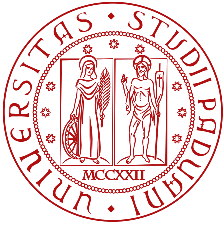
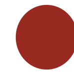


Università degli Studi di Padova



Dipartimento di Ingegneria dell'Informazione

Master Thesis in Control Systems Engineering

Multi-Agent Reinforcement Learning for smart mobility and traffic scenarios

Master Candidate

Matteo Cederle

Student ID 2039829

Supervisor

Prof. Gian Antonio Susto

University of Padova

Co-supervisor

Prof. Mattia Bruschetta

University of Padova

Academic Year 22/23
7 September 2023

Dedicato a:
Anna,
Benedetta,
Gaspare,
Giorgia,
Marta,
Paolo,
Tosca,
Zorro

Abstract

Autonomous Driving is one of the most fascinating and stimulating field in modern engineering. While some partial autonomous cars already exist in the industrial market, they are far from being completely independent in any situation. One of the things that these vehicles lack the most is the ability of handling traffic scenarios and situations in which interaction with other road users is required.

The purpose of this work is that of investigating learning techniques that could be exploited in order to face the challenge described above. The focus will be put on the Multi-Agent Reinforcement Learning (MARL) paradigm, which seems particularly appropriate to address this kind of problem, given its ability to learn and solve complex tasks without any prior knowledge requirement.

The MARL paradigm has its roots both in the classical single agent Reinforcement Learning setup, but also in the Game Theory field. For this reason, after an introduction to the problem and some literature review of related works, the project will begin with an introduction to those two topics. After that the MARL paradigm will be analyzed, focusing both on the theoretical aspects and on the algorithmic point of view.

The thesis will then proceed with an experimental section, where some of the state-of-the-art MARL algorithms will be adapted to the Autonomous Driving setup and tested, making use of a simulator, called SMARTS, specifically developed for this purpose.

To conclude this work the results obtained in simulation will be analyzed and discussed, and also some ideas for future development will be presented.

Sommario

Lo sviluppo di auto a guida autonoma è uno dei settori più affascinanti e stimolanti dell'ingegneria moderna. Anche se alcune auto parzialmente autonome sono già sul mercato, sono ancora distanti dall'essere completamente indipendenti in ogni contesto. Una delle cose che manca di più a questi veicoli è la capacità di gestire scenari di traffico e situazioni dove sono richieste interazioni con altri utenti della strada.

L'obiettivo di questo studio è quello di investigare tecniche di apprendimento automatico che potrebbero essere sfruttate per affrontare la sfida descritta sopra. Il focus verrà messo sul paradigma Multi-Agent Reinforcement Learning (MARL), che sembra particolarmente appropriato per affrontare questo tipo di problema, data la sua capacità di imparare a risolvere problemi complessi senza aver bisogno di alcuna conoscenza preliminare.

Il paradigma MARL ha le sue radici sia nel classico Reinforcement Learning singolo agente, ma anche nel campo della Teoria dei Giochi. Per questo motivo, dopo un'introduzione al problema e l'analisi della letteratura sui lavori correlati, il progetto inizierà con un'introduzione a questi due argomenti. In seguito verrà analizzato il paradigma MARL, focalizzandosi sia sugli aspetti teorici che su quelli implementativi.

La tesi procederà quindi con una sezione sperimentale, dove alcuni sofisticati algoritmi MARL verranno adattati al contesto della guida autonoma e testati, utilizzando un simulatore, chiamato SMARTS, sviluppato specificatamente per questo tipo di situazioni.

Per concludere questo lavoro verranno analizzati e discussi i risultati ottenuti in simulazione, e verrano anche presentate alcune idee per futuri sviluppi.

Contents

List of Figures **xi**

List of Tables **xiii**

List of Algorithms **xv**

1 Introduction **1**
1.1 Related Work 2
1.1.1 Mixed traffic 2
1.1.2 Socially desirable AVs 2
1.1.3 Heterogeneous Human Driven Vehicles (HDVs) 3
1.1.4 Fully-Autonomous Fleet 3
1.2 Our Approach 3

2 Single Agent Reinforcement Learning **5**
2.1 Key elements of RL 6
2.2 An introductory example 8
2.3 Bellman Equations 9
2.4 Dynamic Programming 10
2.5 Temporal Difference Learning 12
2.5.1 SARSA 13
2.5.2 Exploration-Exploitation dilemma 14
2.5.3 Q-Learning 14
2.6 Limits of tabular RL and extensions 15
2.7 Policy Gradient Methods 17

3 Multi-Agent Reinforcement Learning in Games **21**
3.1 Hierarchy of Game Models 21
3.1.1 Matrix Games 22

3.1.2 Repeated Matrix Games 23
3.1.3 Stochastic Games 23
3.1.4 Partially Observable Stochastic Games (POSG) 24
3.2 Solution Concepts for Games 25
3.2.1 Joint Policy and Expected Return 25
3.2.2 Best Response 26
3.2.3 Nash Equilibrium 27
3.2.4 Correlated Equilibrium 27
3.2.5 Conceptual Limitations of Equilibrium Solutions 28
3.2.6 Pareto Optimality 28
3.2.7 Social Welfare and Fairness 28
3.2.8 No Regret 29
3.3 Tabular Multi-Agent Reinforcement Learning 30
3.3.1 General Learning Process 30
3.3.2 Convergence Types 30
3.3.3 Central Learning 31
3.3.4 Independent Learning 31
3.3.5 TD Learning for Games: Joint Action Learning 32
3.3.6 The complexity of computing equilibria 34

4 Deep-MARL Framework and Algorithms 35
4.1 Challenges of Multi-Agent RL 36
4.1.1 Non-stationarity caused by learning agents 36
4.1.2 Multi-Agent Credit Assignment 36
4.1.3 Scaling to many Agents 37
4.2 Classification of MARL algorithms 37
4.2.1 Centralized Training and Execution 38
4.2.2 Decentralised Training and Execution 38
4.2.3 Centralized Training and Decentralized Execution 39
4.3 Independent Deep Q-Networks 40
4.3.1 Independent Double-DQN (IDDQN) 42
4.3.2 Independent Dueling-DQN 43
4.4 Multi-Agent Deep Deterministic Policy Gradient 44
4.5 Value Decomposition Networks & QMIX 47

5 Experimental Setup **51**

5.1 SMARTS simulator . . . 52

5.2 Problem Definition . . . 53

5.2.1 Environment . . . 53

5.2.2 Observation Space . . . 54

5.2.3 Action Space . . . 56

5.2.4 Rewards Structure . . . 56

5.3 Implementation Details of the Algorithms . . . 57

5.3.1 IDQN and Variants . . . 57

5.3.2 MADDPG . . . 58

5.3.3 VDN and QMIX . . . 58

6 Results and Comments **61**

6.1 Results IDQN and Variants . . . 63

6.1.1 Independent Deep Q-Networks . . . 63

6.1.2 Independent Double Deep Q-Networks . . . 65

6.1.3 Independent Dueling Deep Q-Networks . . . 67

6.1.4 Independent Dueling Double Deep Q-Networks . . . 69

6.2 Comments IDQN and Variants . . . 71

6.3 Results MADDPG . . . 72

6.4 Comments MADDPG . . . 73

6.5 Results VDN and QMIX . . . 74

6.5.1 VDN . . . 74

6.5.2 QMIX . . . 75

6.6 Comments VDN and QMIX . . . 76

6.7 Overall Comparison . . . 77

7 Conclusions and Future Works **81**

7.1 Possible future directions . . . 82

References **85**

Acknowledgments **89**

List of Figures

2.1 Basic Reinforcement Learning loop [1] 6
2.2 Example of small grid-world scenario [16] 8

3.1 Hierarchy of game models [1] 22
3.2 Examples of matrix games: (a) is a zero-sum game, (b) a common reward game and (c) a general sum game [1] 23
3.3 Example of Game Models [1] 24
3.4 MARL problem definition [1] 25

4.1 Example of CNN to approximate the action value function [17] . 40
4.2 Comparison between the classical structure of a Deep Q-Network (top) and the Dueling architecture (bottom) [31] 43
4.3 Structure of Multi-Agent centralized critic, decentralized actor [15] 45
4.4 Network architecture of VDN and QMIX [1] 48

5.1 Driving interaction scenarios available in SMARTS [32] 52
5.2 Different combinations of the three-way junction scenario 54
5.3 Example of one of the frames composing the observation of an agent 55

6.1 IDQN 2 AVs scenario 63
6.2 IDQN 2 AVs and 1 HD vehicle scenario 63
6.3 IDQN 3 AVs scenario 64
6.4 IDQN 3 AVs and 1 HD vehicle scenario 64
6.5 IDDQN 2 AVs scenario 65
6.6 IDDQN 2 AVs and 1 HD vehicle scenario 65
6.7 IDDQN 3 AVs scenario 66
6.8 IDDQN 3 AVs and 1 HD vehicle scenario 66
6.9 Independent Dueling DQN 2 AVs scenario 67

6.10 Independent Dueling DQN 2 AVs and 1 HD vehicle scenario . . . 67
6.11 Independent Dueling DQN 3 AVs scenario 68
6.12 Independent Dueling DQN 3 AVs and 1 HD vehicle scenario . . . 68
6.13 Independent Dueling DDQN 2 AVs scenario 69
6.14 Independent Dueling DDQN 2 AVs and 1 HD vehicle scenario . . 69
6.15 Independent Dueling DDQN 3 AVs scenario 70
6.16 Independent Dueling DDQN 3 AVs and 1 HD vehicle scenario . . 70
6.17 MADDPG 2 AVs scenario 72
6.18 MADDPG 2 AVs and 1 HD vehicle scenario 72
6.19 VDN 2 AVs scenario 74
6.20 VDN 2 AVs and 1 HD vehicle scenario 74
6.21 QMIX 2 AVs scenario 75
6.22 QMIX 2 AVs and 1 HD vehicle scenario 75

List of Tables

6.1 2 AVs scenario 77
6.2 2 AVs and 1 HD vehicle scenario 77
6.3 3 AVs scenario 77
6.4 3 AVs and 1 HD vehicle scenario 78

List of Algorithms

1 Policy Iteration 11
2 Value Iteration 12
3 SARSA 13
4 Q-Learning 15
5 Q-Actor-Critic 18
6 Central Q-Learning 31
7 Independent Q-Learning 32
8 Joint Action Learning with Game Theory 34
9 Independent Deep Q-Networks 42
10 Multi-Agent Deep Deterministic Policy Gradient 46
11 VDN and QMIX 49

Introduction

The rapidly growing interest and research in the Autonomous Driving field raises the fundamental problem of the cohabitation between Autonomous Vehicles (AVs) and human drivers. The challenges that need to be addressed are multiple, but for sure the most crucial aspect regards the safety of the road users.

With effective algorithms that prevent fatal accidents we must say that the best way to ensure safety would probably be that of banning the human drivers from the road and keep just AVs. Indeed, according to the World Health Organization (WHO), road accidents kill 1.3 million people and injure 50 million people each year [6]. However this scenario is still far away in the future, and before reaching that we need to design robust strategies for mixed traffic, where Autonomous Vehicles interact with human drivers. This task is particularly complex, because while the behaviour of AVs is homogeneous and predictable, that of human drivers is extremely heterogeneous, unpredictable and many times also irrational. This, added to the fact that the possible driving scenarios that an AV could encounter are nearly endless, makes the design of rule-based models almost certain to fail.

Machine Learning approaches instead give the opportunity to generalize driving scenarios and learn to adapt the AV behaviour in different situations. The (Multi-Agent) Reinforcement Learning (MARL) paradigm in particular seems particularly suited to face these kind of problems, given its ability to learn and solve complex sequential decision problems without any prior knowledge requirement.

1.1 Related Work

Throughout the MARL for Autonomous Vehicles literature there are four different research paradigms at the moment [6], and in the following subsections we will briefly analyze them.

We also note that at this time the majority of the works considers small scenarios, which can be divided into three categories, with increasing complexity:

1. Highway driving
2. Merging and exiting
3. Intersections and roundabouts

Given the complexity of the last category of scenarios, most research concentrate on the first two levels.

Moreover according to [6], the large majority of the works implements independent learning techniques and considers a small number of agents in the simulation (≤ 5). This choice is due to the MARL challenges, which will be analyzed in depth in Section 4.1.

1.1.1 Mixed traffic

In this scenario we consider Autonomous Vehicles cohabiting with human drivers in mixed traffic. Each AV will have its personal goal to achieve, and it will need to learn how to interact with the other road users in order to do that.

The most studied environments in this setup are the highway and the exiting [7], [10]. A downside of these works is that the Autonomous Vehicle learning process is significantly affected by the absence of other AVs in its proximity, since communication between agents is a key aspect of these strategies.

1.1.2 Socially desirable AVs

The researchers working on this paradigm tried to incorporate the AVs with social abilities. Socially desirable AVs will likely include the concept of altruism [6]. In three consecutive papers, Toghi et al. [28], [27], [26] tackled the merging and exiting scenarios with socially desirable AVs.

The results of these works established that socially desirable AVs can improve the success rate of merging and exiting maneuvers [6]. However we must note

that all the human drivers present in the simulation were controlled by the same model, and thus their behaviour was easily predictable.

1.1.3 Heterogeneous Human Driven Vehicles (HDVs)

Some works have focused on designing sophisticated HDV models, because having heterogeneous HDVs would be of great benefit for the learning process of Autonomous Vehicles. However this is still an open challenge, since existing HDV models are unrealistic because they lack some fundamental human peculiarities, such as psychological and biological traits. Indeed introducing AVs trained with these HDV models into real-world traffic would likely result in accidents. [6]

1.1.4 Fully-Autonomous Fleet

This last paradigm considers the situation in which there are no human drivers on the road, but just Autonomous Vehicles, simulating what could be a possible scenario in the future, as already discussed at the beginning. Clearly coordinating a fully-autonomous fleet is more tractable than driving in mixed traffic, given the predictable nature of homogeneous agents. [6]

Also in this case the majority of the works focuses on highway driving and merging environments. Differently from before, in this case the control of all the agents can be cast as a fully cooperative problem.

1.2 Our Approach

In this work we will focus on the mixed traffic paradigm, because we believe that in the next future it will be crucial to have safe and robust cohabitation between Autonomous Vehicles and human drivers.

Differently from the majority of the works present in literature, we will simulate an intersection environment, designing different scenarios of increasing complexity, to analyze the differences in the learning process depending on the number of agents present in the simulation.

We will implement and test four[1] different state-of-the-art MARL algorithms,

[1]One of them will be in turn tested in four different variants.

adapting them to our specific situation. These algorithms have been chosen because they represent the three most important categories of MARL algorithms, and for this reason the comparison between them will be an important aspect of the experimental results analysis.

We will also design realistic observation spaces for each agent, trying to consider just information that in the real-world could be obtained through sensors. Finally, to try and overcome the issue of Section 1.1.1, we will not allow direct communication between the AVs.

To conclude this introduction, we give a brief outline of this project. We will start by introducing the single agent Reinforcement Learning paradigm in Chapter 2, while in Chapter 3 we will discuss some concepts taken from Game Theory, which are useful to describe Multi-Agent interactions. After that, in Chapter 4 we will deeply analyze the Multi-Agent Reinforcement Learning field, along with the four state-of-the-art algorithms mentioned above.

Finally, in Chapters 5 and 6 we will describe our experimental setup and analyze the results. To conclude the thesis, in Chapter 7 we will summarize the work done and present some possible directions for future studies.

Single Agent Reinforcement Learning

What is Reinforcement Learning? It essentially is an approach for mimicking the way in which humans, and more in general animals, learn to solve a problem by trial and error. More formally:

> *Reinforcement Learning (RL) algorithms learn solutions for sequential decision processes via repeated interaction with an environment. [1]*

There are lots of complex and interesting problems that can be formalized in a RL setup:

- Autonomous agents (self driving cars[1], robots, ...)
- Games
- HVAC (Heating, Ventilating, Air Conditioning) energy optimization
- Trading and Portfolio management
- Online advertising and Recommendation systems
- ...

In this chapter we will give a brief introduction to the single agent Reinforcement Learning paradigm, in order to introduce and explain concepts that will be useful when switching to the Multi-Agent setup.

[1]The focus of this thesis.

2.1 Key elements of RL

To understand better the definition of Reinforcement Learning let us first introduce the core elements of a RL problem:

- Agent: the entity aiming at solving a task
- Environment: the world in which the agent lives
- Task: the objective of the agent

To learn how to solve the task the agent repeatedly interacts with the environment, playing many "episodes". Each episode is composed of a sequence of time steps, and each time step consists of the following (see Figure 2.1):

1. The agent is in a state s (a certain configuration of the environment)
2. It performs an action a, chosen from a set of available actions for the agent, called action space
3. Given the current state and action, the environment provides a reward r to the agent and moves it to a new state s', according to a state probability distribution $P(s'|s,a)$

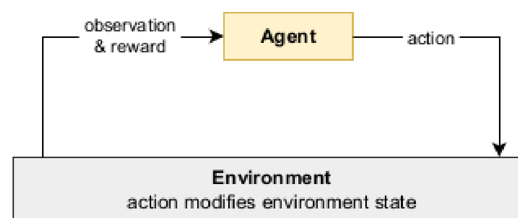


Figure 2.1: Basic Reinforcement Learning loop [1]

The formalization of this setup is given by the concept of Markov Decision Process (MDP):

A Markov Decision Process is a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$ *such that:*

- $\mathcal{S}$ *is a finite set of states*
- $\mathcal{A}$ *is a finite set of actions*
- $\mathcal{P}$ *is a state transition probability matrix with entries:*
$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$
- $\mathcal{R}$ *is a reward function,* $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$
- γ *is a discount factor,* $\gamma \in [0,1]$

The rewards are what make the agent learn, because a "good" action corresponds to a high reward, while a "bad" action corresponds to a low one. Since in RL we are not interested in maximising the value of a single step, but we want to maximise the sum of cumulative rewards, we introduce the following quantity:

The return G_t is the total discounted reward from time step t

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1}$$

the more γ is close to 0 the more we care only about present rewards. The more γ is close to 1 the more we care about all rewards, even those far away in the future.

