|S_t=s] \quad \forall s\in\mathcal{S}$$

3.2 Estimating the optimal policy

Now that the optimal policy π^* has been mathematically defined, an approximation algorithm is needed in order to estimate such policy by interacting with the environment to collect experience. Due to their effectiveness at approximating functions, deep neural networks are a good option for such task. Thus, we define $\pi_\theta \colon \mathcal{S} \longrightarrow \mathcal{A}$ to be a neural network with parameters θ. In this section we explore different DRL techniques that approximate π_θ to the optimal policy π^*.

3.2.1 Q-learning

Q-learning is considered to be the first successful DRL implementation. It was the algorithm that enabled DeepMind to train an agent that successfully learned to play 80 Atari games [10]. Incidentally, this breakthrough lead to Google acquiring the company in 2014.

The main idea of Q-learning is to learn a function that estimates the **value** (expected cumulative reward G_t) of $(state, action)$ pairs. Then, the optimal estimated policy consists simply of selecting the action a that is predicted to have the maximum value at state s.

Formally, we define the value function Q^*_π as

$$\begin{aligned} Q^*_\pi \colon \mathcal{S} \times \mathcal{A} &\longrightarrow \mathbb{R} \\ (s, a) &\mapsto \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \end{aligned}$$

Note that Q^*_π depends on the associated policy π, as this policy defines the behaviour of the agent during the episode, and therefore conditions the future states and rewards that arise under such policy.

It is clear that in order to obtain the exact Q^*_π, an exact understanding of the MDP dynamics (the p function) is required. Q Learning therefore aims to estimate an approximation of the exact Q^*_π function by using a function approximator $Q_\theta \colon \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$, a neural network with parameters θ. This network is commonly referred to as the *Q-Network*.

In order to approximate Q_θ to Q^*, the recursive nature of the Q value function is exploited, using what is known as the Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \pi}[r(s, a) + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a')]$$

The Bellman equation basically states that the value of an action a at state s can be computed by performing action a, observing the new state s', and adding up the obtained reward r and the discounted Q value of the new state s', which under a policy that selects the action that has the maximum Q value, will be $\gamma \max_{a' \in \mathcal{A}} Q^*(s', a')]$.

This equation allows us to iteratively approximate Q^*. Suppose we have collected agent -environment interaction experience. Such experience can consist on (s, a, s', r) sets, containing respectively the preceding state, the action taken, the new state, and the reward obtained. The Q_θ update step consists on:

$$Q_{\theta_{new}}(s, a) \doteq r + \gamma \max_{a' \in \mathcal{A}} Q_{\theta_{old}}(s', a')$$

This theoretical update is however impractical, as neural networks cannot handle such large updates at once and therefore will fail to converge. Instead, in practice updating the θ_{old} update is done using gradient descent, shifting θ towards the direction that minimizes the error between the current $Q_\theta(s, a)$ and the ground truth $r + \gamma \max_{a' \in \mathcal{A}} Q_{\theta_{old}}(s', a')$.

Now, a question arises. How should training data be collected? Clearly, the policy during training defines what the agent does, and therefore will lead to the agent exploring some states more than others. For Q-learning, it is common to use the *ϵ-greedy* π_Q policy, that with probability ϵ selects a random action, and otherwise selects the action with the highest Q value. This ϵ incentivises exploration, motivating the agent to explore a broader state space.

However, a strong point in favor of Q-learning above other DRL algorithms is that it is an *off-policy* method. That is, it can learn from experience that has been collected using any policy, not necessarily the *ϵ-greedy* π_Q policy mentioned above. This is not possible for the following method we tackle.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 3.2: Pseudocode for Q-learning algorithm, reprinted from [10].

Common practices and improvements for Q-learning

So far we have discussed the core algorithm for Q-learning. However, since its release in 2013 some improvements have increased its effectiveness significantly, below we list some of the most important ones:

- **Prioritized Experience Replay.** One can intuitively realise that some experiences are more important for learning than others. Some outcomes *surprise* the agents, in the sense that they yield rewards the agent did not anticipate. Prioritized Experience Learning improves convergence by identifying such actions and using them during the training set either more often, or more heavily.

- **Custom Q-networks.** The Q-network used to estimate the Q value is not necessarily a plain Dense NN. If our data is strongly sequential, one can exploit the effectiveness of Recurrent NNs (such as LSTMs) to capture the temporal dependence in the problem.

- **Double Q-learning.** Studying some corner cases shows that trusting a single Q-network can lead to slow training or even to failing convergence. It can be seen that sometimes it is positive to have more than two Q-networks instead of one. These can be trained separately, alternating weight updates.

- **Advantage Function.** Attempting to learn the Q value of a certain (s, a) pair carries with it two separate complex tasks; estimating how good state s is, and how good action a is at state s, compared to other actions. We can therefore split the task and create two networks, one for estimating the average Q value of a state, and another one for estimating the value deviation for a specific action at that state. This Q value deviation is termed the *advantage function*.
- **Distributional RL.** A more Bayesian and probabilistic approach to Q-learning is to study the Q value as a random variable and to try to infer its underlying probability distribution. For example, one can model the Q value as a normal variable and estimate its mean and standard deviation for each (s, a) pair, allowing for greater uncertainty at some cases by predicting a greater standard deviation. Soft Actor Critic is an example of such approach.

3.2.2 Policy gradients

The ultimate goal of DRL techniques is to estimate a policy that maximizes the expected cumulative reward G_t. Policy gradients therefore aims to estimate the gradient of this objective function G_t with respect to the policy. By estimating such gradient, and performing gradient ascent on the policy network weights, one can optimize for the maximum G_t.

We begin by formally defining the policy network π_θ; a NN with parameters θ. Similar to the Q-network, π_θ receives the current state $s \in \mathcal{S}$. It outputs a vector H of size $\#\mathcal{A}$, denoting the numerical preference for each action. This vector is then normalized (typically using softmax) to obtain a vector of action probabilities. The vector H therefore determines the behaviour of the agent, and a change in H will imply a change in the expected cumulative reward G_t.

Notation. We denote the probability of selecting action a_t at state s_t by $\pi_\theta(a_t|s_t)$.

Formally, we define our objective function as

$$J(\theta) \coloneqq \mathbb{E}_{\pi_\theta}[G_t]$$

The gradient we aim to estimate is

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}[G_t]$$

Developing the expectation term

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}[G_t] = \nabla_\theta \sum_{a \in \mathcal{A}} \pi_\theta(a|s_t) G_t = \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s_t) G_t$$

Multiply and divide by $\pi_\theta(a|s_t)$

$$\nabla_\theta J(\theta) = \sum_{a \in \mathcal{A}} \pi_\theta(a|s_t) \frac{\nabla_\theta \pi_\theta(a|s_t)}{\pi_\theta(a|s_t)} G_t$$

Making use of the property $\partial log x = \dfrac{\partial x}{x}$

$$\nabla_\theta J(\theta) = \sum_{a \in \mathcal{A}} \pi_\theta(a|s_t) \nabla_\theta log \pi_\theta(a|s_t) G_t$$

Rearranging the expression as an expectation

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a|s_t) G_t]$$

This manipulation has been key to place the gradient inside the expectation expression. Now, in order to train the policy network, ideally one would perform the update using the gradient $\nabla_\theta J(\theta)$. However, for the exact computation of such gradient, the exact value of the expectation $\mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a|s_t) G_t]$ is needed, but obtaining the exact value is unfeasible. However, one can roughly estimate the expectation of a random variable using just one sample from that variable. Therefore, using collected experience (s'_t, a', G'_t) the gradient can be approximated as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a|s_t) G_t] \approx \nabla_\theta log \pi_\theta(a'|s'_t) G'_t$$

Note that in order for this approximation to be valid, the experience needs to be collected using the policy π_θ. It is therefore an *on-policy* method, unlike the Q-learning algorithm.

Chapter 4

Simulation

This chapter describes the simulator designed for training the DRL agent. It explains the motion model used for the simulated vehicles, and the domain randomization used to train the agent on a variety of driving scenarios. Then, the control pipeline for non-AI vehicles in the environment is described. Lastly, the interaction between the AI agent and the environment is specified.

4.1 Kinematic Bycicle Model

As the collision avoidance problem challenges are mostly due to unpredictable driving behaviors instead of errors in dynamics modelling, a simple Kinematic Bycicle Model is proposed for vehicle motion simulation.

4.1.1 Simplification hypothesis

The KBM assumes some hypothesis that allow for fast and simple computations of the vehicle motion depending on control actions. Formally:

- **2D motion:** Motion along the vertical axis of the car is considered negligible and only motion on the horizontal plane is considered.
- **2 wheels:** The vehicle model consists of a front and a rear wheel, instead of the 4-wheeled car model.
- **No drifting:** There is no drifting or skidding in the wheel-road contact. This is the strongest simplification of this model and allows to ignore tire forces and car dynamics,

focusing only on kinematic restrictions.

4.1.2 Model description

The continuous-time equations that describe vehicle motion are:

$$\begin{aligned}
\dot{x} &= vcos(\psi + \beta) \\
\dot{y} &= vsin(\psi + \beta) \\
\dot{\psi} &= \frac{v}{l_r} sin(\beta) \\
\dot{v} &= a \\
\beta &= tan^{-1}\left(\frac{l_r}{l_f + l_r} tan\left(\delta_f\right)\right)
\end{aligned}$$

where v and a are the velocity and acceleration of the considered centre of the vehicle, respectively. Spatial and angular variables $x, y, \psi, \beta, \delta_f$ describe the position and configuration of the vehicle and are graphically explained in Figure 4.1. Lengths l_f and l_r are model identification parameters denoting the distance from vehicle center to front and rear axis, respectively.

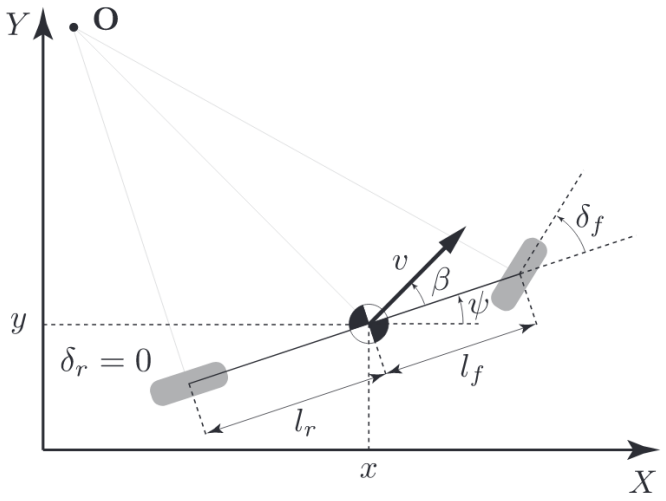


Figure 4.1: The Kinematic Bycicle Model, reprinted from [14]

It is worth noting that the Kinematic Bycicle Model requires just 2 parameters (l_f and l_r) for model identification, which is typically an error prone stage of vehicle simulation.

4.1.3 Control variables and restrictions

Typically, $\mathbf{u} \coloneqq (\delta_f, a)$ denotes the set of control input variables, as they determine vehicle motion.

The steering angle δ_f is limited by the maximum steering angle restriction δ_{max} such that

$$-\delta_{max} < \delta_f < \delta_{max}$$

The increment of δ_f is also limited by δ_{rate} such that

$$-\delta_{rate} < \Delta\delta_f < \delta_{rate}$$

Similarly for a, the Tire Force-Ellipse of the vehicle tires (see [25]) provides an acceptable range of accelerations a_{min}, a_{max} and hence

$$a_{min} < a < a_{max}$$

Excessive increments in acceleration can lead to mechanical damage and discomfort and is therefore limited by a_{rate} such that

$$-a_{rate} < \Delta a < a_{rate}$$

These restrictions are implemented in the simulator, limiting the control actions of the driving agent accordingly.