Up to now we have not said anything about the way in which the agent decides which action to perform at each time step. To do that we have to introduce the concept of policy:

A policy π is a distribution over actions given that we are in a state:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{2.2}$$

The agent follows a policy during each episode, and that completely defines its behaviour.

Note that an MDP policy is stationary (it does not depend on t), but it can be changed in future episodes. We also consider stochastic policies.

The last elements that we introduce in this section are the state value function $V(s)$, and the action value function $Q(s, a)$, defined in this way:

The state value function of a Markov Decision Process is the expected return from state s if we follow policy π:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.3}$$

The action value function of a Markov Decision Process is the expected return from state s if we take action a and then follow policy π:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.4}$$

2.2 An introductory example

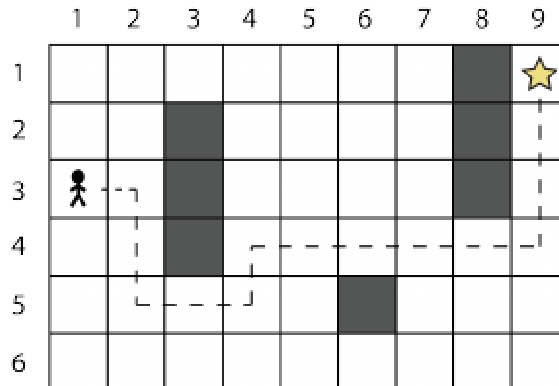


Figure 2.2: Example of small grid-world scenario [16]

Consider the simple example in Figure 2.2. Even if this is a simple scenario, it will be useful to introduce and better understand all the elements introduced in the previous section.

Here the agent is the little person living in that 6 x 9 grid-world environment, and its task consists in reaching the star in the least possible number of steps. The state of the agent is the tuple containing its actual coordinates ((3,1) in the starting position shown in the figure), and at each time step it can move by performing an action sampled from the action space $\mathcal{A} = \{\text{up, down, left, right}\}$. The gray squares correspond to positions where the agent cannot go, and for simplicity we assume that if an action would move the agent outside of the grid, it just keeps it where it is. Of course the state space $\mathcal{S}$ consists of all the possible positions in which the agent can be.

The agent receives a reward of -1 for each time step, and a reward of 0 for reaching the star. In this way it has to maximise its expected return to reach the star in the least possible number of steps.

Each episode begins in the configuration shown in the figure an terminates when the agent reaches the star.

As said in the previous section, the agent decides which action to take at each time step by following a policy. The simplest possible policy is the random one, where each action has equal probability of being taken at each time step. In this case:

$$\pi(a|s) = 0.25 \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{2.5}$$

Clearly this is not a smart policy and it will not maximise the agent's expected return. The next step will be to understand how to modify the policy in order to achieve the agent's goal, and we will discuss that starting from the next section.

2.3 Bellman Equations

Up to now we have just introduced the setup of standard Reinforcement Learning and its key concepts, but we are still missing the fundamental aspect of the problem: How can the agent learn?

To answer this question we first need to introduce a set of recursive equations, called Bellman Equations. We start by considering again the expression of the value function:

$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$

$\ldots \qquad = \mathbb{E}_\pi[R_t + \gamma G_{t+1}|S_t = s]$

$\ldots \qquad = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} P(s'|s,a)[R_s^a + \gamma\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']]$

$\ldots \qquad = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} P(s'|s,a)[R_s^a + \gamma V_\pi(s')]$

Following the same procedure we can find a similar relation for the action value function:

$$Q_\pi(s,a) = R_s^a + \gamma \sum_{s'\in\mathcal{S}} P_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a'|s')Q_\pi(s',a') \tag{2.6}$$

The Bellman Equations just found are crucial for the RL setup because they allow us to put the value of a state (or state-action couple) in relation with the next one. We will see that this concept will be used in every Reinforcement Learning algorithm. Our final objective is to find a policy π_* such that the state value function and the action value function are maximised. To do that we need to find:

$$V_*(s) = \max_\pi V_\pi(s) \tag{2.7}$$

$$Q_*(s,a) = \max_\pi Q_\pi(s,a)$$

An MDP can be considered solved on a RL perspective when the optimal action value function is known: we always know the optimal action to take from a certain state.
Indeed if we know $Q_*(s,a)$ we can easily recover the optimal policy:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \text{argmax}_{a \in \mathcal{A}} Q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

At this point the last thing we need is a way to calculate the optimal state value function and action value function. Following a reasoning similar to before we can find the so-called Bellman Optimality Equations:

$$V_*(s) = \max_a [R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_*(s')] \tag{2.9}$$

$$Q_*(s,a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a'} Q_*(s',a') \tag{2.10}$$

Since there is a non-linear operator in the expression we cannot find a closed form solution, but we will resort to iterative methods instead. The first and most simple one is Dynamic Programming, which will be discussed in the next paragraph.

2.4 Dynamic Programming

In this section we will introduce the first collection of algorithms that can be used to compute optimal policies, given a perfect model of the environment as a Markov Decision Process. [24]

Indeed in this section we will make three strong assumptions, that later on will be relaxed in order to be able to face real and complex problems:

- The state and action spaces are discrete and finite
- The state space is sufficiently small
- Both the state transition probability and the reward function are known

Dynamic Programming (DP) algorithms use the Bellman equations as operators to iteratively estimate the value function and optimal policy. [1].

The basic DP approach is called Policy Iteration, which alternates between two tasks: [1]

- Policy evaluation: compute value function V_π for current policy π
- Policy improvement: improve current policy π with respect to V_π

We could prove that under the three assumptions described above, if we iteratively repeat this two steps, then π and V_π will converge to the optimal policy π_* and the optimal value function V_π^*.

Algorithm 1 Policy Iteration

Initialize: $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$; $V(\text{terminal}) \doteq 0$

1. Policy Evaluation
$\Delta \leftarrow 0$
while $\Delta < \theta$ (small positive number) **do**
  for all $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
end while

2. Policy Improvement
$policy_stable \leftarrow true$
for all $s \in \mathcal{S}$ **do**
  $old_action \leftarrow \pi(s)$
  $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
  if $old_action \neq \pi(s)$ **then**
    $policy_stable \leftarrow false$
  end if
end for
if $policy_stable$ **then**
  return $V = V_*$ and $\pi = \pi_*$
else
  go to 1.
end if

Since the policy evaluation step can be very expensive in terms of computational cost, we would like a way to truncate this step without losing the convergence guarantees. There are many ways to do that, but the simplest approach is the Value Iteration algorithm, where we use the Bellman Optimality Equation to combine policy evaluation and update in a single step.

Algorithm 2 Value Iteration

Initialize: $V(s) \in \mathbb{R}$ for all $s \in \mathcal{S}$; $V(\text{terminal}) \doteq 0$

$\Delta \leftarrow 0$
while $\Delta < \theta$ (small positive number) **do**
  for all $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
end while

Output a deterministic policy $\pi \approx \pi_*$ such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

2.5 Temporal Difference Learning

Up to now we have considered the simplest case of RL problem, because we had the very strong assumption that both the state transition probability and the reward function were known. Starting from that we used the Dynamic Programming paradigm to iteratively compute and update $V(s)$ and $\pi(s)$. Note that we did not need to interact with the environment because we had perfect knowledge of it.

Removing that assumption, we need to interact and gather experience from the environment in order to learn the state transition probability and the reward function. That is exactly what the Temporal Difference Learning paradigm does: it uses data collected from the environment and bootstrapping to learn action value functions. The general update rule is the following: [1]

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\mathcal{X} - Q(s_t, a_t)] \tag{2.11}$$

where $\mathcal{X}$ is the update target, and $\alpha \in (0, 1]$ is the learning rate (or step size). Many different quantities can be used as target, in this section we present two basic variants that are widely used and interesting to introduce a couple of crucial concepts in Reinforcement Learning.

2.5.1 SARSA

For this algorithm we use as update target $\mathcal{X}$ the quantity $r_t + \gamma Q(s_{t+1}, a_{t+1})$. This expression is an approximation of the Bellman Equation 2.6, but since we do not have complete knowledge of the environment, we use the quintuple of elements $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, that make up a transition from one state-action pair to the next [24]. Note that this quintuple is what gives the name SARSA to the algorithm.

Let us first introduce the algorithm, and then make all the necessary comments about its structure and concepts.

Algorithm 3 SARSA

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize: $Q(s, a) \in \mathbb{R}$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$; $Q(\text{terminal}, \cdot) \doteq 0$

for all episodes **do**
  Initialize s
  Choose a from s using policy derived from Q (e.g. ϵ-greedy)
  for all steps of the episode **do**
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g. ϵ-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'; a \leftarrow a'$
  end for(when s is terminal)
end for

Here we can see for the first time the standard structure of a RL algorithm: two nested loops, the first one sweeping through all the episodes we want to play, and the second one iterating over each time step of every episode.

Another crucial aspect that we can see here is the type of policy that we choose for the agent. The ϵ-greedy policy is probably the most used one in Reinforcement Learning, and it is defined like this:

$$\pi(s) = \begin{cases} \text{argmax}_a Q(s, a) & \text{with probability } 1 - \epsilon \\ \text{random action } a & \text{with probability } \epsilon \end{cases} \tag{2.12}$$

This policy is widely used because it can balance well Exploration and Exploitation during the algorithm. The Exploration-Exploitation dilemma is a fundamental concept of RL, and it will be presented in the next paragraph.

2.5.2 Exploration-Exploitation dilemma

When running a RL algorithm we need to explore the environment in order to find the best possible solution to the problem and to better approximate the state transition probability and reward function. However, we also want to achieve good results throughout the learning process, exploiting the current knowledge of the environment. This duality gives rise to the Exploration-Exploitation dilemma:

- Exploration allows us to improve knowledge for long-term benefit
- Exploitation uses current knowledge for short-term benefit

Clearly we cannot do both at the same time: implementing trade-offs is necessary.

There are many different techniques to implement a good trade-off, and the ϵ-greedy policy is one of the most used and effective. The one presented in the SARSA algorithm is the basic version, but more complicated approaches could also be used. For example ϵ could be variable, higher at the beginning of the algorithm, and slowly decreasing while the episodes proceed. This is intuitive because at the beginning we know nothing about the environment and we need to explore more. On the contrary when we have gathered a good knowledge about it we can exploit this knowledge to achieve better results.

2.5.3 Q-Learning

This algorithm is very similar to SARSA, but it uses a different update target, $\mathcal{X} = r_t + \gamma \max_{a' \in \mathcal{A}} Q(s, a')$. Note that now we are approximating the Bellman Optimality Equation 2.10. This difference makes the Q-Learning algorithm faster than SARSA, because it does not iterate between policy evaluation and improvement, but it learns the optimal values directly.

Comparing SARSA and Q-Learning we can introduce an important categorization for Reinforcement Learning algorithms, between on-policy and off-policy techniques.

On-policy algorithms learn about policy π from experience sampled from policy π itself. Off-policy algorithms on the contrary learn about policy π from experience sampled from another policy b (behaviour policy).

This is exactly the case of SARSA and Q-Learning: in the update target of SARSA, the following action is sampled from the current policy ($a_{t+1} \sim \pi(s)$), while in Q-Learning it is sampled from a different policy ($a_{t+1} = \text{argmax}_{a'} q(s', a')$).

Algorithm 4 Q-Learning

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize: $Q(s, a) \in \mathbb{R}$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$; $Q(\text{terminal}, \cdot) \doteq 0$

for all episodes **do**
  Initialize s
  for all steps of the episode **do**
    Choose a from s using policy derived from Q (e.g. ϵ-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  end for(when s is terminal)
end for

2.6 Limits of tabular RL and extensions

Recall the three simplifying assumptions made at the beginning of Section 2.4:

- The state and action spaces are discrete and finite
- The state space is sufficiently small
- Both the state transition probability and the reward function are known

The third one has already been removed when we moved from Dynamic Programming to Temporal Difference Learning. In order to face large and complex problems we need to remove also the other two.

Up to now we have considered "tabular methods", in the sense that state value functions (or action value functions) could have been represented with large tables, where each entry corresponded to the value of a particular state (or state-action couple).[2]

We immediately see that if we remove the first and second assumptions this approach is no longer feasible, both because there could be an infinite possible number of state-action pairs (if the joint space is infinite or continuous), and because even if it is still discrete and finite, it could give rise to an extremely big table, that cannot fit into the memory of a computer. As a simple example, the small grid-world scenario presented in Section 2.2 has a joint state-action space

[2]From now on we will only refer to the action value function, but the same reasoning applies also for the state value function.

of dimension 185, which is small and tractable, but if, for example, we would like to have an agent which learns how to play chess, we would have a state space of dimension $\approx 10^{50}$, which is enormous even for a computer.

For this reason we need to find a way to approximate the action value function. Among the many different possible ways to do it, the simplest one is the linear function:

$$Q(s,a) \approx \hat{Q}(s,a;\theta) = x(s,a)^T\theta \tag{2.13}$$

where $x(s,a)$ is a feature vector and θ is the vector of parameters to be optimized. Clearly this approximation is quite rough and it does not work well for complex problems.

To address this issue the best function approximators are Neural Networks, that exploit their property of being universal function approximators in order to produce good estimates of the action value function.

To learn the approximated Q-function a gradient based optimization is performed, trying to move the parameter vector θ in the direction that minimises an objective function $J(\theta)$. Many different choices for $J(\theta)$ can be made, but the simplest and most used one is the Mean Squared Error (MSE) loss:

$$J(\theta) = \mathbb{E}_\pi[(Q(s,a) - \hat{Q}(s,a,\theta))^2] \tag{2.14}$$

As already said θ is modified at each time step in order to find a minimum of the objective function:

$$\theta_{t+1} = \theta_t + \Delta\theta, \qquad \Delta\theta = -\frac{1}{2}\alpha\nabla_\theta J(\theta) \tag{2.15}$$

where α is the step size.

The biggest issue at this point is that we do not know the true $Q(s,a)$ and we need to find a suitable approximation for it.

One of the first but yet most effective approaches consists in using as approximation for $Q(s,a)$ the update target of the Q-Learning algorithm:
$r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a', \theta)$. The algorithm exploiting this technique is called DQN (Deep Q-Network) [17], and it probably is the most famous Deep Reinforcement Learning algorithm. Its concepts are not so different to the ones presented here, but it includes a couple of smart ideas that are crucial to solve two problems introduced by the approximation of the action value function with a

Neural Network: [1]

- **Moving target problem:** The target estimates are computed with bootstrapped value estimates of the next state which change as the value function is trained. This challenge, already present in tabular RL, is further exacerbated in Deep-RL, because updating a particular state s generalizes so that the estimated values of many other states are changed as well.
- **Breaking correlations:** In many paradigms of machine learning, it is generally assumed that the data used to train the function approximation are i.i.d. (independent from each other and identically distributed). This assumption is typically violated in Reinforcement Learning, due to the formalism of an environment as an MDP.

The details of how this problems are solved and the pseudocode are left to Section 4.3, where also the Multi-Agent extension of this algorithm will be presented.

2.7 Policy Gradient Methods

So far all the treated methods have been based on action value functions; they learned the values of actions and then selected them based on those estimates action values. In this section instead we consider methods that learn a parameterized policy that can select actions without consulting a value function. [24]

We write $\pi(a|s;\phi) = P\{A_t = a|S_t = s;\phi\}$ to indicate the probability of taking action a at time t given that we are in state s with policy parameters ϕ. Clearly the possible approximations for the policy are the same that we used before for the action value function. Also in this case Neural Networks are the most used functions.

Pros and Cons of Policy-Based RL:

- Pros:
  1. Better convergence properties
  2. Effective in high-dimensional or continuous action spaces
  3. Can learn stochastic policies

- Cons:
  1. Typically converge to a local rather than global optimum
  2. Evaluating a policy is typically inefficient and high variance

We want to learn the policy parameters based on the gradient of some scalar performance measure $J(\phi)$ with respect to the policy. These methods seek to maximise performance, so their updates approximate gradient ascent in $J(\phi)$: [24]

$$\phi_{t+1} = \phi_t + \alpha \nabla \widehat{J(\phi_t)} \tag{2.16}$$

If we use as performance measure the true value function for π_ϕ starting from the initial state of each episode s_0 (i.e. $J(\phi_t) \doteq v_{\pi_\phi}(s_0)$), we have a fundamental theoretical result, called Policy Gradient Theorem, which gives an operational expression for $\nabla \widehat{J(\phi_t)}$:

$$\nabla J(\phi) \propto \mathbb{E}_\pi[Q_\pi(S_t, A_t)\, \nabla \log\left(\pi(A_t|S_t; \phi)\right)] \tag{2.17}$$

Clearly the proportionality constant is not a problem because it can be incorporated in the step-size α of Equation 2.16.

Many different algorithms can be designed exploiting this setup, depending on the choice that we make for approximating the true action value function $Q_\pi(s, a)$. A widely adopted strategy is the so-called Actor Critic method, in which we parameterize the action value function, $Q(s, a) \approx \hat{Q}(s, a, \theta)$. Note that Actor Critic algorithms maintain two sets of parameters:

- **Actor:** Updates policy parameters ϕ, in direction suggested by the critic
- **Critic:** Updates action value function parameters θ

Algorithm 5 Q-Actor-Critic

Algorithm parameters: step sizes $\alpha, \beta \in (0, 1]$

for all episodes **do**
  Initialize s
  Sample $a \sim \pi(a|s; \phi)$
  for all steps of the episode **do**
    Take action a, observe r, s'
    Sample $a' \sim \pi(a'|s'; \phi)$
    $\delta = r + \gamma \hat{Q}_\pi(s', a'; \theta) - \hat{Q}_\pi(s, a; \theta)$
    $\phi = \phi + \alpha\, \nabla \log\left(\pi(a|s; \phi)\right) \hat{Q}_\pi(s, a; \theta)$
    $\theta = \theta - \beta\delta\, \nabla\, \hat{Q}_\pi(s, a; \theta)$
    $s \leftarrow s'; a \leftarrow a'$
  end for(when s is terminal)
end for

A great number of different Actor Critic algorithms exist in literature, and here we have presented the pseudocode of a simple example, just to give a general idea of how this technique works. More complicated strategy will be analyzed in Chapter 4, when we will switch to the Multi-Agent setup.

3

Multi-Agent Reinforcement Learning in Games

In this second chapter we will introduce the Game Theory concepts that are fundamental for the Multi-Agent Reinforcement Learning paradigm. We will begin by defining the different classes of game models, which allow us to describe the setup for a MARL problem. Then we will introduce the solution concepts that can be chosen in order to evaluate a game and decide when it can be considered solved.

Finally we will merge the first two topics in order to define a Multi-Agent Reinforcement Learning problem. Some tabular MARL algorithms will also be presented, to prepare the ground for the Deep-MARL framework that we will introduce in the next chapter, but also to show that, like in single agent RL, tabular strategies are unfeasible to solve complex problems, and resorting to Deep Learning techniques is the natural choice to face this kind of situations.

3.1 Hierarchy of Game Models

The Markov Decision Process (MDP) setup introduced in the previous chapter is a particular case of model of multi-agent interaction. Since these models are rooted in Game Theory, they are called games. In Figure 3.1 we show a hierarchy of increasingly complex game models.

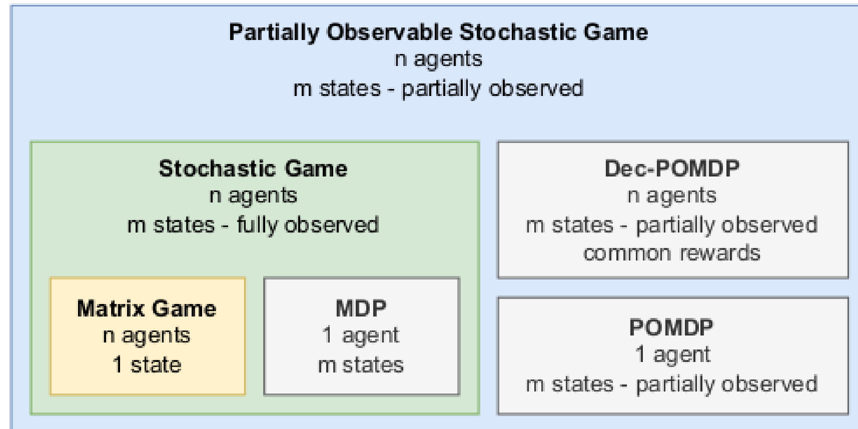


Figure 3.1: Hierarchy of game models [1]

3.1.1 Matrix Games

They represent a single interaction between two or more agents.

- Finite set of agents $\mathcal{I} = \{1, \ldots, n\}$
- For each agent $i \in \mathcal{I}$:
  - Finite set of actions $\mathcal{A}_i$
  - Reward function $R_i : \mathcal{A} \rightarrow \mathbb{R}$, where $\mathcal{A} = \prod_{i=1}^{n} \mathcal{A}_i$

Structure of the game:

- Each agent i selects a policy $\pi_i : \mathcal{A}_i \rightarrow [0, 1]$
- Each agent samples an action $a_i \in \mathcal{A}_i$ with probability $\pi(a_i)$. $\implies$ joint action $a = (a_1, \ldots, a_n)$
- Each agent receives a reward $r_i = R_i(a)$

Matrix games are classified based on the relationship between the reward functions of the agents. This classification is very important and it will be mentioned many times throughout this project:

- Zero-sum game (competitive): $\sum_{i \in \mathcal{I}} R_i(a) = 0 \quad \forall a \in \mathcal{A}$
- Common reward game (cooperative): $R_i = R_j \quad \forall i, j$
- General-sum game: There are no restrictions

	R	P	S
R	0,0	-1,1	1,-1
P	1,-1	0,0	-1,1
S	-1,1	1,-1	0,0

(a) Rock-Paper-Scissors

	A	B
A	10	0
B	0	10

(b) Coordination Game

	C	D
C	-1,-1	-5,0
D	0,-5	-3,-3

(c) Prisoner's Dilemma

Figure 3.2: Examples of matrix games:
(a) is a zero-sum game, (b) a common reward game and (c) a general sum game [1]

3.1.2 Repeated Matrix Games

Repeated matrix games are the most basic way of sequential multi-agent interaction: they repeat the same matrix game for a finite or infinite number of steps. At each time step t each agent samples an action a_i^t with probability $\pi_i(a_i^t|h^t)$, where $h^t = (a^0, \ldots, a^{t-1})$ is the joint action history. Given a^t, each agent receives a reward $r_i^t = R_i(a^t)$.

Besides the time index t, the important addition in repeated matrix games is that policies can now make decisions based on the entire history of past joint actions. [1]

3.1.3 Stochastic Games

Stochastic games define a state-based environment in which the evolution of states and the agents' rewards depend on the actions of all agents. [22]

The components of a stochastic game are:

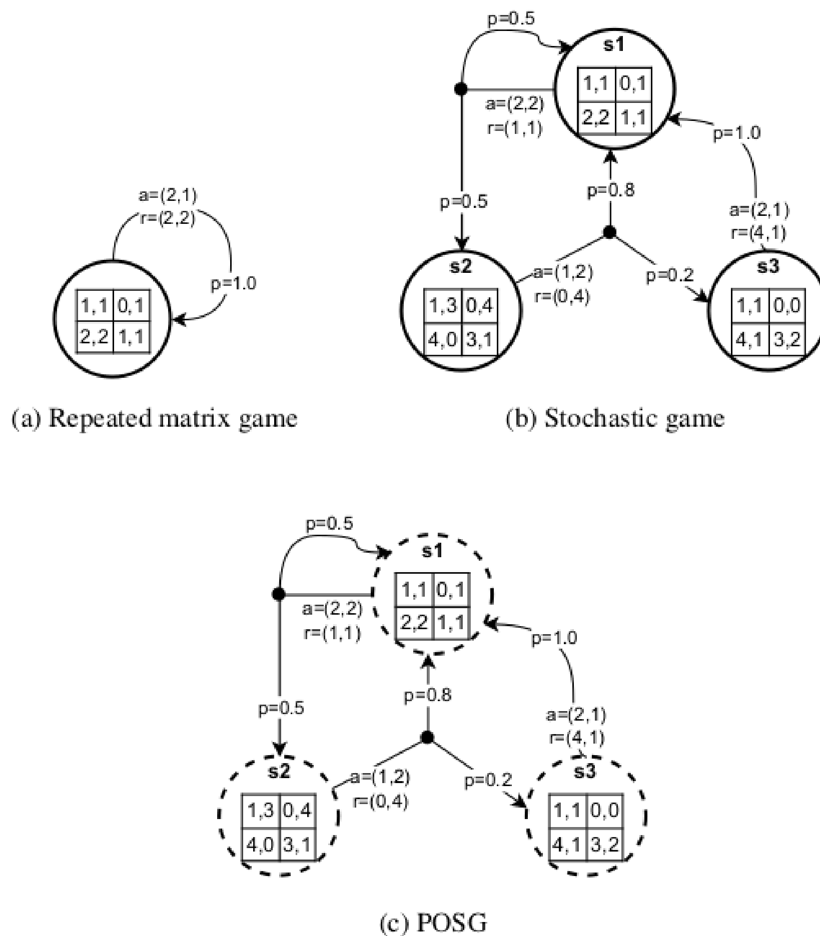
- Finite set of agents $\mathcal{I} = \{1, \ldots, n\}$
- Finite set of states $\mathcal{S}$
- For each agent $i \in \mathcal{I}$:
  - Finite set of actions $\mathcal{A}_i$ ($\mathcal{A} = \prod_{i=1}^{n} \mathcal{A}_i$)
  - Reward function $R_i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$
- State transition probability function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$
- Initial state distribution $P^0 : \mathcal{S} \to [0, 1]$

A stochastic game proceeds as follows:

1. Start at $s_0 \in \mathcal{S}$ sampled from P^0

2. At each time step t, each agent i observes s_t and chooses action a_i^t with probability given by $\pi_i(a_i^t|h^t)$. $\implies a^t = (a_1^t, ..., a_n^t)$. Here $h^t = (s^0, a^0, ..., s^t)$ is observed from all the agents

3. The game transitions to s^{t+1} with probability $\mathcal{T}(s^t, a^t, s^{t+1})$, and each agent i receives reward $r_i^t = R_i(s^t, a^t, s^{t+1})$



(a) Repeated matrix game

(b) Stochastic game

(c) POSG

Figure 3.3: Example of Game Models [1]

3.1.4 Partially Observable Stochastic Games (POSG)

They are the most general class of game models. As we will see, almost every MARL problem can be defined as a POSG.

The agents receive observations which carry some incomplete information about the environment's state and agents' actions.

A POSG is defined by the same elements of a stochastic game, and additionally defines for each agent $i \in \mathcal{I}$:

- Finite set of observations O_i
- Observation function $\mathcal{O}_i : \mathcal{A} \times \mathcal{S} \times O_i \rightarrow [0, 1]$

The game proceeds as follows:

- Start at $s_0 \in \mathcal{S}$ sampled from P^0
- At each time step t, each agent i receives $o_i^t \in O_i$ with probability $\mathcal{O}_i(a^{t-1}, s^t, o_i^t)$ and chooses action a_i^t with probability given by $\pi_i(a_i^t|h_i^t)$.
  $\implies a^t = (a_1^t, ..., a_n^t)$. Here $h_i^t = (o_i^0, ..., o_i^t)$ is specific for each agent
- The game transitions to s^{t+1} with probability $\mathcal{T}(s^t, a^t, s^{t+1})$, and each agent i receives reward $r_i^t = R_i(s^t, a^t, s^{t+1})$

Some practical examples of game models can be seen in Figure 3.3.

3.2 Solution Concepts for Games

What is a solution to a game? This is a central question of Game Theory, and many different solution concepts have been proposed which specify when a collection of agent policies constitute a stable or desirable outcome. [1]

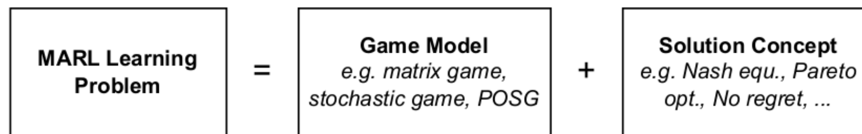


Figure 3.4: MARL problem definition [1]

3.2.1 Joint Policy and Expected Return

A game can be considered solved when we find a joint policy $\pi = (\pi_1, ..., \pi_n)$ which satisfies certain requirements defined by the solution concept in terms of expected return, $G_i(\pi)$, yielded at each agent i under the joint policy. [1]

Before defining the expected return in the context of the POSG model we give some additional notation:

- $\hat{h}^t = \{(s^\tau, o^\tau, a^\tau)_{\tau=0}^{t-1}, s^t, o^t\}$: full history up to time t

- $\sigma(\hat{h}^t) = (o^0, ..., o^t)$ returns the history of joint observations
- $\prod_i O_i(o^t|a^{t-1}, s^t)$: probability of a joint observation o^t

At this point we can give two equivalent definitions of expected returns:

- History-based expected return: define the set $\hat{H}$ to contain all full histories $\hat{h}^t$ for $t \to \infty$. Then,

$$G_i(\pi) = \mathbb{E}_{\hat{h}^t \in \hat{H}, \hat{h}^t \sim (P^0, \mathcal{T}, O, \pi)}[g_i(\hat{h}^t)] = \sum_{\hat{h}^t \in \hat{H}} P(\hat{h}^t|\pi) g_i(\hat{h}^t) \tag{3.1}$$

  where,

$$P(\hat{h}^t|\pi) = P^0(s^0) O(o^0|\emptyset, s^0) \prod_{\tau=0}^{t-1} \pi(a^\tau|h^\tau) \mathcal{T}(s^{\tau+1}|s^\tau, a^\tau) O(o^{\tau+1}|a^\tau, s^{\tau+1}) \tag{3.2}$$

  and

$$g_i(\hat{h}^t) = \sum_{\tau=0}^{t-1} \gamma^\tau R_i(s^\tau, a^\tau, s^{\tau+1}), \quad \gamma \in [0, 1] \tag{3.3}$$

- Recursive expected return:[1]

$$V_i^\pi(\hat{h}) = \sum_{a \in \mathcal{A}} \pi(a|\sigma(\hat{h})) Q_i^\pi(\hat{h}, a) \tag{3.4}$$

$$Q_i^\pi(\hat{h}, a) = \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s(\hat{h}), a)[R_i(s(\hat{h}), a, s') + \gamma \sum_{o' \in O} O(o'|a, s') V_i^\pi(< \hat{h}, a, s', o' >)] \tag{3.5}$$

  given those expressions we can define the expected return as

$$G_i(\pi) = \mathbb{E}_{s^0 \sim P^0, o^0 \sim O(\cdot|\emptyset, s^0)}[V_i^\pi(< s^0, o^0 >)] \tag{3.6}$$

3.2.2 Best Response

Given a set of policies for all agents except agent i, denoted by $\pi_{-i} = (\pi_1, ..., \pi_{i-1}, \pi_{i+1}, ..., \pi_n)$, a best response for agent i to π_{-i} is a policy π_i which maximizes the expected return for i when played against π_{-i}:

$$\text{BR}_i(\pi_{-i}) = \text{argmax}_{\pi_i} G_i(< \pi_i, \pi_{-i} >) \tag{3.7}$$

Note that the best response is not always unique.

[1]Analogous to the Bellman equations used for MDP theory.

3.2.3 Nash Equilibrium

In a general-sum game with n agent, a joint policy $\pi = (\pi_1, ..., \pi_n)$ is a Nash equilibrium (NE) if

$$\forall i, \pi'_i : \; G_i(\pi'_i, \pi_{-i}) \leq G_i(\pi) \tag{3.8}$$

In a NE no agent i can improve its expected return by changing its policy π_i specified in the equilibrium joint policy π, assuming the policies of the other agents stay fixed. [1]

$\implies \pi_i \in \text{BR}_i(\pi_{-i}) \quad \forall i$

Two important aspects of NE as a solution concept:

1. A NE is said to be deterministic if every policy π_i in the equilibrium π is deterministic. However it may also be probabilistic (if at least one policy π_i is probabilistic).
2. A game may have multiple Nash equilibria, and each equilibrium may give different expected returns to the agents. This leads to the so-called equilibrium selection problem.

Given a joint policy π, how to check if it is a NE?

For each agent i, we can keep the other policies π_{-i} fixed and compute an optimal best-response policy π'_i for i. If $G_i(\pi'_i, \pi_{-i}) > G_i(\pi_i, \pi_{-i})$, then we know that π is not a NE. [1]

3.2.4 Correlated Equilibrium

A restriction in Nash equilibria is that the policies must be probabilistically independent. The correlated equilibrium generalizes the NE by allowing for correlation between policies.

In a general-sum game with n agents, let $\pi_c(a|h)$ be a joint policy which assigns conditional probabilities to joint actions $a \in \mathcal{A}$ given joint observation history h.

Let $\phi_i : \mathcal{A}_i \rightarrow \mathcal{A}_i$ be an action modifier for agent i. Then, π_c is a correlated equilibrium if for all full histories $\hat{h}$, i and ϕ_i:

$$\sum_{a \in \mathcal{A}} \pi_c(a|\sigma(\hat{h}))(Q_i^{\pi_c}(\hat{h}, <\phi_i(a_i), a_{-i}>) - Q_i^{\pi_c}(\hat{h}, a)) \leq 0 \tag{3.9}$$

where $\sigma(\hat{t}) = (o^0, ..., o^t)$.

Clearly the set of CE contains the set of NE.

To better understand the concept of Correlated equilibrium we may consider each action recommendation a_i given by π_c to be a best response to the actions a_{-i} recommended to the other agents.

3.2.5 Conceptual Limitations of Equilibrium Solutions

- **Sub-Optimality:** In general, finding equilibrium solutions is not synonymous with maximising expected returns.
- **Non-Uniqueness:** Which of the different equilibria should the agents adopt, and how can they agree on a specific equilibrium? One approach to tackle equilibrium selection is to use additional criteria such as Pareto optimality and social welfare and fairness (see Sections 3.2.6 and 3.2.7).
- **Incompleteness:** An equilibrium solution π is incomplete in the sense that it does not specify equilibrium behaviours for off-equilibrium paths. An off-equilibrium path is any full history $\hat{h}$ which has probability $P(\hat{h}, \pi) = 0$ under the equilibrium π.