4.1.4 Simulation step

The state of the vehicle is defined as:

$$\mathbf{x} \coloneqq (x, y, \psi, v)$$

A simulation step consists of calculating the state of the vehicle $\mathbf{x}_{t+\Delta t}$ given the state $\mathbf{x}_t$ and control input $\mathbf{u}_t$. $\mathbf{x}_{t+\Delta t}$ is computed using the Euler Method, which approximates

$$f(\mathbf{x}_{t+\Delta t}) \approx f(x) + f'(x)\Delta t$$

Using the equations described in Section 4.1.2 and the Euler method lead to the following discrete-time model:

$$
\begin{aligned}
x_{t+\Delta t} &\approx x_t + v cos(\psi_t + \beta_t)\Delta t \\
y_{t+\Delta t} &\approx y_t + v sin(\psi_t + \beta_t)\Delta t \\
\psi_{t+\Delta t} &\approx \psi_t + \frac{v_t}{l_r} sin(\beta_t)\Delta t \\
v_{t+\Delta t} &\approx v_t + a_t \Delta t \\
\beta_t &= tan^{-1}\left(\frac{l_r}{l_f + l_r} tan\left(\delta_{f_t}\right)\right)
\end{aligned}
$$

At each time step, the simulator will update the position of each simulated car according to the equations above.

4.2 Simulated surrounding vehicle behaviour

In order to provide challenging traffic interactions to the RL agent, it is necessary to create a control method that directs surrounding cars along thoughout the driving environment. From now on, the term **surrounding vehicle** applies to any simulated vehicle that is not the **ego vehicle**, which is the vehicle controlled by the RL agent.

4.2.1 Behaviour planner

We begin by defining the high-level control. The behaviour planner is in charge of assigning a lane to the surrounding vehicle. The low-level control will then be in charge of changing lanes or following the current lane. The behaviour planner is designed to perform unexpected lane-changes and driving maneauvers, in order to challenge the RLagent collision avoiding capacity.

At each time step, the behaviour planner reevaluates the assigned lane of the controlled surrounding vehicle. Most of the times, the planner will decide to stay on the current lane. However, with relatively low probability ϵ, the vehicle will decide to execute an unexpected maneauver. This can be:

- **Lane-change:** The planner will change the assigned lane to an adjacent lane, after checking that no other surrounding vehicles obstruct such change (note that it does not

take into account the ego vehicle).

- **Speed update:** Updating the desired update, either suddenly increasing, or decreasing it, to test the RL agent capacity.
- **Sudden stop:** Unexpectedly braking and staying stopped until the episode ends.

Such maneauvers will motivate the agent to develop collision avoidance capacity that can be extrapolated to other situations.

4.2.2 Low level control

Executing high level behaviour planner commands requires a low level control that compares the current vehicle state $\mathbf{x}_t$ to the reference and outputs concrete control commands. For such purpose, a line following PID control is considered to be sufficient.

PID steering controller

In order to describe the PID controller, one needs to define the Process Variable (PV), Controlled Variable (CV), and PID parameters:

- **Process Variable:** The PV will be the **distance to lane centerline** of the controlled surrounding vehicle center.
- **Controlled Variable:** The CV will be the **steering angle** $\delta_{\mathbf{f}}$ of the controlled surrounding vehicle.
- **PID parameters:** A **range of values** for each parameter has been studied and has empirically proved to suffice to control the surrounding vehicle successfully. To favour experience randomization, each vehicle is created with a randomly sampled set of PID parameters from such range.

The result is a line following steering controller that robustly performs turns and lane changes.

PI cruise control

Controlling vehicle speed is far more simple and a simple PI control is enough to attain the desired speed at each timestep.

4.3 Road generation and vehicle spawning

A richer range of roads, turns, lane number and turning angles is important to ensure that the agent is exposed to a wide variety of challenges.

Each episode needs its own randomized environment. For each episode, A $\sim$ 200 m road track is created, with a random turn of angle ± 80. The road will have a random number of lanes, from 1 to 4. The number of lanes random variable is more heavily shifted towards 2 lane roads, in order to match frequent real traffic environments.

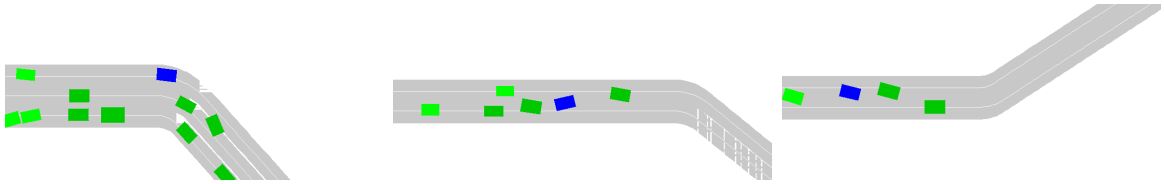


Figure 4.2: Examples of random road generation for simulation.

The ego vehicle is generated at the start of the generated road, in a random lane. Afterwards, the surrounding vehicles are generated, collision free with each other and with the ego vehicle. While the episode lasts new surrounding vehicles are generated in the beginning of the road track to simulate real traffic flow.

4.4 Agent-Environment Interaction

In this section we describe how the DRL agent interacts with the simulation environment. The simulator is designed following the OpenAI Gym standards (see [22]).

4.4.1 Interaction standards

According to the industry DRL standards proposed by OpenAI, the agent-environment interaction can be conducted using an *env* Python objects that conforms to the standards specified in [22]. Such standards allow agile interchange of DRL training algorithms by abstracting the particular characteristics of our environment. More specifically, an example Python implementation of such interaction is detailed below:

```
# reset environment to start a new episode
s = env.reset()
```

```
terminal = False
# start
while not terminal:
    # optional: visualize the simulation
    env.render()
    # Ask the RL Agent to select an action based on state s
    a = RL_Agent. select_action(s)
    # run simulation step executing action a
    s, reward, terminal, info = env.step(a)

env.close()
```

The **step()** method provides the new state **s**, the transition reward **reward**, the Boolean **terminal** that specifies whether the episode has finished, an extra information **info**, which is irrelevant in our case.

4.4.2 State

The state is the representation of the environment that the agent perceives. In our case, the states consists of n laser distance readings, which measure the distance from the ego vehicle to the closest obstacle in each of the n equidistributed directions. An example for 8 lasers is shown in Figure 4.3.

Formally, at time **t**, we can define the environment state $\mathbf{s_t}$ as the n-dimensional vector of distances measured by the n lasers in the ego vehicle. This state $\mathbf{s_t}$ will be the environment representation provided in the **env.step(action)** method described in Section 4.4.