3.2.6 Pareto Optimality

We want to narrow down the space of equilibrium solutions by requiring additional criteria that a solution must achieve. One such criterion is Pareto optimality:

A joint policy π is Pareto-dominated by another joint policy π' if:

$$\forall i: \quad G_i(\pi') \geq G_i(\pi) \;\; \textit{and} \;\; \exists i : G_i(\pi') > G_i(\pi)$$

A joint policy π is Pareto-optimal if it is not Pareto-dominated by any other joint policy. We then also refer to the expected returns entailed by π as Pareto-optimal.

3.2.7 Social Welfare and Fairness

Pareto optimality does not make any statements about the total amount of rewards and their distribution among the agents. Thus, we may consider concepts of social welfare and fairness to further constrain the space of desirable solutions:

The social welfare of a joint policy π *is defined as* $W(\pi) = \sum_{i \in \mathcal{I}} G_i(\pi)$

A joint policy π *is welfare-optimal if* $\pi \in argmax_{\pi'} W(\pi')$

The social fairness of a joint policy π *is defined as* $F(\pi) = \prod_{i \in \mathcal{I}} G_i(\pi)$

A joint policy is fairness-optimal if $\pi \in argmax_{\pi'} F(\pi')$

It is easy to prove that welfare optimality implies Pareto optimality.

3.2.8 No Regret

The regret measures the difference between the rewards an agent received and the rewards it could have received if it had chosen a different action in the past against the observed actions of the other agents. Given a full history $\hat{h}^t$, the regret of agent i for not having chosen action a_i is defined as:

$$Regret_i(a_i|\hat{h}^t) = \sum_{\tau=0}^{t-1} R_i(s^\tau, < a_i, a_{-i}^\tau >, s^{\tau+1}) - R_i(s^\tau, a^\tau, s^{\tau+1}) \tag{3.10}$$

$\Longrightarrow$ The regret is a function of the history, and it is independent of the joint policy that generated that history.

A joint policy π *exhibits no regret if*

$$\forall i, a_i : \lim_{t \to \infty} \frac{1}{t} Regret_i(a_i|\hat{h}^t) \leq 0 \tag{3.11}$$

for all full histories $\hat{h}^t$ *with* $P(\hat{h}^t|\pi) > 0$

Despite its possible interesting applications, the regret has a conceptual limitation: indeed it assumes that the action of other agents $-i$ remain fixed in the history. This assumption is violated if the other agents condition their action choices on the past actions of agent i.

From this it also follows that minimising regret is not necessarily equivalent to maximising returns.

3.3 Tabular Multi-Agent Reinforcement Learning

3.3.1 General Learning Process

In classical Machine Learning, we call learning the process which optimises a model or function based on data. In our setting, the model is a joint policy consisting in policies for each agent, and the data consists in one or more histories in the game. The core elements of a learning process are:

- **Game model:** see Section 3.1
- **Data:** set of z histories $D^z = \{h^{t,e} | 1 \leq e \leq z\}, z \geq 0$. Each history $h^{t,e}$ was produced by a joint policy π^e, used during episode e. t is the length of the episode, which can be variable.
- **Learning algorithm** $\mathbb{L}$**:** $\pi^{z+1} = \mathbb{L}(D^z, \pi^z)$
- **Learning goal:** policy π^* which satisfies the properties of a chosen solution concept (introduced in the previous section).

3.3.2 Convergence Types

The main theoretical evaluation criterion is:

$$\lim_{z \to \infty} \pi^z = \pi^* \tag{3.12}$$

Sometimes we can also use weaker types of convergence:

- Convergence of expected return: $\lim_{z \to \infty} G_i(\pi^z) = G_i(\pi^*) \quad \forall i \in \mathcal{I}$
- Convergence of empirical action distribution: $\lim_{z \to \infty} \bar{\pi}^z = \pi^*$ where $\bar{\pi}^z(a|h) = \frac{1}{z} \sum_{x=1}^{z} \pi^x(a|h)$
- Convergence of average return: $\lim_{z \to \infty} G_i(\bar{\pi}^z) = G_i(\pi^*) \ \forall i \in \mathcal{I}$

In large and complex games, it is almost always computationally intractable to check for these convergence properties. Instead, a common approach is to monitor the expected returns $G_i(\pi^z)$ achieved by the joint policy as z increases, usually by visualising learning curves that show the progress of expected returns during learning. That is also what we will do when we will evaluate our experimental results in Chapter 6.

At this point we are ready to introduce some different strategies for designing tabular MARL algorithms.

3.3.3 Central Learning

In this approach we train a single central policy π_c, which receives the local observations from all the agents and selects an action for each one of them, by selecting joint actions from $\mathcal{A} = \mathcal{A}_1 \times ... \times \mathcal{A}_n$. It essentially is a way to reduce the problem to a single agent RL setup.

Algorithm 6 Central Q-Learning

Algorithm parameters: small $\epsilon > 0$, step size $\alpha \in (0, 1]$
Initialize: $Q(s, a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$

for all episodes **do**
  for all steps t of the episode **do**
    Observe current state s^t
    Choose a^t from s^t using policy derived from Q (e.g. ϵ-greedy)
    Take action a^t, observe rewards $r_1^t, ..., r_n^t$ and next state s^{t+1}
    Transform $r_1^t, ..., r_n^t$ into scalar reward r^t
    $Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \alpha[r^t + \gamma \max_{a'} Q(s^{t+1}, a') - Q(s^t, a^t)]$
  end for
end for

Despite the advantages that this approach could give in circumventing the multi-agent aspects of the non-stationarity and credit assignment problems, in practice it has a number of limitations:

1. How to transform the joint reward $(r_1, ..., r_n)$ into a single scalar reward r? It is easy for common-reward games, where $r_i = r \; \forall i \in [1, n]$, but very complicated for zero-sum and general-sum games.
2. The action space grows exponentially in the number of agents.
3. Agents are often localised entities which are physically or virtually distributed $\Longrightarrow$ they need a local policy π_i, based just on local observations.[2]

3.3.4 Independent Learning

This approach is conceptually the opposite of the previous one. Here each agent learns its own policy π_i using only the local history of its own observations,

[2]This is the main reason for which we will not use Central Learning for Autonomous Driving.

actions and rewards, while ignoring the existence of other agents (which are considered as part of the environment).

Note that this again a way to go back to a single agent RL problem, this time seen from the perspective of each agent.

Algorithm 7 Independent Q-Learning

The algorithm must be run in parallel for each agent i
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize: $Q_i(s, a_i) = 0$ for all $s \in \mathcal{S}, a_i \in \mathcal{A}_i$

for all episodes **do**
  for all steps t of the episode **do**
    Observe current state s^t
    Choose a_i^t from s^t using policy derived from Q_i (e.g. ϵ-greedy), meanwhile other agents $j \neq i$ choose their actions a_j^t
    Observe action a^t, own reward r_i^t and next state s^{t+1}
    $Q_i(s^t, a_i^t) \leftarrow Q(s^t, a_i^t) + \alpha[r_i^t + \gamma \max_{a_i'} Q(s^{t+1}, a_i') - Q(s^t, a_i^t)]$
  end for
end for

From the perspective of each agent i the policies π_j of other agents $j \neq i$ become part of the environment's state transition function:

$$\mathcal{T}_i(s^{t+1}|s^t, a_i^t) = \eta \sum_{a_{-i} \in \mathcal{A}_{-i}} \mathcal{T}(s^{t+1}|s^t, < a_i^t, a_{-i} >) \prod_{j \neq i} \pi_j(a_j|s^t) \tag{3.13}$$

This fact gives rise to a non-stationarity problem caused by the concurrent learning of all agents, and in some cases it may produce unstable learning and it may not converge to any solution of the game.

Despite this downside, independent learning algorithms are still competitive with state-of-the-art MARL algorithms, as we will see in the following chapters.

3.3.5 TD Learning for Games: Joint Action Learning

Joint Action Learning (JAL) refers to a family of MARL algorithms based on Temporal Difference Learning which seek to address the issues present both in Central and Independent Learning.

Given the action value function for each agent i under joint policy π:

$$Q_i^\pi(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a)[R_i(s,a,s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_i^\pi(s',a')] \tag{3.14}$$

the aim is to learn equilibrium values for the Q-function $Q_i^{\pi^*}$, where π^* is an equilibrium joint policy for the stochastic game.

To learn equilibrium values we can use the solution concepts from Game Theory introduced in the previous sections. The idea is to view the set of joint action values $Q_1(s,\cdot),...,Q_n(s,\cdot)$ as a matrix game Γ_s for state s, in which the reward function for agent i is given by:

$$R_i(a_1,...,a_n) = Q_i(s,a_1,...,a_n) \tag{3.15}$$

Just to give a visual example, consider a stochastic game with two agents i and j, and three possible actions for each agent. The matrix game $\Gamma_s = \{\Gamma_{s,i}, \Gamma_{s,j}\}$ in each state s can be written as:

$$\Gamma_{s,i} = \begin{bmatrix} Q_i(s,a_{i,1},a_{j,1}) & Q_i(s,a_{i,1},a_{j,2}) & Q_i(s,a_{i,1},a_{j,3}) \\ Q_i(s,a_{i,2},a_{j,1}) & Q_i(s,a_{i,2},a_{j,2}) & Q_i(s,a_{i,2},a_{j,3}) \\ Q_i(s,a_{i,3},a_{j,1}) & Q_i(s,a_{i,3},a_{j,2}) & Q_i(s,a_{i,3},a_{j,3}) \end{bmatrix}$$

with an analogous matrix $\Gamma_{s,j}$ for agent j using Q_j.

After having solved Γ_s at each time step to get the joint policy π, we perform action selection following an ϵ-greedy approach. Finally each action value function Q_i is updated following a rule very similar to the one in the Q-Learning algorithm.

Starting from the pseudocode written in the next page we can build many different algorithms, by choosing which solution concept to adopt for solving the matrix game at each time step to get the policies $(\pi_1,...,\pi_n)$.

An example could be that of using Nash Equilibrium as solution concept, but we must notice that highly restrictive assumptions are needed to solve the problem.

Another choice could be that of computing Correlated Equilibria. This has the benefit of being easier to compute, but since the space of correlated equilibria is in general larger than the space of Nash equilibria, the equilibrium selection problem would become even more pronounced.

Algorithm 8 Joint Action Learning with Game Theory

The algorithm must be run in parallel for each agent i
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize: $Q_i(s, a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}, i \in \mathcal{I}$

for all episodes **do**
  for all steps t of the episode **do**
    Observe current state s^t
    With probability ϵ choose random action a_i^t
    Else: solve Γ_{s^t} to get policies $(\pi_1, ..., \pi_n)$, then sample action $a_i^t \sim \pi_i$
    Observe joint action a^t, rewards $r_1^t, ..., r_n^t$ and next state s^{t+1}
    for all $i \in \mathcal{I}$ **do**
      $Q_i(s^t, a^t) \leftarrow Q_i(s^t, a^t) + \alpha[r_i^t + \gamma Value_i(\Gamma_{s^{t+1}}) - Q_i(s^t, a^t)]$
      # where $Value_i(\Gamma_{s_{t+1}}) = \sum_{a \in \mathcal{A}} \Gamma_{s^{t+1},i}(a) \pi_{s+1}(a)$
    end for
  end for
end for

3.3.6 The complexity of computing equilibria

Up to these days no efficient MARL algorithm has been developed to compute Nash equilibria in polynomial time, and it has also been shown [2] that it is likely that such algorithm does not even exist. Indeed it is probable that any MARL algorithm will require exponential time in the worst case.

For this reason when we will switch to the Deep-MARL framework in the next chapter we will not look for theoretical proofs of convergence to some kind of equilibria, but we will instead focus on the empirical results that an algorithm is able to achieve when trying to solve a particular problem.

4

Deep-MARL Framework and Algorithms

As we saw in the previous chapters, Multi-Agent Reinforcement Learning is a research area in the middle between Machine Learning and Game Theory. The latter gives us the theoretical foundations and solution concepts, while the former is the crucial tool that can help us to exploit data in order to solve real problems. Indeed we saw at the end of last chapter that when facing a complex task, it is usually impossible to find closed form solutions or convergence proofs, but we have to resort to learning methods in order to solve the problem.

Starting from this chapter we enter in the core part of the thesis. We will begin by analyzing the crucial challenges of a Multi-Agent Reinforcement Learning problem, that must be kept in mind every time we want to design a MARL algorithm. Later we will give a brief overview and classification of Deep-MARL algorithms, to introduce the different paradigms and strategies that currently exist to face a Multi-Agent Reinforcement Learning problem.

Finally we will describe in depth the four algorithms that we will use in the experimental part of the thesis, to face Multi-Agent Autonomous Driving scenarios. These algorithms have been chosen both because they are representative of the three important categories of MARL algorithms, and also because they are among the most used algorithms in literature.

4.1 Challenges of Multi-Agent RL

Due to its huge complexity, there exist a number of challenges that make the search for a solution in a MARL problem really difficult. Moreover this issues are summed to the ones already present in the single agent Deep-RL scenario. For these reasons a universal strategy to solve MARL problems does not exist. This is for some reasons a flaw, but it is also what makes this research area very interesting and potentially extremely powerful.

In this section we will talk about the three main challenges of a Multi-Agent Reinforcement Learning problem.

4.1.1 Non-stationarity caused by learning agents

Before talking of the non-stationarity problem caused by the presence of multiple agents, we must note that non-stationarity already existed in the single agent RL setup. This is because in an MDP the stochastic process $\mathcal{X}^t$, which samples the state s^t at each time step t is non-stationary. Indeed it is completely defined by the state transition probability function $\mathcal{T}(s^t|s^{t-1}, a^{t-1})$ and the agent's policy π, which selects actions $a^t \sim \pi(\cdot|s^t)$. The fact that the latter changes over time makes the process $\mathcal{X}^t$ non-stationary.

This problem becomes much bigger in the multi-agent setup, because all the agents change their policy over time, and for this reason the entire environment appears to be non-stationary from the point of view of each agent. This is one of the main reasons for which designing MARL algorithms that have useful learning guarantees is very difficult and subject of ongoing research. [1]

4.1.2 Multi-Agent Credit Assignment

Similarly to the non-stationarity problem, also the credit assignment is a challenge already present in single agent RL. In that case we talk about temporal credit assignment. It essentially is the problem of determining, during the learning process, which specific actions have been crucial to achieve the objective of the task.

This is extremely difficult, because many times the positive reward is given just at the end of the episode, and understanding which actions, maybe far away in the past, have been crucial to achieving a particular result is very complex.

In the multi-agent setting this problems is much bigger, because on top of the temporal credit assignment problem we also have to understand which agents performed crucial actions for reaching a certain result. To summarize:

> *Credit assignment in RL is the problem of determining which past actions have contributed to received rewards. In Multi-Agent RL, there is the additional question of whose actions among multiple agents contributed to the rewards. [1]*

An interesting way to face multi-agent credit assignment consists in attributing values to joint actions. This can be easily seen in a simple example of the famous Rock-Paper-Scissors game. Consider the two agents select actions (a_1, a_2) = (Rock,Scissors) and agent 1 receives a reward of +1. Then they choose actions (a_1, a_2) = (Rock,Paper) and agent 1 receives a reward of -1.

If agent 1 uses an action value function $Q(s, a_1)$ which depends just on a_1, the average value of taking action R would be 0, and this does not represent well the impact of agent 2's action. On the contrary, a joint action value function $Q(s, a_1, a_2)$ can correctly represent the impact of agent 2's action when agent 1 plays Rock.

4.1.3 Scaling to many Agents

An important goal of MARL is that of scaling efficiently to many agents. Indeed up to these days state-of-the-art MARL algorithms can efficiently manage a small number of agents in complex problems (usually not more than 5/6, but in many problems we are limited to just 2/3 agents). This goal is complicated mainly by the fact that the joint action space can grow exponentially with the number of agents.

Clearly the use of Deep Learning approaches in Multi-Agent Reinforcement Learning algorithms is driven also by the purpose of improving scalability.

4.2 Classification of MARL algorithms

Multi-Agent Reinforcement Learning algorithms[1] can be divided into three big categories, depending on the type of information available to the agents

[1]Since from now on we will only talk about Deep-MARL algorithms, we will drop the prefix "Deep" for ease of notation.

during training and execution:

- Centralized Training and Execution (CTCE)
- Decentralized Training and Execution (DTDE)
- Centralized Training and Decentralized Execution (CTDE)

4.2.1 Centralized Training and Execution

This paradigm will be analyzed very shortly, since it is not suitable to address the Autonomous Driving challenge. Similarly to Section 3.3.3, we consider all the n agents as a single entity, which observes the global state s at each time step and takes a joint action $a = (a_1, ..., a_n)$. This setup remains the same both at training and test time.

This is a simple approach and it is able to address some of the MARL challenges, but just because we are reducing the problem to the single agent setup. The reasons for which this approach is almost never used in practice are evident, and they are the same explained in Section 3.3.3. For Autonomous Driving in particular, we cannot realistically expect to transmit and receive sensor information among the different vehicles in real time, and even if it was possible, learning a centralized policy to control all vehicles would be very difficult due to the complexity of the problem.

4.2.2 Decentralised Training and Execution

This approach is on the opposite extreme with respect to the CTCE paradigm. In this setup we treat each agent in a completely independent way, both at training and test time. As we already saw in Section 3.3.4, this is again a way to go back to the single agent case, but this time from the perspective of each agent.

This paradigm is interesting because it can improve scalability, since it avoids the exponential growth in the action space. However training can be highly affected by non-stationarity, because from the point of view of each agent the environment dynamics is continuously changing due to the varying policies of the other agents.