4.4.3 Action

At each timestep, the environment asks the RL agent which action to execute. In our case, this is a control action. As described in Section 4.1.3, this action $\mathbf{u_t}$ comprises the steering angle and motor acceleration to be executed. Formally, $\mathbf{u_t}$ is a 2-dimensional vector containing the desired (δ_f, a).

The simulation environment will restrict the action suggested by the RL agent to the limits in value and change rate described in Section 4.1.3.

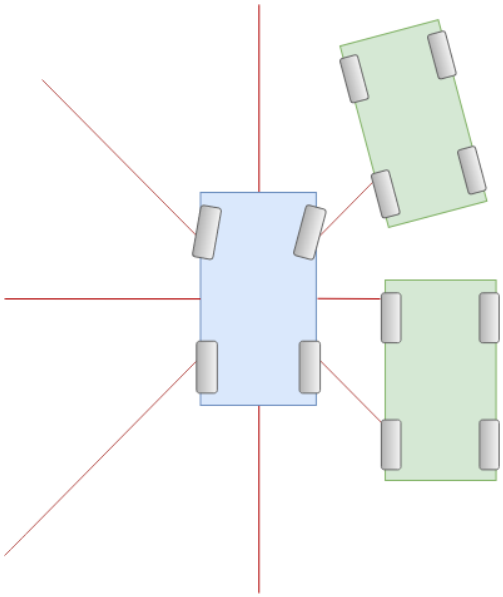


Figure 4.3: The case of 8 lasers ($n = 8$). In red, the laser readings that measure the distance from the ego vehicle (blue) to surrounding vehicles (green).

4.4.4 Reward

The reward provided by the environment guides the RL agent training. Effective reward shaping is critical to align the learning goal of the RL agent with the human task accomplishment metric. For collision avoidance, the main goal is not to crash, and therefore it is straightforward to punish crashes with a strong negative reward.

But this alone is not enough to achieve satisfactory driving behaviour. It is necessary to motivate the agent to advance towards the end of the road segment to complete the navigation task, otherwise, it might find spots on the road where it is relatively safe to stay stopped, avoiding collisions.

As a result of this two simple goals (avoid collisions and complete the navigation task), the proposed reward metric is:

$$r_t = \begin{cases} R_{crash}, & \text{if crashed at time t} \\ R_{navigate}\Delta d, & \text{else} \end{cases}$$

where Δd represents the distance towards the end of the segment that the ego vehicle has advanced in the last time step.

Balancing R_{crash} and $R_{navigate}$ is important to achieve a system that avoids collisions while still completing the navigation task successfully. For our implementation:

$$\begin{cases} R_{crash} = -4 \\ R_{navigate} = 0.01 \end{cases}$$

Chapter 5

RL Implementation

This chapter presents the proposed implementation for the RL agent training. First a DRL training algorithm is selected by evaluating which state-of-the-art algorithm is more suitable for the control task. Then, we describe an overview of the problems that arised during training, and workarounds around such problems.

5.1 Baseline for performance comparison

In Chapter 2, the Conformal Lattice Planner method was described, presented as state-of-the-art for autonomous driving planning. In this section, an implementation of this technique is implemented for our simulator. This will be useful to estimate the performance gain of the proposed RL technique with respect to the current Collision Avoidance techniques in the field.

5.1.1 Problem formulation

Formally, Conformal Lattice Planner (CLP) is composed of several mathematical objects that are outlined below:

1. **Ego-vehicle position $\mathbf{x} \coloneqq (x, y, \psi, v)$** defined in Section 4.1.
2. **Set of candidate goal points $\mathcal{G}$.** These points are considered at different road longitudes and are laterally offset to cover the range of future ego-vehicle positions.
3. **Set of cubic splines $\mathcal{S}$.** A cubic spline is a parameterized curve $s(u) = (x(u), y(u))$

such that

$$x(u) = a_3u^3 + a_2u^2 + a_1u + a_0$$
$$y(u) = b_3u^3 + b_2u^2 + b_1u + b_0$$

This set is associated to $\mathcal{G}$ in the sense that for each goal point $\mathbf{g}$, $\mathcal{S}$ contains a spline $\mathbf{s}$ that defines a smooth trajectory from current ego-vehicle position $\mathbf{x}$ to goal point $\mathbf{g}$. Formally,

$$\forall \mathbf{g} \in \mathcal{G}, \exists \mathbf{s} \in \mathcal{S} \mid \begin{Bmatrix} s(0) = \mathbf{x} \\ s(1) = \mathbf{g} \end{Bmatrix}$$

Each spline is computed by imposing boundary conditions at $u = 0$ and $u = 1$ such that the spline not only matches starting and ending point positions, but also such that the derivative of such spline matches the non-holonomic car motion restriction. Formally,

$$\begin{Bmatrix} s(0) = (x_{ego}, y_{ego}) \\ s'(0) = \vec{v_{ego}} \\ s(1) = g \\ s'(0) = \vec{v_{road}} \end{Bmatrix}$$

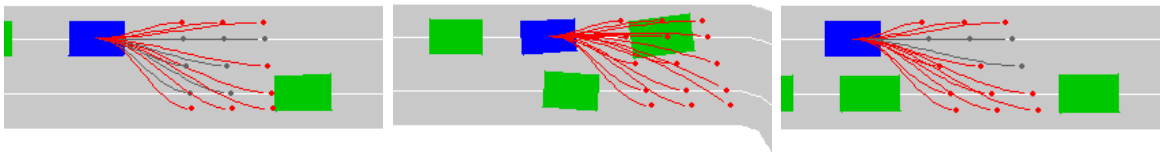
where $\vec{v_{ego}}$ is the ego-vehicle orientation vector and $\vec{v_{road}}(\mathbf{g})$ is the vector parallel to the road at point $\mathbf{g}$.

4. **The objective function $\mathbf{J}(\mathbf{s}) : \mathcal{S} \longrightarrow \mathbb{R}$.** This function aims to evaluate how satisfactory each possible trajectory. Common goals for standard autonomous driving control systems are minimizing collision risk, maximizing passenger comfort, minimizing travelled distance, reassuring stable car motion and dynamics. These can be quantitatively measured by several indicators such as trajectory curvature computation and collision checking. Specifically, the set of quantitative measures considered for the objective functions are:

   - **Collisions in trajectory.** For each trajectory spline, collision checking is essential to discard trajectories that lead to crashing with high probability. For that, trajectories that lie too close to dynamic obstacles or road edges are identified.