Despite of this downside, DTDE algorithms are still widely used in Multi-Agent Reinforcement Learning problems because they have shown good empirical results in an important variety of situations.

In our case of study, Autonomous Driving in traffic scenarios, we will test the most famous DTDE algorithm, called Independent Deep Q-Networks, which, as we will see, is able to achieve surprisingly good results in all the developed scenarios.

For a detailed explanation and pseudocode of this algorithm we refer to Section 4.3.

4.2.3 Centralized Training and Decentralized Execution

This third paradigm tries to combine the benefits of the previous two, by allowing the agents to access global information during training, while relying just on their local observations at test time.

This can be useful to reduce the non-stationarity problem of the DTDE paradigm, while keeping the possibility of having physically or virtually distributed agents, which is a crucial aspect in a major number of applications.

In literature there exist both policy-based and value-based CTDE algorithms. The former is a category mainly composed by (Multi-Agent) Actor-Critic algorithms, where the critic is learnt using global information. The policy of each agent is instead trained exploiting this global critic, but at test time it is only dependent on the local observations of the corresponding agent.

These algorithms are interesting because they can be applied both in common-reward and general-sum games, but they have the downside of being high variance, like the policy-based algorithms in single agent RL.

In the experimental part of the thesis we will test one of the first but yet most used policy-based CTDE algorithms, called Multi-Agent Deep Deterministic Policy Gradient (MADDPG). Its detailed analysis is postponed to Section 4.4.

Finally the second category of CTDE algorithms consists of value-based strategies that exploit a technique called value decomposition. The starting point for this strategy is a centralized action value function that is conditioned on the observations and actions of all the agents. To allow for local execution this function is then decomposed into different utility functions, one for each agent.

For simplicity we can think of the utility function of each agent as if it was

its local action value function. Also here training is centralized, while the action selection step of each agent just relies on its local observations, exploiting the utility function and following some exploration-based policy, like the already mentioned ϵ-greedy strategy.

Different value based CTDE algorithms exist, depending on the strategy that we use to decompose the centralized action value function. For our Autonomous Driving challenge we will test both the VDN and the QMIX algorithms (for their thorough description see Section 4.5).

4.3 Independent Deep Q-Networks

Independent Deep Q-Networks (IDQN) is a DTDE strategy that exploits the famous DQN algorithm [17] in the Multi-Agent setup, by simply running it in parallel for each agent.

As already discussed in Section 2.6, DQN is the Deep Learning extension of the Q-Learning algorithm, where the action value function is approximated using a Neural Network. In particular we use an architecture called Convolutional Network (CNN) [14], which has been specifically developed to work with images. For this reason when using this algorithm the observation of each agent has to be an image.

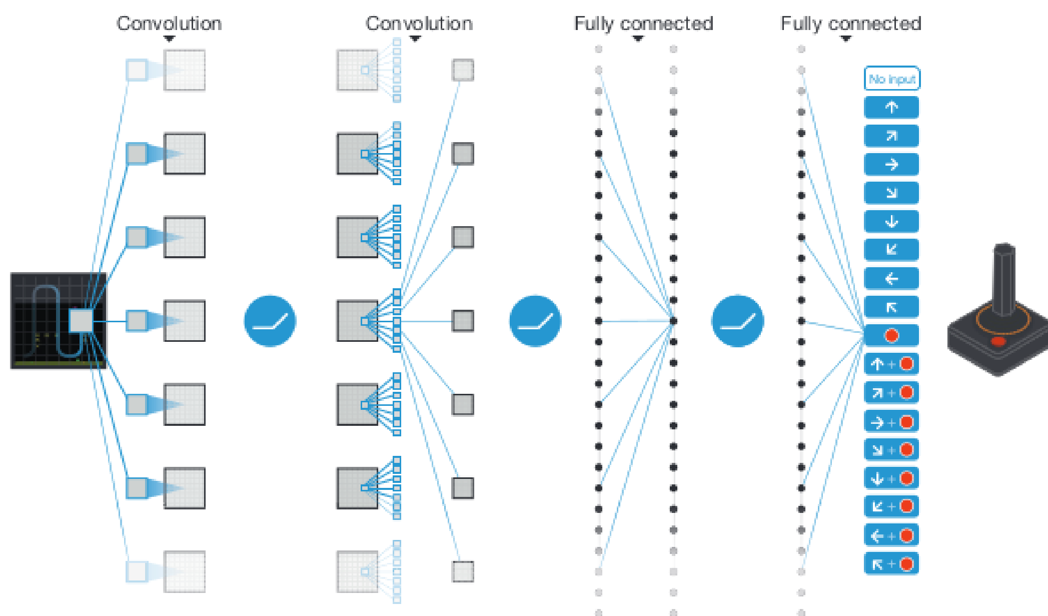


Figure 4.1: Example of CNN to approximate the action value function [17]

Usually the network takes as input the observation of the agent and outputs the corresponding Q-value function for each possible action.

As we already said in Chapter 2, simply substituting the tabular Q-function with a CNN is not sufficient, because we have to deal with a couple of problems introduced by the fact that we are approximating the action value function with a Neural Network (recall Section 2.6):

- **Moving target problem:**

  > *we use an iterative update that adjusts the action values toward target values that are only periodically updated, thereby reducing correlations with the target. [17]*

  In other words, we keep a copy of the Q-Network and we use it when computing the targets values for the Q-Learning update, but we modify its parameters only periodically, in order to reduce correlations between the current Q-function and the target.

- **Breaking correlations:**

  > *we use a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. [17]*

  To do that we initialize a replay buffer for each agent, and at each time step we store the agent's experience (o^t, a^t, r^t, o^{t+1}). Then, during learning we draw uniformly at random minibatches of experience $\mathcal{B}$, and we perform a learning step using these data. In this way we remove the correlation between consecutive samples.

Finally we recall that in the learning phase we update the parameters of the network of each agent by minimising the following loss function:

$$\mathcal{L}(\theta_i) = \frac{1}{B} \sum_{(h_i^t, a_i^t, r_i^t, h_i^{t+1}) \in \mathcal{B}} (r_i^t + \gamma \max_{a_i} Q_i(h_i^{t+1}, a_i; \bar{\theta}_i) - Q_i(h_i^t, a_i^t; \theta_i))^2 \quad (4.1)$$

As we can see from the pseudocode in the following page, the structure of the algorithm is very similar to Q-Learning, except for the peculiarities just discussed and fact that we must run it in parallel for all the agents.

The DTDE paradigm in this algorithm is quite clear, indeed each agent is completely independent with respect to the others, both at training and test time.

Algorithm 9 Independent Deep Q-Networks

Initialize n action value networks Q_i with random parameters $\theta_1, ..., \theta_n$
Initialize n target action value networks $\bar{Q}_i$ with parameters $\bar{\theta}_1 = \theta_1, ..., \bar{\theta}_n = \theta_n$
Initialize a replay buffer for each agent $\mathcal{D}_1, ..., \mathcal{D}_n$

for all episodes **do**
  Collect environment observations $o_1^0, ..., o_n^0$
  for all time steps t of the episode **do**
    for agent $i = 1, ..., n$ **do**
      With probability ϵ select a random action a_i^t
      otherwise select $a_i^t = \text{argmax}_a Q_i(h_i^t, a; \theta_i)$, where $h_i^t = (o_i^0, ..., o_i^t)$
    end for
    Apply actions and collect observations o_i^{t+1} and rewards r_i^t for all the agents
    Store transitions in the replay buffers $\mathcal{D}_1, ..., \mathcal{D}_n$
    for agent $i = 1, ..., n$ **do**
      Sample random mini-batch of transitions from replay buffer $\mathcal{D}_i$
      Update parameters θ_i by minimising the loss function from Equation 4.1
    end for
    Every k steps, update target network parameters $\bar{\theta}_i$ for each agent i
  end for
end for

Before moving to the next algorithm, we will briefly introduce two possible variations to the IDQN algorithm, that can improve its performances. Since these two strategies are not mutually exclusive, we will have four possible versions of the IDQN algorithm, and in the experimental section of this work we will analyze them all, highlighting the different results among them.

4.3.1 Independent Double-DQN (IDDQN)

A problem of Q-Learning, and consequently also of DQN, is that sometimes it overestimates action values. In some conditions this is not necessarily a problem, but if overestimations are not uniform, then they might negatively affect the quality of the resulting policy.

To address this issue, a slight modification to the standard DQN algorithm has been proposed some years ago [29]. Consider the standard update target $r + \gamma \max_{a'} Q(s, a'; \bar{\theta})$. It has been shown that what leads to overoptimistic value

estimates is the fact that in the max operator we are using the same values both to select and to evaluate an action. To solve this problem the key idea is to decouple the action selection and evaluation step, by using two different networks (recall that now we are just using the target network). For this reason we exploit the online network in the action selection step, while we keep using the target network for evaluation. Finally the new update target becomes:

$$r + \gamma Q(s, \text{argmax}_{a'} Q(s, a'; \theta); \bar{\theta}) \tag{4.2}$$

Note that the pseudocode for the IDDQN algorithm is exactly the same of IDQN, apart from the slight modification of the update target in the loss function equation (4.1).

4.3.2 Independent Dueling-DQN

This second modification comes from the fact that in many RL and also MARL problems there are some states where actions do not affect the environment in any relevant way. In order to learn which states are (or not) valuable, the *dueling architecture* has been developed. [31]

As the name suggests, the modification brought by this approach is on the architecture of the Neural Network, not on the structure of the algorithm.

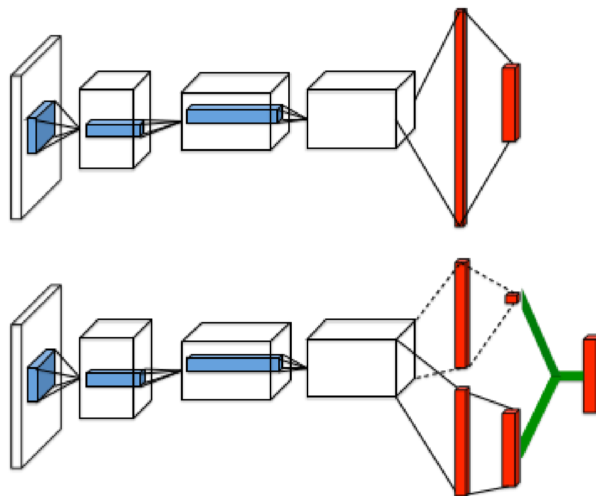


Figure 4.2: Comparison between the classical structure of a Deep Q-Network (top) and the Dueling architecture (bottom) [31]

Looking at Figure 4.2 we can see that the dueling architecture consists of two streams that are combined through an aggregating layer to produce an estimate of the action value function.

The first stream approximates the state value function, giving the information of how good it is to be in a particular state. The second stream instead estimates the advantage function A, defined in this way:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{4.3}$$

This function gives a relative measure of the importance of each action in a particular state.

When designing the network architecture, a naïve approach could be that of building the aggregated model like this: $Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$[2]. This equation however is unidentifiable, in the sense that given Q we cannot recover V and A uniquely.

To address this issue we can implement different strategies, and for our Autonomous Driving application we decided to choose the following because it increased the stability of the optimization:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)) \tag{4.4}$$

As we will see, this network architecture is particularly suited for the Multi-Agent Autonomous Driving problem, because the value stream can learn to pay attention to the road, while the advantage stream learns to pay attention when there are cars in front, in order to avoid collisions.

4.4 Multi-Agent Deep Deterministic Policy Gradient

The second algorithm that we will consider is Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [15]. It is one of the first Multi-Agent policy gradient algorithms following the CTDE paradigm.

Its core idea consists in extending the single agent Actor Critic policy gradient

[2]Here, θ denotes the parameters of the convolutional layers, while α and β are the parameters of the two streams of fully connected layers.

method to the Multi-Agent setup. To do that we augment the critic of each agent with extra information about the policy of other agents, while each policy relies just on the local observations of the corresponding agent.

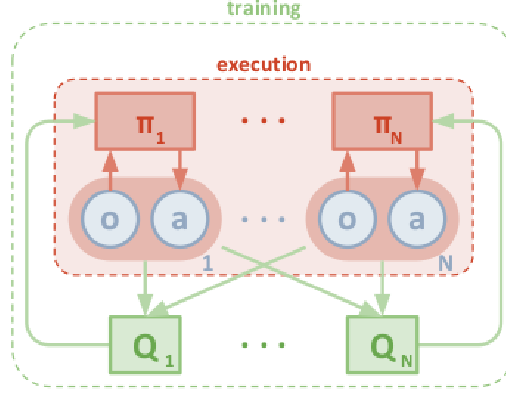


Figure 4.3: Structure of Multi-Agent centralized critic, decentralized actor [15]

Similarly to Section 2.7, when we introduced the policy gradient theorem, we can write the gradient of the expected return for agent i, $J(\theta_i) = \mathbb{E}_\pi[r_i]$ as:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{x,a_i\sim\pi_i}[\nabla_{\theta_i}\log \pi_i(a_i|o_i) Q_i^\pi(x, a_1, ..., a_n)] \tag{4.5}$$

where $Q_i^\pi(x, a_1, ..., a_n)$ is the centralized action value function that takes as input the actions of all the n agents, and some state information x, which is the concatenation of all the agents' observations.

This idea can be extended to handle deterministic policies, with a slight modification of the gradient's expression:

$$\nabla_{\theta_i} J(\mu_i) = \mathbb{E}_{x,a\sim\mathcal{D}}[\nabla_{\theta_i}\mu_i(a_i|o_i)\nabla_{a_i} Q_i^\mu(x, a_1, ..., a_n)|_{a_i=\mu_i(o_i)}] \tag{4.6}$$

Note that also in this case we are using a replay buffer $\mathcal{D}$, containing the tuples $(x, x', a_1, ..., a_n, r_1, ..., r_n)$.

Finally, the centralized action value function of each agent Q_i^μ is updated in this way:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{x,a,r,x'}[(Q_i^\mu(x, a_1, ..., a_n) - y)^2], \quad y = r_i + \gamma Q_i^{\mu'}(x', a_1', ..., a_n')|_{a_j'=\mu_j'(o_j)} \tag{4.7}$$

The general structure of the algorithm is very similar to the single agent Actor Critic algorithm presented in Section 2.7, except for the normal extensions necessary to handle the Multi-Agent setup, as visible from Algorithm 10.

Algorithm 10 Multi-Agent Deep Deterministic Policy Gradient

for all episodes **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $x^0 = (o_1^0, \dots, o_n^0)$
  for all time steps t of the episode **do**
    for agent $i = 1, \dots, n$ **do**
      Select action $a_i^t = \mu_i(o_i^t)$
    end for
    Execute actions $a = (a_1, \dots, a_n)$ and observe rewards $r_i^t, \dots, r_n^t$ and new state $x^{t+1} = (o_1^{t+1}, \dots, o_n^{t+1})$
    Store (x^t, a^t, r^t, x^{t+1}) in a replay buffer $\mathcal{D}$
    for agent $i = 1, \dots, n$ **do**
      Sample random mini-batch of B transitions (x, a, r, x') from replay buffer $\mathcal{D}$
      Update the critic by minimising the loss from Equation 4.7
      Update the actor using the sampled policy gradient recovered from Equation 4.6
    end for
    Every k steps, update target network parameters for each agent i:

$$\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i \tag{4.8}$$

  end for
end for

In this algorithm we can see the typical approach for exploring the environment when using policy-based methods, that is to perturb the action given by the policy using some kind of noise (usually white Gaussian noise). Clearly this step is performed only during training, while at test time we remove the perturbation to maximise the performances.

Like in IDQN, also in this algorithm we use target networks to reduce the moving target problem when optimizing both the actor and the critic. However, differently from before, we perform a soft update for the parameters of the target networks every k time steps (see Equation 4.8).

4.5 Value Decomposition Networks & QMIX

The last two algorithms that we will analyze and later on test in the experimental part of the thesis are Value Decomposition Networks (VDN) and QMIX. They both are value-based algorithms following the Centralized Training and Decentralized Execution paradigm.

The starting point for both the algorithms is a centralized action value function $Q(s,a;\theta) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r^t | s, a]$. Since it is difficult to directly learn this function, the idea is to decompose it into simpler quantities, called utility functions (one for each agent), which can be learned more efficiently. Moreover this is crucial to apply the CTDE paradigm, because at the test time each agent will just rely on the information given by its own utility function.

Such decomposition is most commonly used in common reward games, in which all agents have the same reward function. For this reason when we will test these algorithms in our setup we will need to modify the definition of the problem, to make it cooperative.

We note that whatever decomposition we choose for $Q(s,a;\theta)$, it must satisfy the *individual-global-max* (IGM) property:

$$\arg\max_{a=(a_1,\dots,a_n)} Q(s,a;\theta) = \begin{pmatrix} \arg\max_{a_1} Q_1(h_1,a_1;\theta_1) \\ \vdots \\ \arg\max_{a_n} Q_n(h_n,a_n;\theta_n) \end{pmatrix} \tag{4.9}$$

where h_i is the observation history of agent i.

The VDN algorithm exploits a linear decomposition of the centralized action value function:

$$Q(s,a;\theta) = \sum_{i \in \mathcal{I}} Q_i(h_i,a_i;\theta_i) \tag{4.10}$$

Despite its simplicity, this approach has been proven effective in a number of applications [23]. Also in our Multi-Agent Autonomous Driving setting we will see the effectiveness of this kind of decomposition.