This can be seen in Figure 5.1. The collision monitoring variable is a binary variable defined as follows:

$$C(s) = \begin{cases} 1, & \text{if collisions are detected for trajectory } s \\ 0, & \text{else} \end{cases}$$



(a) Left paths too close to road limits.

(b) No collision free trajectory.

(c) Right lane change is ruled out due to the cars on the right.

Figure 5.1: Some examples of trajectory collision checking. In red, trajectories that are too close to obstacles and road limits. In grey, collision-free trajectories.

- **Maximum curvature.** Computing the maximum curvature of each trajectory is time-efficient and is necessary to discard trajectories that require excessive steering. The curvature of a parametric curve can be computed as

$$\kappa(u) = \frac{x'(u)y''(u) - x''(u)y'(u)}{(x'(u)^2 + y'(u)^2)^{\frac{3}{2}}}$$

- **Bending energy.** Bending energy is considered a standard indicator of overall road curvature. Road curvature negatively affects passenger comfort. In physics, bending energy is the amount of flexing energy that needs to be applied to an straight bar to bend it in an specific shape. The bending energy for a parametric curve $s(u)$ can be computed as

$$E(s) = \int_0^1 \kappa(u)^2 du$$

- **Road progress.** In order to minimize traveled distance, it is interesting to evaluate which proportion of the trajectory distance is dedicated to longitudinal motion

along the road. By rewarding trajectories that focus on advancing forward on the road, swerving is penalized and the control yields smoother and more stable trajectories. The *road progress* variable is defined as follows:

$$R_p(s) = \frac{\Delta d}{l(s)}$$

where Δd is the length of the projection of s on the road center line, and $l(s)$ is the length of trajectory s.

The objective function can be therefore calculated as

$$J(s) = w_C C(s) + w_{\kappa_{max}} \kappa_{max}(s) + w_E E(s) + w_{R_p} R_p(s)$$

5.1.2 Implementation

Path planning

At each timestep, the set of goal points $\mathcal{G}$ (in red in Figure 5.2) is computed. Then, splines (in grey in figure 5.2) for each goal point are calculated, imposing the boundary conditions described in the previous section. The optimal trajectory is selected by evaluating the cost function $J(s)$ for each spline trajectory s and choosing the trajectory with minimum cost.

Specific values for $J(s)$ weights are detailed in the following table.

Objective Function Weights				
Description	Collisions in trajectory	Maximum curvature	Bending energy	Road progress
Symbol	w_C	$w_{\kappa_{max}}$	w_E	w_{R_p}
Value	10.0	2.0	2.0	1.0

Control

The relationship between the car trajectory curvature κ and the steering angle δ that generates such curvature can be geometrically obtained and yields the following equality

$$\delta = arctan(\frac{l_f + l_r}{l_r} tan(\kappa))$$

Therefore, given the desired trajectory spline s, one can compute the required δ by computing $\kappa(0)$ and evaluating the resulting steering angle with the equation above.

Regarding motor acceleration, this cannot be directly determined by the shape of the spline s. Instead, a workaround solution would be to determine the acceleration required by considering the length of the optimal spline s. The intuition behind this approach is that longer optimal trajectories allow the agent to increase speed as a long part of the road is obstacle clear. Instead, short optimal trajectories suggest that the longer ones are occupied by obstacles and therefore it is prudent to decrease speed.

$$a = \frac{l(s) - A}{B}$$

where A and B are constants calibrated empirically.

In particular, for cases where none of the considered possible trajectories are collision-free, the acceleration is set to $a = 0$ in order to await for future road clearance.

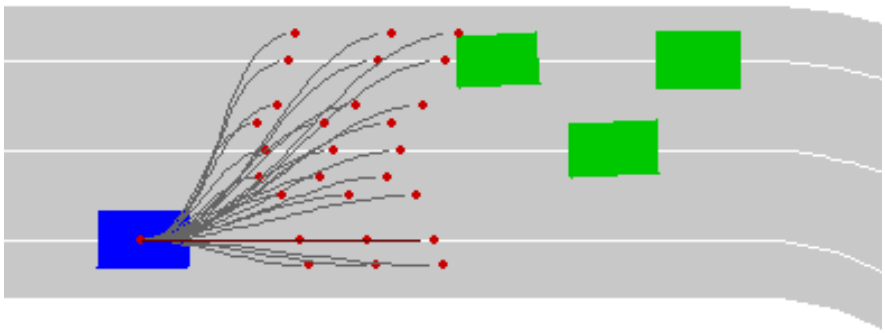


Figure 5.2: CLP: In red, the set of candidate goal points. In grey, the cubic spline trajectories to each goal point.

5.1.3 Baseline Performance

The CLP baseline has been evaluated in the same environment simulator as the RL method, and it is sound to conclude that both methods have faced similarly complex and challenging situations.

Stengths

It can be concluded that the CLP technique works for simple road navigation, especially when no dynamic objects are involved. The method also shows good performance at overtaking slow vehicles if space between cars is sufficient, and overall shows intelligent driving behaviour except for dense traffic situations.

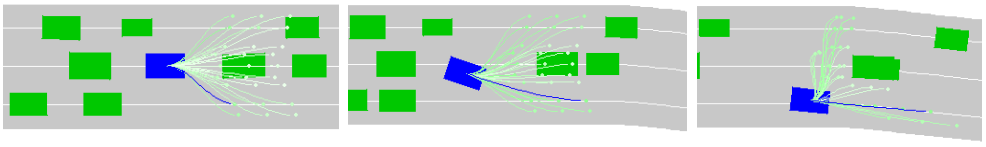


Figure 5.3: Overtaking maneauver performed by CLP control. In blue, the optimal trajectory generated by the planner.

Weaknesses

This implementation shows unstable performance for cases where the ego-vehicle is surrounded by vehicles. It also fails to consider faster vehicles coming from behind.

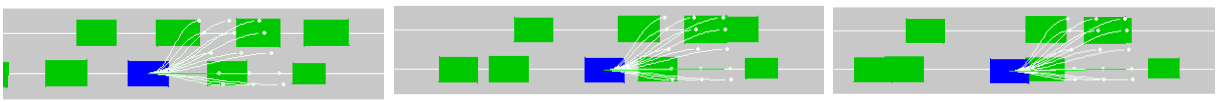
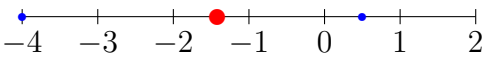


Figure 5.4: Drastic breaking of the front vehicle leads to crash as CLP fails to adapt to speed change.

Performance metric

The metric for baseline performance is the average episode reward, as defined for the RL method. The average episode reward measured by extensive simulation is $R_{CLP} = -1.42$. Episode rewards can range between -4 (worst case scenario) and $+0.5$ (optimal performance).



The CLP method therefore performs roughly in the middle ground between a random policy (that crashes almost immediately) and the ideal policy (never crashing).

5.2 DRL for Collision Avoidance

This section constructs on the previous chapters and analyses the specific solution presented in this thesis to the Collision Avoidance Problem. First, a brief review of the state-of-the-art DRL methods focusing on this specific problem is outlined. Then, the concrete input and output for the DRL algorithm is discussed. Lastly, specific Collision Avoidance RL characteristics are outlined.

5.2.1 Algorithm selection

The field of DRL has seen outstanding progress in recent years. Latest state-of-the-art methods were described in Section 2.2. The main DRL algorithm candidates for the collision avoidance task are therefore SAC and DDPG, as both are designed for continuous action spaces.

5.2.2 System input and output

OpenAI Gym standards [22] for custom DRL task implementations enable to test multiple algorithms interchangeably, without the need of further processing algorithm inputs and outputs. In this scope, we define the input and output formats.

Input: State

In Section 4.4.2, the ego vehicle state was defined as the set of laser readings. This information by itself is very limited and does not comprise temporal and velocity information. The state provided to the DRL algorithm needs to be modified to incorporate this extra information.

Recall that state s_t is n-dimensional, comprising the laser distances in the n equidistributed directions around the ego-vehicle. Some numerical modifications are useful to capture the functional nature of s_t. For instance, a laser reading distance is mostly of interest when it is close to zero, as it indicates that an obstacle is close to the ego vehicle. Therefore, a trick to improve training is to modify the state to obtain a new $\hat{s}_t$ that is more numerically related

to our task

$$\hat{s}_t^i := \frac{5}{0.5 + s_t^i}$$

With this trick, a laser reading of 14.5 meters is transformed to 0.33, whereas a reading of 2 meters is mapped to 2.

Now, in order to incorporate temporal information, distance increments are concatenated to the modified state. The final modified state S_t is $2n + 2$ dimensional, and comprises:

1. $[n]$ Modified state $\hat{s}_t$.
2. [1] Ego-vehicle velocity v_t.
3. $[n]$ Modified state increment $\frac{\hat{s}_t - \hat{s}_{t-1}}{\Delta t}$.
4. [1] Ego-vehicle velocity increment $\frac{v_t - v_{t-1}}{\Delta t}$.

This final S_t is the state representation fed to the DRL algorithm.

Output: Action

DRL algorithms output an action vector (2-dimensional for this case) in the standard $(-1, 1)$ range. The action vector is then scaled to the $a \in [a_{min}, a_{max}]$ and $\delta \in [\delta_{min}, \delta_{max}]$ ranges, and control input rate restrictions are applied if necessary.

5.2.3 Problem Adaptation

It is important to note a few differences between standard RL problems and the Collision Avoidance problem. Understanding these key differences allows for informed hyperparameter tuning and implementation tweaks that can improve training significantly. These key differences are:

1. **Credit assignment.** In collision avoidance, the effect of wrong actions does not take long to lead to crashes. The time gap between action and effect is therefore shorter. This implies that assigning credit to each action is more straightforward than in most other RL problems. Although long term effects of some actions are sometimes important (for example, deciding to change lanes might lead to safer road navigation further down the road), usually wrong actions a few time steps before the crash are the ones that cause the crash.

The implication of this property is that the discount factor γ should be lowered for our problem, as this parameter intends to account for long term action-effect dependence. Whereas in standard DRL training $\gamma = 0.99$, for our case the appropriate value is considered to be lower, specifically $\gamma = 0.92$.

2. **Few action dimensions.** Latest general DRL techniques have focused on high dimensional action spaces. For example, locomotion tasks require controlling many articulations simultaneously, and it is only the correct synchronicity at the set of articulation commands that provides useful locomotion progress data for training. In such cases, exploration of the broad action space is essential, and it is in these environments that soft entropy based algorithms like SAC thrive.

   However, the action space for autonomous driving is only 2-dimensional (steering angle and motor acceleration), and it is likely that exploration of the action space does not need to be artificially favoured. In this case, SAC algorithm's main strength - exploration - might prove inefficient.

3. **Imperfect environment information.** Whereas many of the RL benchmark problems in the field rely on perfect information - that is, the complete set of useful information is accessible to the agent -, in the collision avoidance case the agent relies on a limited low resolution laser reading of its environment. It cannot sense whether each laser reading corresponds to a static obstacle (road limit) or a dynamic one (vehicle). Its readings are also local and fail to obtain a complete map of the driving situation.

5.3 Algorithm implementation and performance

For this work, SAC and DDPG have been implemented according to papers [28] and [29] respectively. Both algorithms were trained on the simulator until convergence. The results are analysed in the pages that follow.

5.3.1 SAC

Pseudocode

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Figure 5.5: SAC pseudocode, reprinted from [28]

Hyperparameters

This subsection presents the set of hyperparameters used for training.

SAC hyperparameter values					
Description	Discount Factor	Polyak constant	Q learning rate	π learning rate	Entropy term
Symbol	γ	ρ	lr_Q	lr_π	α
Value	0.92	0.995	0.001	0.001	0.2

Performance analysis

Figure 5.6 shows how SAC performance evolved during training on the simulated environment detailed in 4 for 100 epochs of 4000 steps each.