As usual in Deep-MARL algorithms, also in this case we keep a replay buffer $\mathcal{D}$, where we store the experience of all the agents. The structure of the algorithm

is very similar to IDQN, but we just optimise the following loss:

$$\mathcal{L}(\theta) = \frac{1}{B} \sum_{(h_t, a_t, r_t, h_{t+1}) \in \mathcal{B}} (r_t + \gamma \max_a Q(h_{t+1}, a; \bar{\theta}) - Q(h_t, a_t; \theta))^2 \tag{4.11}$$

where $\mathcal{B}$ is a batch of transitions and the optimisation objective is propagated through the individual utilities of all the agents.

The only difference between VDN and QMIX is the type of decomposition that we apply to the centralized Q-function. While the decomposition of VDN is linear and simple, it may not always be realistic. The key observation to introduce QMIX is that to satisfy the IGM property we just have to satisfy the following relation between $Q(s, a, \theta)$ and the utility functions of all the agents: [20]

$$\frac{\partial Q}{\partial Q_i} \geq 0, \quad \forall i \in \mathcal{I} \tag{4.12}$$

To enforce this relation, QMIX represents Q using an architecture consisting of a mixing network and a set of hypernetworks [9]. The mixing network takes as input all the utility functions of the agents, and mixes them monotonically to produce the values of Q. To enforce the monotonicity constraint of Equation 4.12, all the weights of the mixing network are restricted to be non-negative.

These weights are produced by hypernetworks, which take as input the global state s and generate the weights of all the layers in the mixing network. In this way we are also able to condition the centralized action value function on global state information.

Finally, to ensure that the weights of the mixing network are non-negative, we use the absolute function as non-linear activation for all the hypernetworks.

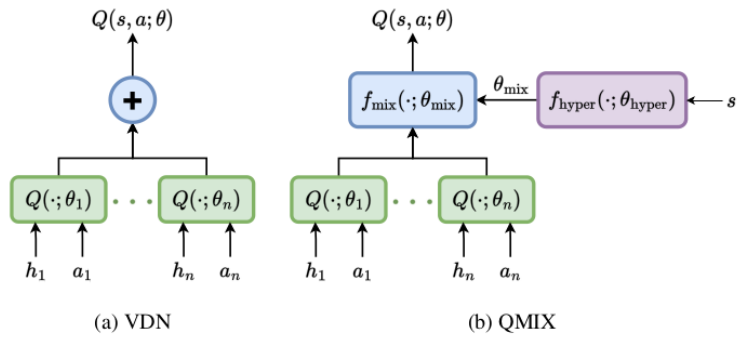


Figure 4.4: Network architecture of VDN and QMIX [1]

As we will see in Chapter 6, the higher complexity of QMIX with respect to VDN is not always synonym of better performances, due to the huge number of parameters to optimise and hyperparameters to tune.

Algorithm 11 VDN and QMIX

Initialize n utility networks Q_i with random parameters $\theta_1, ..., \theta_n$
Initialize n target utility networks $\bar{Q}_i$ with parameters
$\bar{\theta}_1 = \theta_1, ..., \bar{\theta}_n = \theta_n$
Initialize a replay buffer $\mathcal{D}$
if QMIX **then**
  Initialize the mixing network and all the hypernetworks
end if

for all episodes **do**
  Collect environment observations $o_1^0, ..., o_n^0$
  for all time steps t of the episode **do**
    for agent $i = 1, ..., n$ **do**
      With probability ϵ select a random action a_i^t
      otherwise select $a_i^t = \text{argmax}_a Q_i(h_i^t, a; \theta_i)$, where $h_i^t = (o_i^0, ..., o_i^t)$
    end for
    Apply actions and collect observations o_i^{t+1} and reward r^t for all the agents
    Store transition in the replay buffer $\mathcal{D}$
    Sample a random mini-batch of transitions from the replay buffer $\mathcal{D}$
    Update the parameters θ of all the networks by minimising the loss function from Equation 4.11
    Every k steps, update the parameters of all the target networks $\bar{\theta}$
  end for
end for

A final note about these two algorithms regards the network architecture used to approximate the utility functions of the agents. Differently from the previous approaches, in this case we implement Recurrent Neural Networks, exploiting the GRU architecture [3], to facilitate learning over longer timescales.

5

Experimental Setup

With this chapter we enter in the experimental section of the thesis. As already mentioned, Multi-Agent Autonomous Driving is an extremely interesting and studied research area in these years. Despite of that, the problem of how competently interact with diverse road users in diverse scenarios remains largely unsolved. Indeed the Autonomous Vehicles available nowadays in the industrial market are capable of driving in situations where few social interactions are required, but when encountering complexly interactive scenarios, they tend to slow down and wait rather than acting proactively to find a solution. [32]

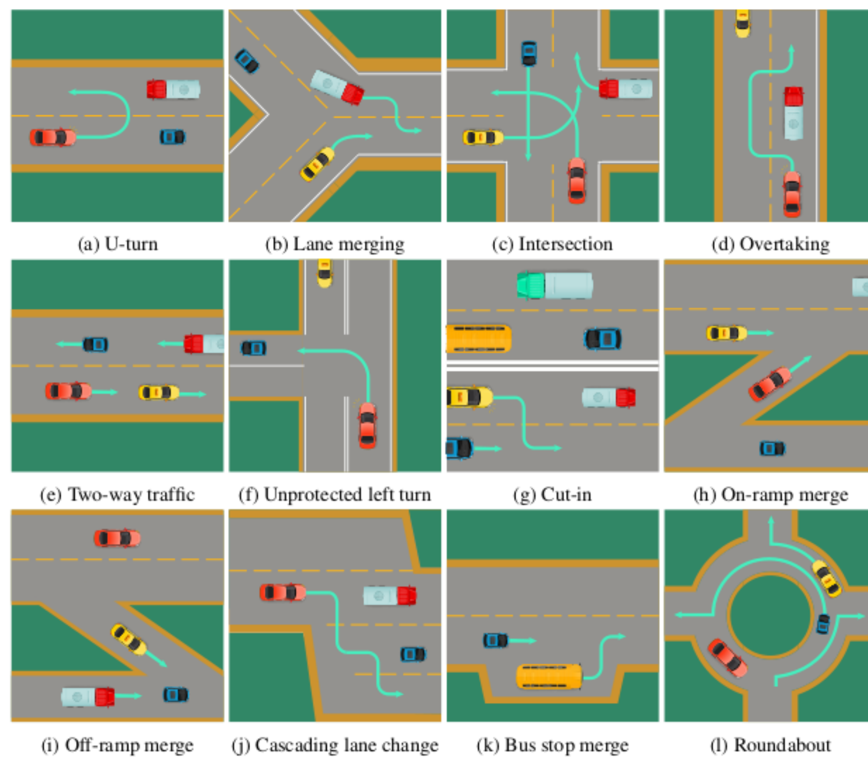
Learning methods, and in particular the Multi-Agent Reinforcement Learning paradigm, seem particularly promising to address this kind of problem, but to be investigated they need a realistic simulator to interact with. For this reason a few years ago a simulation platform called SMARTS (Scalable Multi-Agent RL Training School) has been designed. [32]

The first section of this chapter will be devoted to the description and analysis of the SMARTS simulator. Then we will precisely define which scenarios we will try to solve, specifying all the important elements of the problem. Finally we will discuss the implementation details of all the algorithms, in order to adapt them to a Multi-Agent Autonomous Driving environment. We will leave the experimental results and the comments about them for Chapter 6.

5.1 SMARTS simulator

SMARTS is a platform that allow us to develop scenarios for Multi-Agent Autonomous Driving problems. One of its biggest strengths is the possibility of designing a great number of different road scenarios. Moreover it also offers the chance of combining many of them, in order to increase the dimension of the environment. Indeed potentially we could even design city scaled maps.

An overview of the different scenarios available in SMARTS is provided in Figure 5.1.



(a) U-turn (b) Lane merging (c) Intersection (d) Overtaking
(e) Two-way traffic (f) Unprotected left turn (g) Cut-in (h) On-ramp merge
(i) Off-ramp merge (j) Cascading lane change (k) Bus stop merge (l) Roundabout

Figure 5.1: Driving interaction scenarios available in SMARTS [32]

SMARTS relies on a number of external providers in order to make the simulation as realistic as possible. The background traffic provider is SUMO [13], while physics is supported through the vehicle provider, which simulates the specified physical vehicle model and exposes a control interface at different levels of abstraction [32]. The current implementation uses the *Bullet physics engine* [4]. Finally, there is a motion plan provider which can be used to implement social agents dedicated to highly specific maneuvers, such as cut-in or U-turn.

A strong point of SMARTS is the possibility of combining "Social" vehicles[1] with Human Driven (HD) vehicles. Indeed a key aspect in the Multi-Agent Autonomous Driving research is the interaction between these two different categories of road users.

For this reason in our experiments we tested both situations in which there were just Autonomous Vehicles (AVs), and situations in which there were also HD vehicles. As we will see in Chapter 6, it makes a big difference if the HD vehicles are present or not.

5.2 Problem Definition

5.2.1 Environment

To test all the algorithms we designed a scenario consisting of a three-way junction, and we implemented four different combinations of vehicles, with increasing complexity:

- 2 Autonomous Vehicles
- 2 Autonomous Vehicles and 1 Human Driven vehicle
- 3 Autonomous Vehicles
- 3 Autonomous Vehicles and 1 Human Driven vehicle

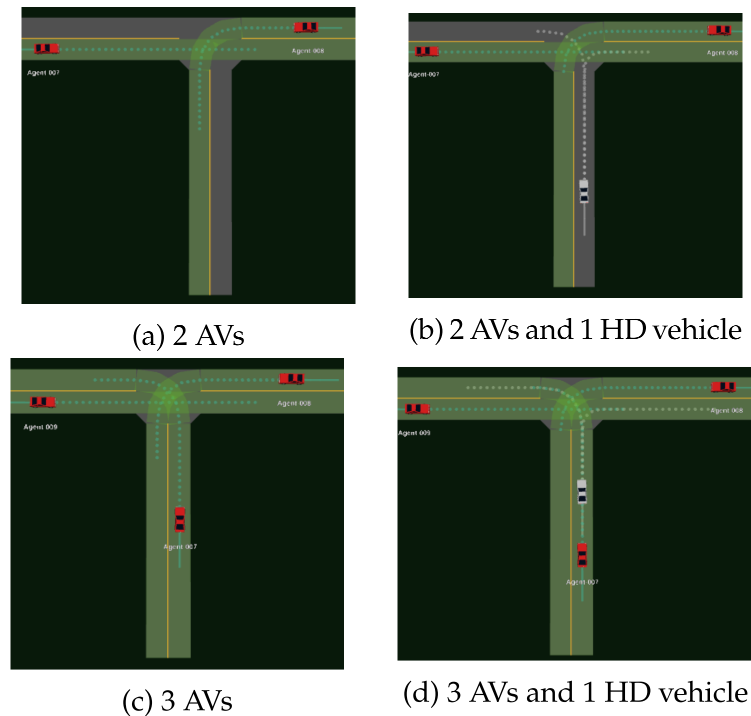
The initial configuration of the vehicles in each of the four scenarios is reported in Figure 5.2[2].

It is important to remark that, like in the majority of the Reinforcement Learning problems, each vehicle has its own goal position. However, at the beginning of the training phase the agents do not know their objectives. The purpose of the experiment is indeed to make the agents learn which are their goals and, most importantly, how to achieve them.

As we can see from Figure 5.2, in each of the four scenarios there is at least one agent that has to perform an unprotected left turn. This is because it has been seen ([8], [32]) that this is one of the most complicated maneuvers to learn for an Autonomous Vehicle, and we wanted to investigated it in our experiments.

[1]The vehicles controlled by the MARL algorithm we are testing.

[2]The red vehicles are the AVs, while the white ones are the HD vehicles.



(a) 2 AVs (b) 2 AVs and 1 HD vehicle

(c) 3 AVs (d) 3 AVs and 1 HD vehicle

Figure 5.2: Different combinations of the three-way junction scenario

Finally, the HD vehicles have been designed to behave in an aggressive way, meaning that they do not care about the presence of other vehicles, but they drive as if they were alone on the road. This approach further complicates the learning for the AVs, because they are forced to adapt to that in order to avoid collisions.

5.2.2 Observation Space

Our problem can be modelled as a Partially Observable Stochastic Game (recall Section 3.1.4), because each agent receives as observation just a subset of the current state at each time step. This is done to simulate the partial information that an Autonomous Vehicle can recover from its sensors.

With this idea in mind we tried to keep the observation space as small as possible, in contrast to previous approaches ([21], [32]), which considered a much bigger observation space for each agent.

We designed two different types of observation spaces, depending on the algorithm that we want to use. Indeed we recall that Independent Deep Q-Networks and its variants need to work with images, while the other algorithms require a flat vector of features:

- **IDQN and variants:** At each time step the observation of every agent is an image containing a stack of three consecutive frames, where each frame consists in a birds-eye view gray-scale image with the agent at the center.

  The idea of stacking consecutive frames, taken from [17], is crucial because it gives a sense of movement to the image, and that really helps understanding in which situation the agent is.

Figure 5.3: Example of one of the frames composing the observation of an agent

- **MADDPG, VDN and QMIX:** At each time step the observation of every agent is a vector containing the following information about the environment:

  1. Relative position with respect to the goal (x and y coordinates)
  2. Euclidean distance to the goal
  3. Relative position with respect to the other vehicles (x and y coordinates)
  4. Euclidean distance from the other agents
  5. Linear velocity
  6. Steering angle
  7. Heading error with respect to the optimal unknown path (the dashed lines in Figure 5.2)
  8. Distance travelled from the beginning of the episode

We note that the dimension of the observation space varies with the number of vehicles present in the simulation (11 when there are just two agents, 14 when there are three agents and 17 when there are four agents).

5.2.3 Action Space

SMARTS makes available many different possibilities to define the action space of each agent. We decided to use a discrete space, both because it has been proved itself valid in other works ([21], [32]), and because we think that a high level approach is beneficial for this kind of problem. Indeed we do not need the AVs to learn the basics of how to drive, but just how to act in conditions of traffic and where interaction with other vehicles is needed.

This is justified by the fact that there already exist successful strategies which allow AVs to have the basic driving skills, like braking, steering, and keeping the lane.

In our experiments the action space of all the agents is 4-dimensional, and at each time step they have to choose between one of the following actions:

1. *keep lane*
2. *slow down*
3. *turn left*
4. *turn right*

5.2.4 Rewards Structure

To conclude this section we analyze the rewards structure of our problem. We will have two slightly different structures, depending on the algorithm that we are using. This is needed because VDN and QMIX can be applied just to cooperative games, while in our standard definition of the problem we consider the game to be general-sum.

Starting from the latter case, at each time step every agent i receives a reward signal composed in this way:

1. A default reward given by the SMARTS environment, which is a function of the distance travelled:
$$r_i(x) = \begin{cases} x, & \text{if } ||x|| > 0.5 \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$
Where x is the distance travelled in meters from the last time step where a non-zero reward was given.

2. The reward obtained at the previous point is then modified if some events have happened during the current time step. If the agent has driven off road, on road shoulder, on wrong way, off route or if it has collided, then the reward is reduced by 10. Otherwise if the agent has reached its goal the reward is increased by 10.

If we are testing the VDN or QMIX algorithms, instead of having a different reward for each agent, we need a global reward common to all the agents.

To do that the most intuitive and straightforward way is to sum together the rewards of all the agents, and consider the result as the global reward of the environment. This will be our strategy when testing those two algorithms.

5.3 Implementation Details of the Algorithms

In this section we provide a detailed description of how we implemented the different algorithms and the choices that we made for their hyperparameters.

5.3.1 IDQN and Variants

- **Replay buffer:** we keep a different replay buffer for every agent i, and at each time step we store the transition data $(o_i^t, a_i^t, r_i^t, o_i^{t+1})$. The size of each buffer is of 20000 elements.
- **Networks architecture:** we have two slightly different network structures, depending on the choice that we make between standard DQN and Dueling DQN:
    1. Standard DQN: the first hidden layer convolves 32 filters of dimension 8×8 with stride 4, the second one convolves 64 filters of dimension 4×4 with stride 2 and the last convolutional layer uses 64 filters of dimension 3×3 with stride 1. Then we have a fully connected hidden layer, consisting of 512 units. Finally the output layer is fully connected, with a single output for each action. All the layers use the ReLU activation function.
    2. Dueling DQN: the structure of the network is exactly like before, except for the output layer, which is now split in two different layers, one with just one output to approximate the state value function, and the other one with an output for each action, to approximate the advantage function.
- **Learning phase:** we perform learning after each time step, and we try to optimise the Mean Squared Error loss using the *RMSprop* optimizer [25], with a learning rate equal to 10^{-4}. To compute the loss we sample batches of 32 elements and we use a discount factor γ equal to 0.99. Finally we use

an ϵ-greedy strategy for action selection, with ϵ linearly decreasing from 1 to 0.01 through 200000 time steps. The target networks are updated after every 1000 time steps.

5.3.2 MADDPG

- **Replay buffer:** for this algorithm we have both a global critic buffer and an actor buffer for each agent. The critic buffer stores at each time step the tuple (s_t, r_t, s_{t+1}), where s is the concatenation of the observations of all the agents, and r is the rewards list.
  Each agent memory instead stores the transition $(o_i^t, a_i^t, o_i^{t+1})$ at each time step. All the buffers have a size of 1000000 samples[3].

- **Networks architecture:** both the critic and actor networks are built using two fully connected hidden layers, with 64 rectified linear units for each layer. The difference between the two is in the output layer: the latter uses a fully connected layer with *tanh* activation function and one output corresponding to the chosen action. The former instead exploits a fully connected layer with one output corresponding to the action value function, but this time there is no activation function.

- **Learning phase:** we update the networks parameters after every 100 time steps. We use the Mean Squared Error loss for the critic, computed by sampling batches of 1024 transitions, with discount factor $\gamma = 0.95$. All the networks are updated using the *Adam* optimizer [12], but we use two different learning rates for the actor and critic networks: $\alpha = 10^{-4}$ for the former and $\beta = 10^{-3}$ for the latter.
  To explore the environment we modify each action selected by the agent by adding white Gaussian noise. Finally the parameters of the target networks are softly updated every 100 steps with $\tau = 0.01$.

5.3.3 VDN and QMIX

- **Replay buffer:** for both the algorithms we keep a global replay buffer which stores transition data $([o_1^t, \ldots, o_n^t], [a_1^t, \ldots, a_n^t], [r_1^t, \ldots, r_n^t], [o_1^{t+1}, \ldots, o_n^{t+1}])$ for all the n agents. The capacity of the replay buffer is of 5000 samples.

- **Networks structure:** the networks of the utility functions are the same both for VDN and QMIX. What changes is of course the centralized action value function.

[3] Note that the buffers are much bigger with respect to IDQN because before we were working with images, while now we are considering low dimensional vectors.

As said in Section 4.5, we use Recurrent Neural Networks for the utility functions: in particular each network is composed by three layers; the first is a fully connected layer, composed by 64 rectified linear units, the second one is a GRU cell again of dimension 64, and the output layer is fully connected with no activation function and a different output for each available action.

For what regards the centralized action value function with VDN we just sum all the utility functions. For QMIX instead we use a mixing network consisting of a single hidden layer of 32 units, with ELU activation function. The hypernetworks are then sized to produce weights of appropriate size. The hypernetwork producing the final bias of the mixing network consists of a single hidden layer of 32 units with ReLU activation function.

- **Learning phase:** we perform learning after each time step, and we try to optimise the Mean Squared Error loss using the *Adam* optimizer, with a learning rate equal to $5 \cdot 10^{-4}$. To compute the loss we sample batches of 32 elements and we use a discount factor γ equal to 0.99. Finally we use an ϵ-greedy strategy for action selection, with ϵ linearly decreasing from 1 to 0.05 through 50000 time steps. The target networks are softly updated after each time step, with $\tau = 0.005$.

6

Results and Comments

In this chapter we will show the experimental results obtained by testing all the presented algorithms in the Multi-Agent Autonomous Driving environment described in Chapter 5.

Before doing that we describe the training and evaluation procedures, and the metrics we used to evaluate the performances.

First of all we remark that any scenario is considered solved if all the agents correctly pass through the junction and at least one of them reaches its goal position. This is because the SMARTS simulator ends the episode as soon as one agent reaches its goal. This is not a problem for us because if that happens, it means that all the agents have successfully gone through the junction, and they are very close to their goal position. In terms of rewards,the environments with 2 Autonomous Vehicles are considered solved when we achieve a score of 100 or higher, while in the environments with 3 AVs we have to achieve a score of at least 120 in order to solve them.

The other events that could terminate an episode are the collision of two or more agents and the achievement of the maximum number of available time steps in an episode[1]. Clearly in both this situations the environment cannot be considered solved.

To analyze the results we will consider the two main metrics used in literature to evaluate MARL algorithms, as detailed in [1]. The first metric is the *learning*

[1]This number varies between 50 and 75, depending on the specific scenario we are considering.

curve, a two dimensional plot where the x-axis represents the training steps, and the y-axis shows an estimation of the agents' episodic returns. To collect this data for each algorithm we adopt the following procedure:

- Train the algorithm for 500-thousand time steps[2]
- Pause training after every 5000 time steps to *evaluate*[3] the algorithm
- Store the mean of the episodic returns obtained during evaluation
- Resume training

The second metric we consider is the maximum value of the learning curve. This is clearly a condensed and less informative metric with respect to before, but it can still be useful to understand if at some point the algorithm solved the environment.

Finally, to help us compare all the algorithms in the different scenarios, we will answer the following three questions:

1. Did the algorithm solve the environment at any point?
2. Did the learning curve asymptotically converge to a result which solved the environment?
3. If the algorithm converged, how fast was it? (i.e. how many steps did it need to reach the steady state?)

[2]Except for IDQN and its variants, which were trained for 300-thousand time steps, due to the huge number of computations and time required to handle images.

[3]When we evaluate an algorithm we do not perform learning.