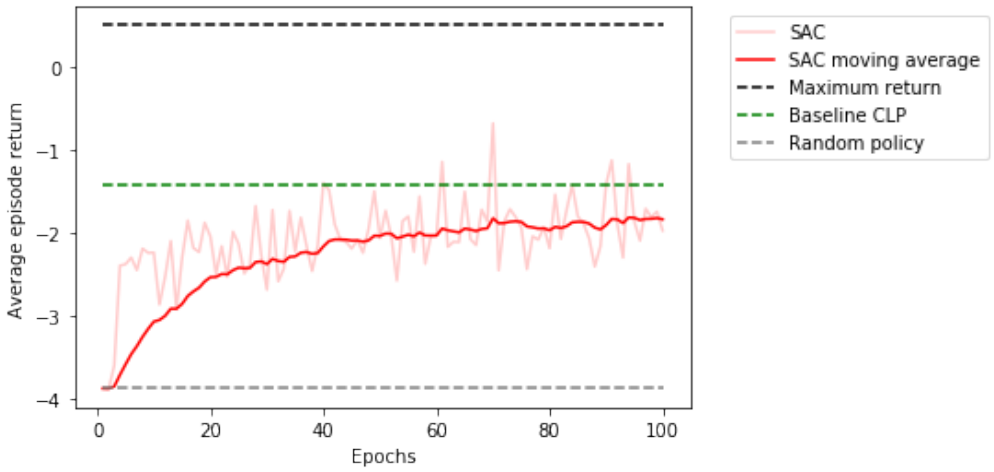


Figure 5.6: SAC performance during training.

It can be seen that SAC performs significantly worse than the CLP baseline. The reason for this might be the one explained in Section 5.2.3. SAC algorithm is designed to broadly explore a more extensive action space. Its strength might not be applicable to the Autonomous Driving task as the action space is only 2-dimensional.

5.3.2 DDPG

Pseudocode

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

Figure 5.7: DDPG pseudocode, reprinted from [29].

Hyperparameters

This subsection presents the set of hyperparameters used for training.

DDPG hyperparameter values					
Description	Discount Factor	Polyak constant	Q learning rate	π learning rate	Action noise std
Symbol	γ	ρ	lr_Q	lr_π	ϵ
Value	0.92	0.995	0.001	0.001	0.1

Performance analysis

Figure 5.8 shows how DDPG performance evolved during training. Starting with a random policy, the agent progressively learns a policy that outperforms the CLP baseline.

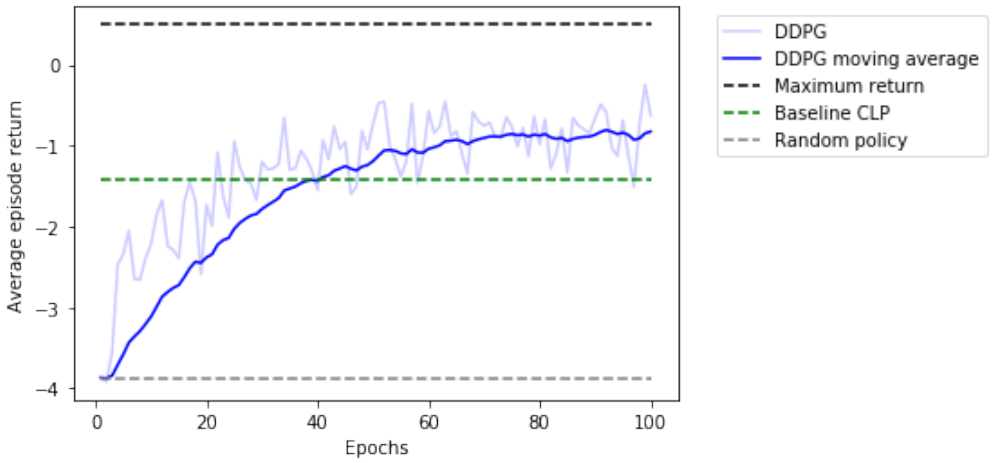


Figure 5.8: DDPG performance during training.

After 100 epochs (roughly 0.4 million steps) the policy performance converges. Evaluating the learned policy for 300 episodes yielded an average episode return of $r = -0.59$, significantly outperforming the CLP average episode return $r = -1.42$.

A more visual analysis of the resultant DDPG control performance can be done by going through several driving situations and analysing how the agent faced each challenge. The following page shows a set of episodes in the simulated environment.

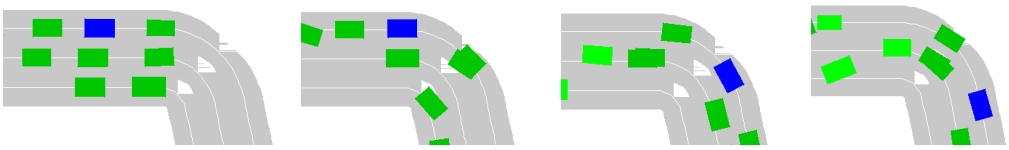


Figure 5.9: The agent performs successfully a sharp turn, in high traffic density. The agent correctly adapts its speed to traffic conditions.

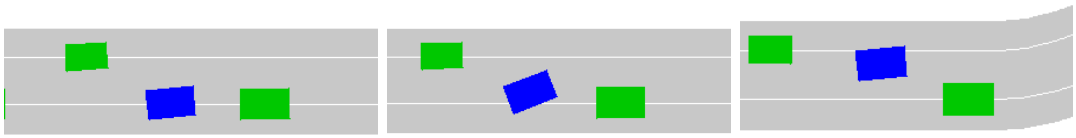


Figure 5.10: Lane change autonomously executed by the DDPG policy.

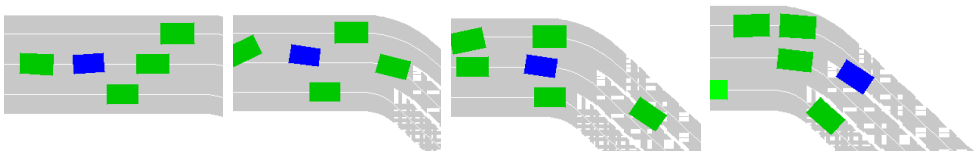


Figure 5.11: The agent identifies an opening in the front and uses it to safely navigate to a less traffic dense area.

Chapter 6

Conclusions and future work

As said in the thesis introduction, it is not the main goal of this thesis to provide a full autonomous driving pipeline. This algorithm could be used instead as an assistant, providing a reference when the main control system identifies sensor failure or uncertainty.

It is worth noting that during training the agent constantly faces driving situations that are far more challenging than regular driving conditions. In simulation, surrounding vehicles are not compliant with the ego vehicle and might invade the ego vehicle's lane unexpectedly. This chaotic environment trains the agent for extreme situations for which the main control pipeline might not be specifically tailored.

Based on the performance presented in Section 5.3, it is safe to conclude that the DDPG algorithm achieves significant collision avoidance capacity. The algorithm shows intelligent driving behaviour. More specifically, the agent performs lane changes and overtakes successfully, aiming for areas with less traffic density which minimize crash risk. Furthermore, the agent steers to the side whenever an adjacent vehicle approaches it laterally, in order to avoid lateral crashes. The algorithm shows ability to adapt its speed to traffic conditions, to avoid front and rear crashes.

However, there is still room for improving the developed technique, mostly in terms of safety. Autonomous Driving requires almost perfect performance, as mistakes can have serious dramatic consequences.

Regarding future work that might spin off this thesis, an interesting possibility might be to implement collaborative driving, where the AI not only controls a single vehicle, but a set of vehicles. This task should give rise to coordination between driving entities, likely reducing

crash risk.

Another technical possibility might be to use a more complex model for vehicle motion, such as a Dynamic Bicycle Model instead of a Kinematic Bicycle Model (KBM). This would make sense as it is in extreme situations that tire forces are strong and vehicle motion is unstable.

On the personal side, the author understands that the research conducted not only aims to explore alternative possibilities in the Autonomous Collision Avoidance field. As a Final Degree Thesis, it is also intended to educate the author, prompting him to work independently and acquire the broad set of skills required for technical endeavors that might arise in his professional future. In that sense, the author has investigated profoundly interesting topics like Deep Learning and Reinforcement Learning, implementing RL techniques for a broad set of tasks. The author has as well implemented a traffic simulator from scratch, comprising vehicle motion models, vehicle control systems, and behaviour planning techniques.

Bibliography

[1] J. S. A. Winkler. Dynamic collision avoidance of industrial cooperating robots using virtual force fields. 2012.

[2] J. L. A. R. T. F. Andy Zeng, Shuran Song. Tossingbot: Learning to throw arbitrary objects with residual physics. 2019.

[3] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch. Emergent tool use from multi-agent autocurricula, 2019.

[4] M. E. S. D. Berger-Tal O, Nathan J. The exploration-exploitation dilemma: A multidisciplinary framework. 2014.

[5] K. Borenstein. The vector field histogram - fast obstacle avoidance for mobile robots. 1991.

[6] N. T. T. K. M. G. B. S. Byungsoo Kim, Vinicius C. Azevedo. Deep fluids: A generative network for parameterized fluid simulations. 2018.

[7] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[8] D. N. N. et al. Virtual force field algorithm for a behaviour-based autonomous robot in unknown environments. 2011.

[9] H. et al. Implementation of autonomous line follower robot. 2012.

[10] M. et al. Playing atari with deep reinforcement learning. 2013.

[11] Z. et al. Reward shaping via meta-learning. 2019.

[12] F. Fan, J. Xiong, and G. Wang. On interpretability of artificial neural networks, 2020.

[13] C. Huang, W. Li, C. Xiao, B. Liang, and S. Han. Potential field method for persistent surveillance of multiple unmanned aerial vehicle sensors. *International Jour-*

nal of Distributed Sensor Networks, 14:155014771875506, 01 2018. doi: 10.1177/1550147718755069.

[14] G. S. F. B. Jason Kong, Mark Pfeiffer. Kinematic and dynamic vehicle models for autonomous driving control design. 2015.

[15] P. D. A. R. O. K. John Schulman, Filip Wolski. Proximal policy optimization algorithms. 2017.

[16] P. M. M. I. J. P. A. John Schulman, Sergey Levine. Trust region policy optimization. 2015.

[17] Jon Liberal. Python Traffic Simulator [Online]. URL here.

[18] O. Khatib. Real-time obstacle avoidance for manipulators ans mobile robots. 1985.

[19] K. E. M. M. Almasri, A. Alajlan. Trajectory planning and collision avoidance algorithm for mobile robotics system. 2016.

[20] M. McNaughton, C. Urmson, J. M. Dolan, and J. Lee. Motion planning for autonomous driving with a conformal spatiotemporal lattice. In *2011 IEEE International Conference on Robotics and Automation*, pages 4889–4895, 2011.

[21] H. Moravec. Certainty grids for sensor fusion in mobile robots. *Sensor Devices and Systems for Robotics*, pages 243–276, January 1989.

[22] OpenAI. OpenAI Gym [Online]. URL here.

[23] S. W. P.J. Antsaklis, K.M. Passino. An introduction to autonomous control systems. 1991.

[24] B. W. Z. K. Po-Wei Wang, Priya L. Donti. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. 2019.

[25] M. B. R. Brach. The tire-force ellipse (friction ellipse) and tire characteristics. 2011.

[26] M. Roser and H. Ritchie. Technological progress. *Our World in Data*, 2020. https://ourworldindata.org/technological-progress.

[27] W. T. V. N. H. L. P. M. e. a. Stafford T, Thirkettle M. A novel task for the investigation of action acquisition. 2012.

[28] P. A. S. L. T. Haarnoja, A. Zhou. Soft actor-critic:off-policy maximum entropy deep reinforcement learning with a stochastic actor. 2018.

[29] A. P. N. H. T. E. Y. T. D. S. D. W. T. P. Lillicrap, J. J. Hunt. Continuous control with deep reinforcement learning. 2016.

[30] I. Ulrich and J. Borenstein. Vfh+: reliable obstacle avoidance for fast mobile robots. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, volume 2, pages 1572–1577 vol.2, 1998.

[31] M. M. A. I. E. V. Balaji, M.Balaji. Optimization of pid control for high speed line tracking robots. 2015.

[32] D. C. D. L. H. H. Xiaohui Li, Zhenping Sun. Development of a new integrated local trajectory planning and tracking control framework for autonomous ground vehicles. 2017.