6.1 Results IDQN and Variants

6.1.1 Independent Deep Q-Networks

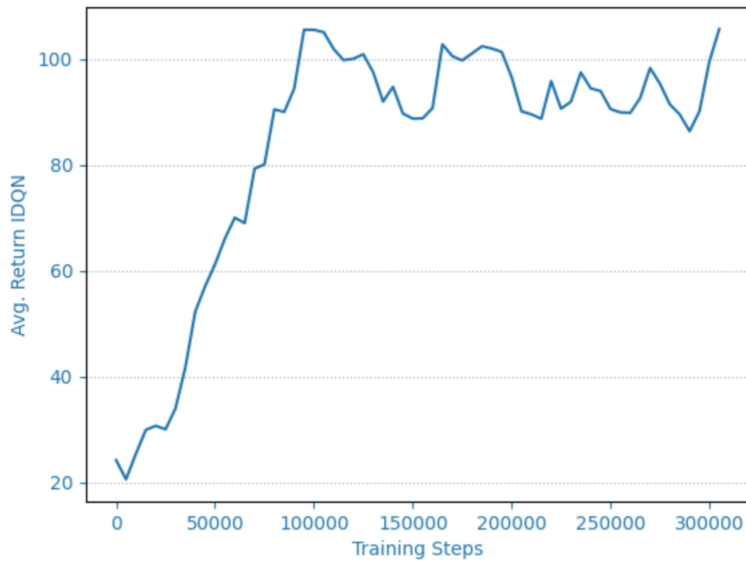


Figure 6.1: IDQN 2 AVs scenario

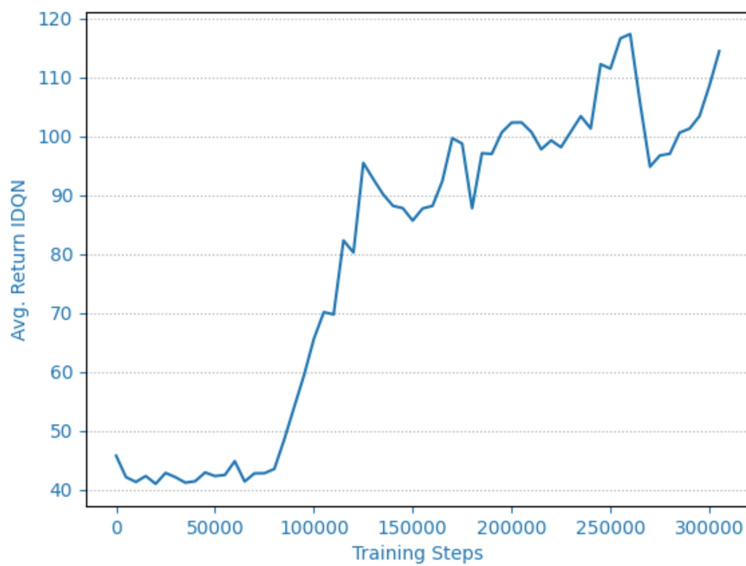


Figure 6.2: IDQN 2 AVs and 1 HD vehicle scenario

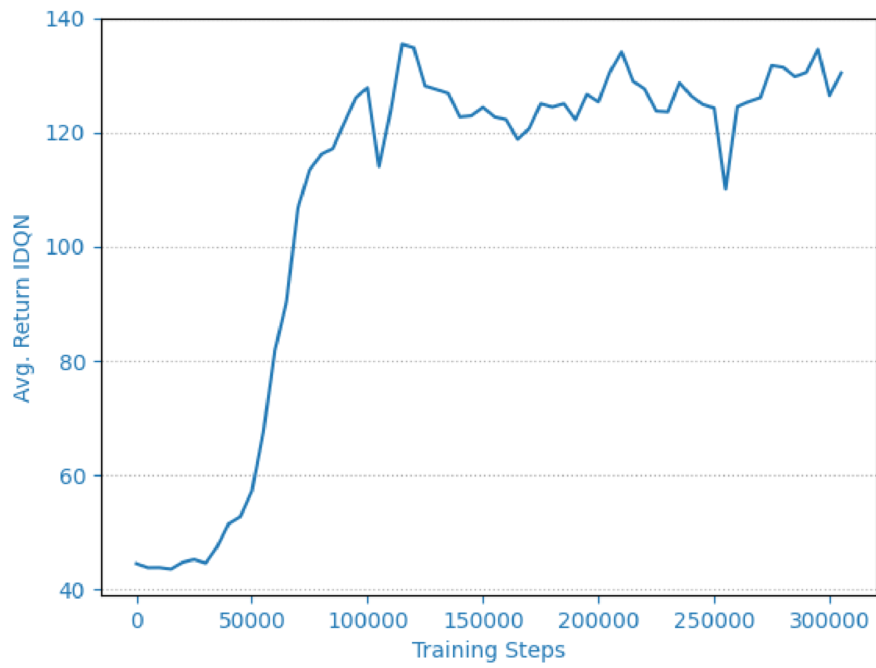


Figure 6.3: IDQN 3 AVs scenario

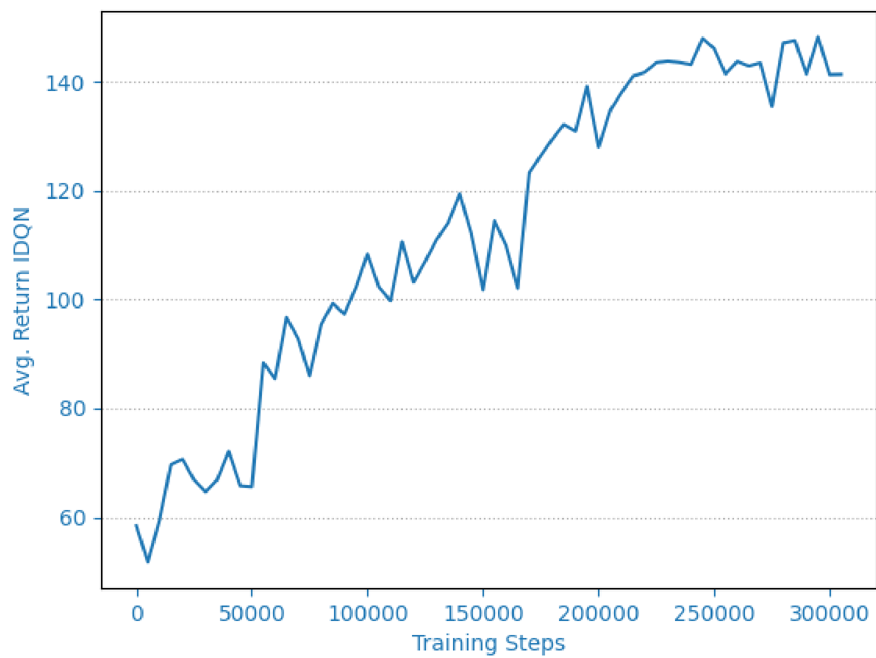


Figure 6.4: IDQN 3 AVs and 1 HD vehicle scenario

6.1.2 Independent Double Deep Q-Networks

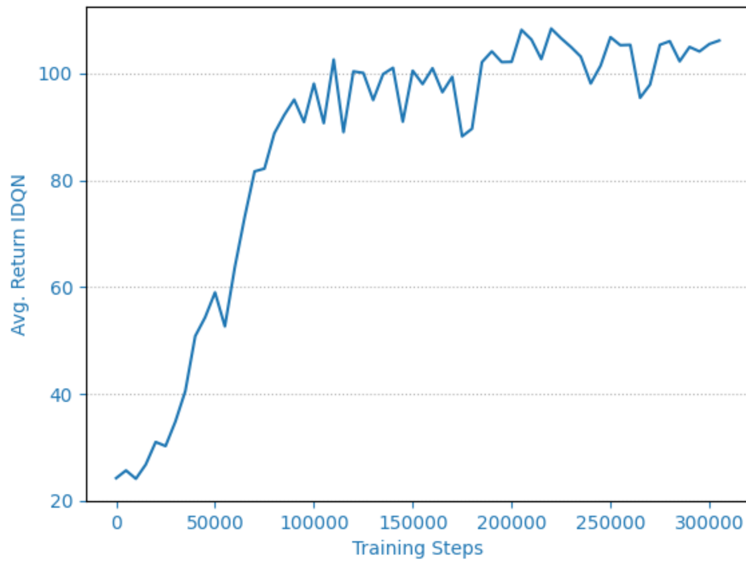


Figure 6.5: IDDQN 2 AVs scenario

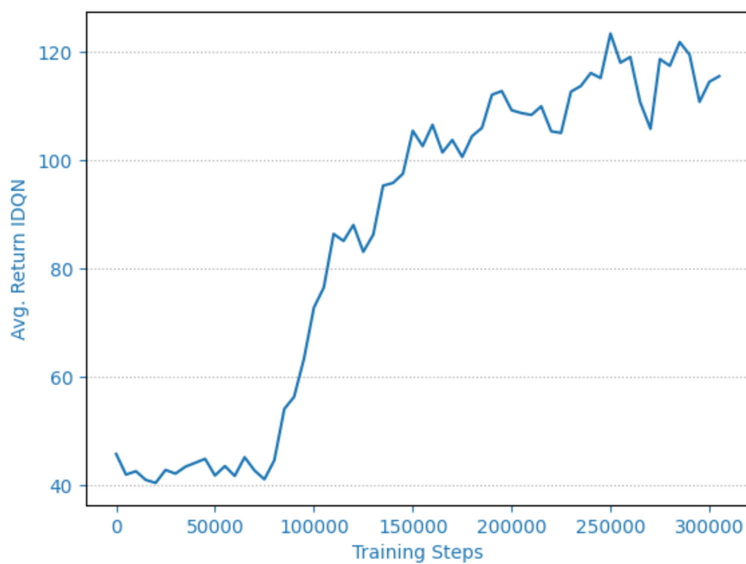


Figure 6.6: IDDQN 2 AVs and 1 HD vehicle scenario

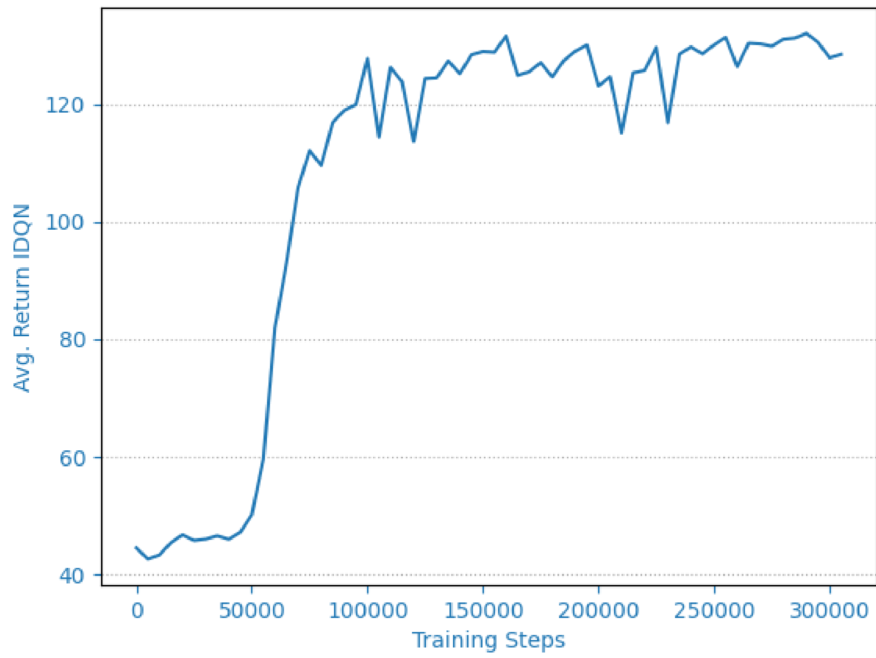


Figure 6.7: IDDQN 3 AVs scenario

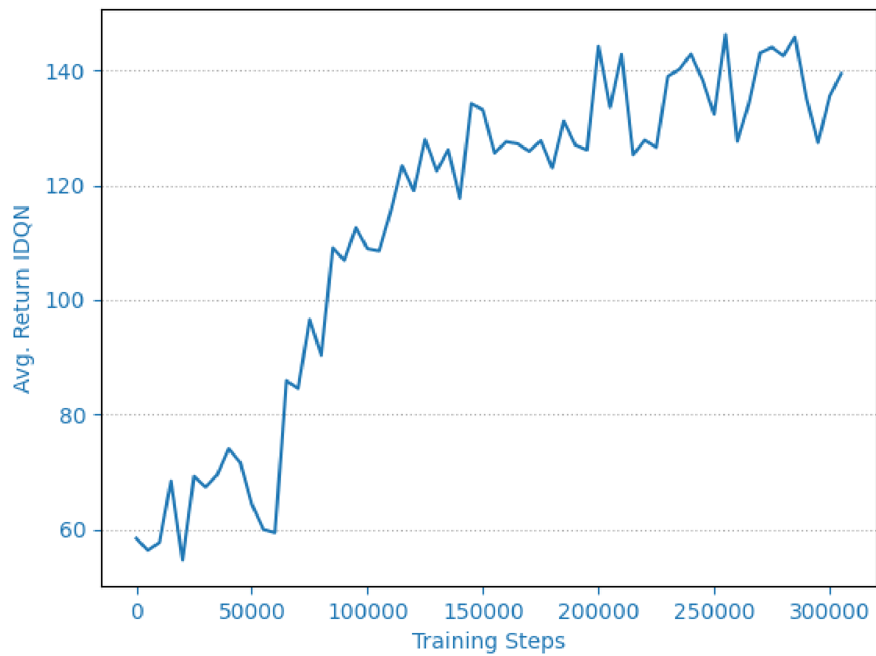


Figure 6.8: IDDQN 3 AVs and 1 HD vehicle scenario

6.1.3 Independent Dueling Deep Q-Networks

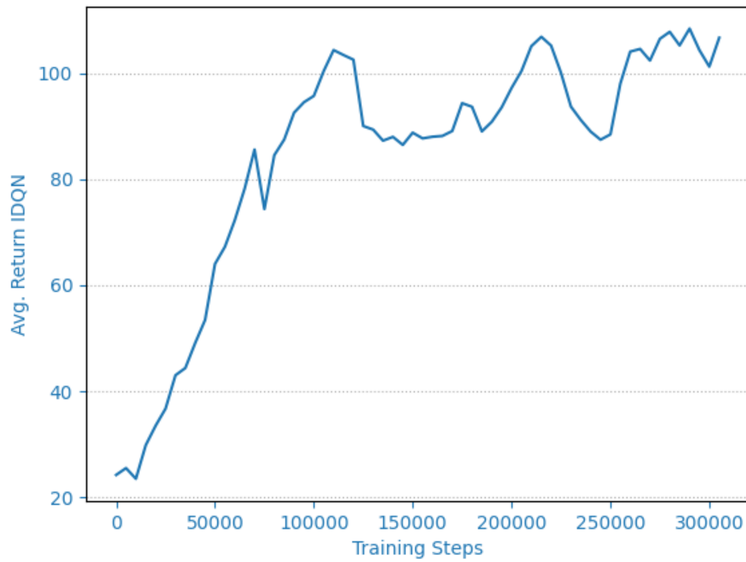


Figure 6.9: Independent Dueling DQN 2 AVs scenario

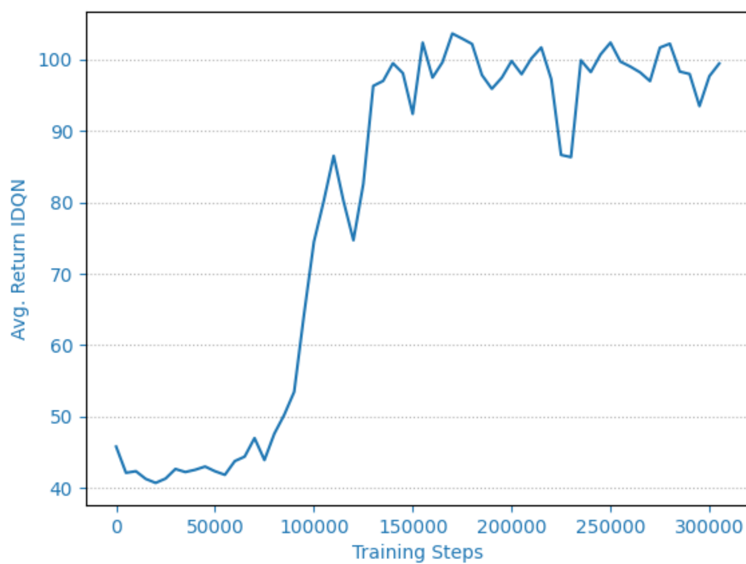


Figure 6.10: Independent Dueling DQN 2 AVs and 1 HD vehicle scenario

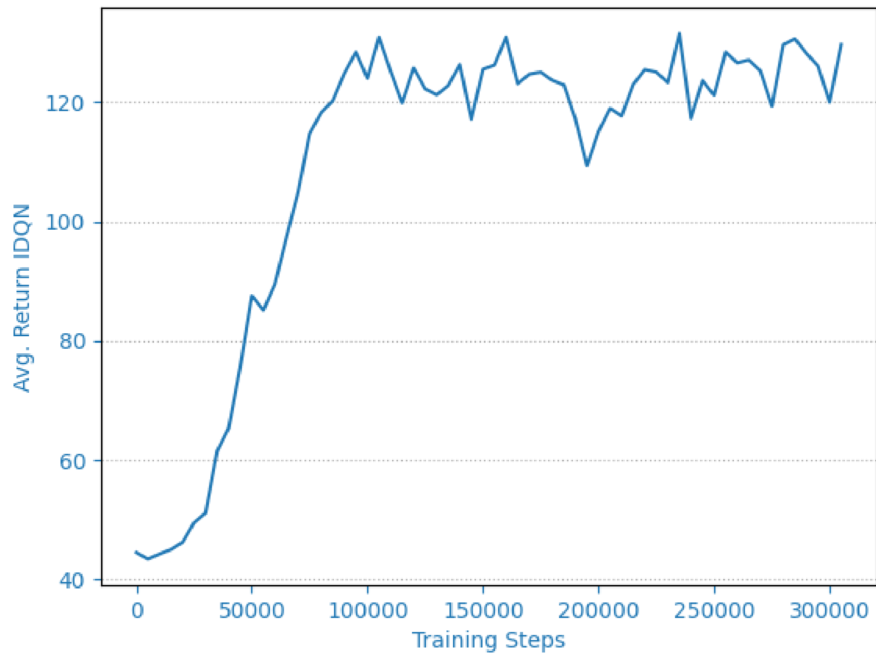


Figure 6.11: Independent Dueling DQN 3 AVs scenario

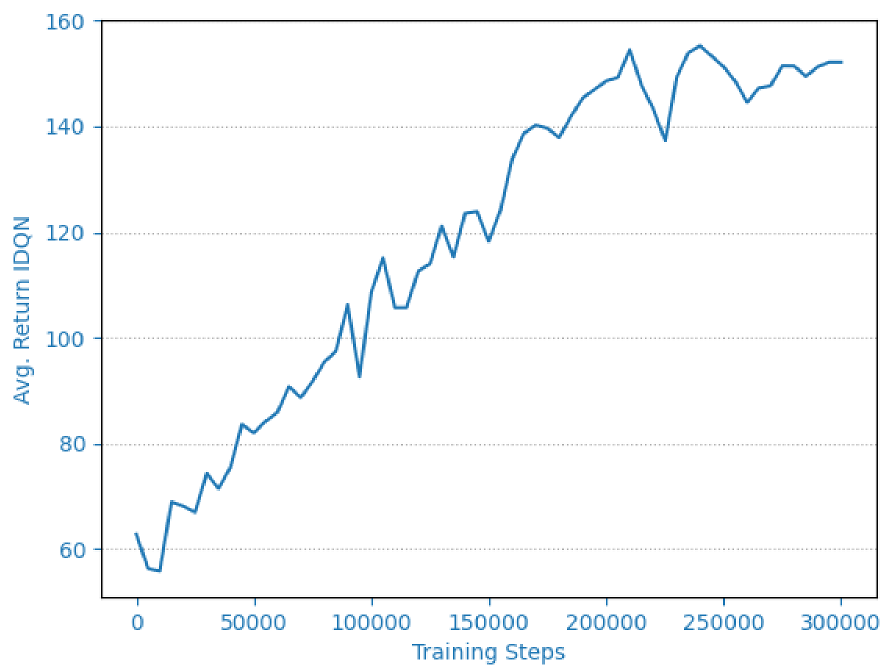


Figure 6.12: Independent Dueling DQN 3 AVs and 1 HD vehicle scenario

6.1.4 Independent Dueling Double Deep Q-Networks

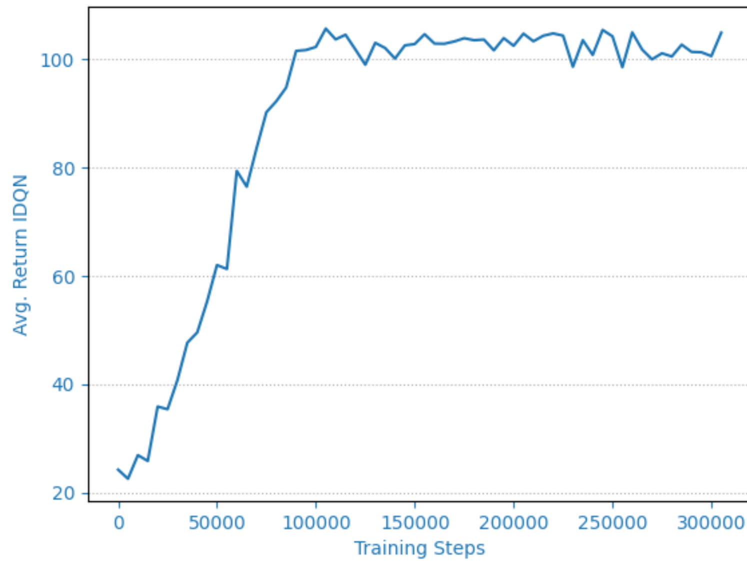


Figure 6.13: Independent Dueling DDQN 2 AVs scenario

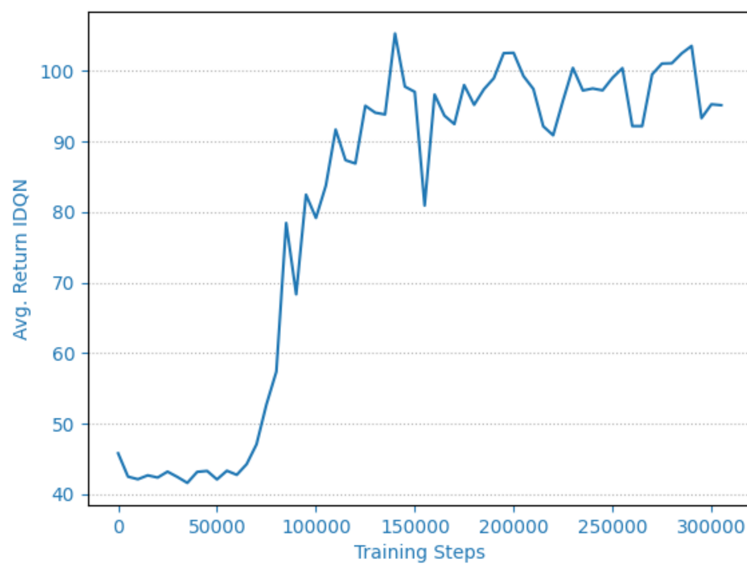


Figure 6.14: Independent Dueling DDQN 2 AVs and 1 HD vehicle scenario

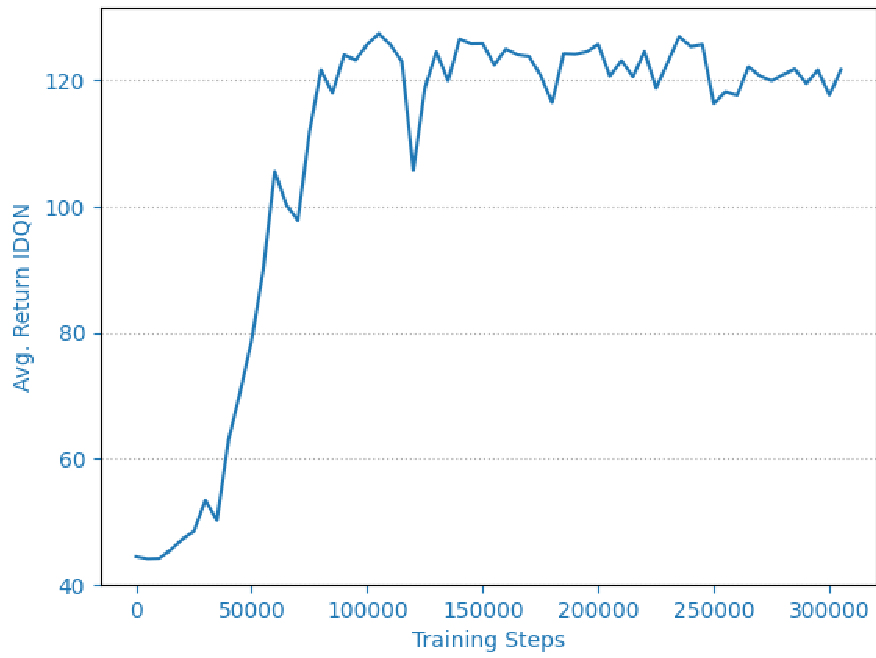


Figure 6.15: Independent Dueling DDQN 3 AVs scenario

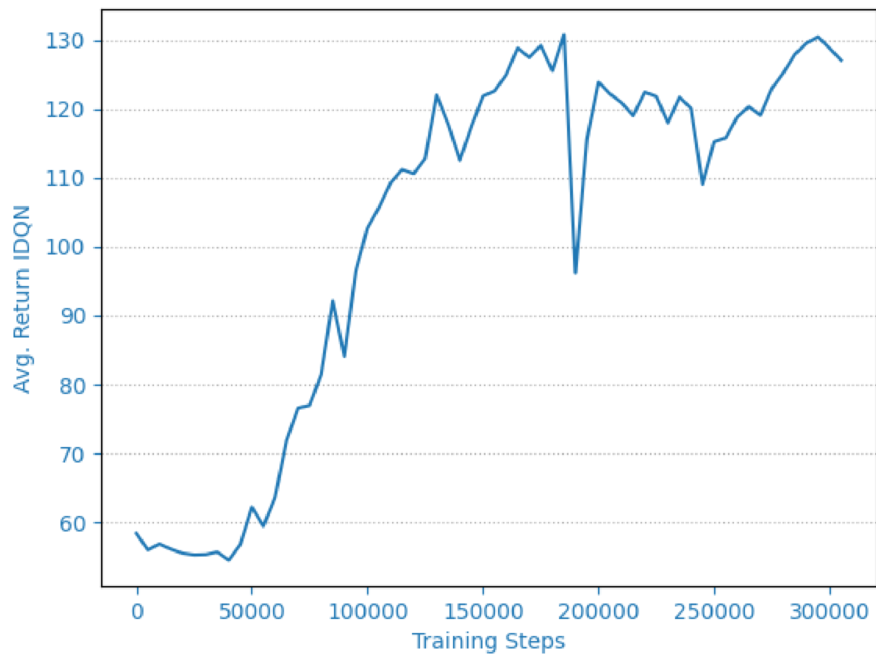


Figure 6.16: Independent Dueling DDQN 3 AVs and 1 HD vehicle scenario

6.2 Comments IDQN and Variants

Looking at the plots the first thing to note is that the performances of IDQN and its variants are very good in all the four scenarios. Indeed each environment is asymptotically solved with all the algorithms, with some differences that we will now discuss.

As expected the Double network strategy and the Dueling architecture generally improve the performances. The Double network is useful to improve stability during learning, while the Dueling architecture can make the learning process faster, reducing the time needed to reach convergence. These things are visible for example in the 2 Autonomous Vehicles scenario (Figures 6.1, 6.5, 6.9 and 6.13).

Another interesting observation is that in scenarios with human driven vehicles, the Dueling architecture sometimes achieves a lower steady state score with respect to the standard DQN network, despite its higher speed in reaching convergence. This is clearly visible in the 2 AVs plus HD vehicle environment (Figures 6.6 and 6.14). The reasons for that could be multiple, but the main issue is that the presence of HD vehicles exacerbates the non-stationarity of the environment, thus complicating a lot the job of the advantage function present in the Dueling architecture. Indeed now the advantage of taking a particular action in a certain state for an agent can drastically change in different episodes, depending not only on the other Autonomous Vehicles, but also on the human driven one.

Finally, we must observe that the presence of HD vehicles makes the learning process much harder for all the algorithms. Clearly this behaviour was expected, however it is very interesting, even if not surprising, that a Decentralized Training and Execution algorithm like IDQN is capable of handling so well these kind of environments, without considering any type of communication between the agents.

6.3 Results MADDPG

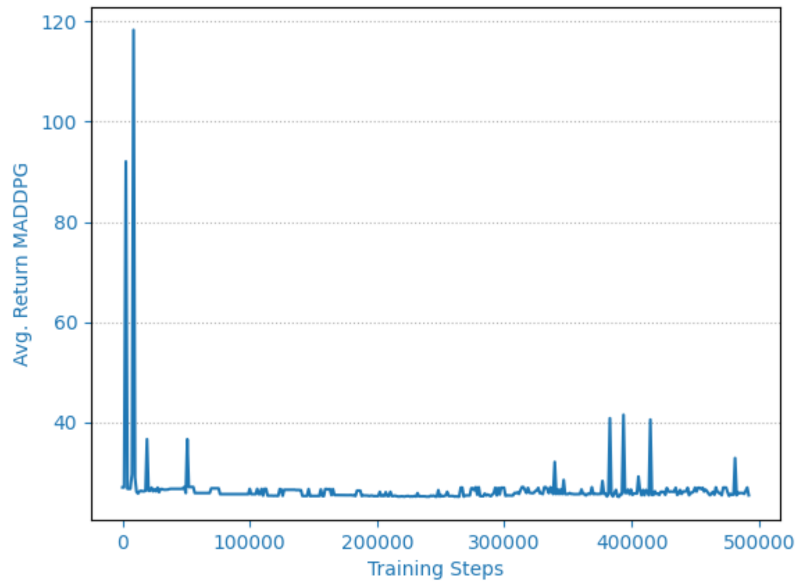


Figure 6.17: MADDPG 2 AVs scenario

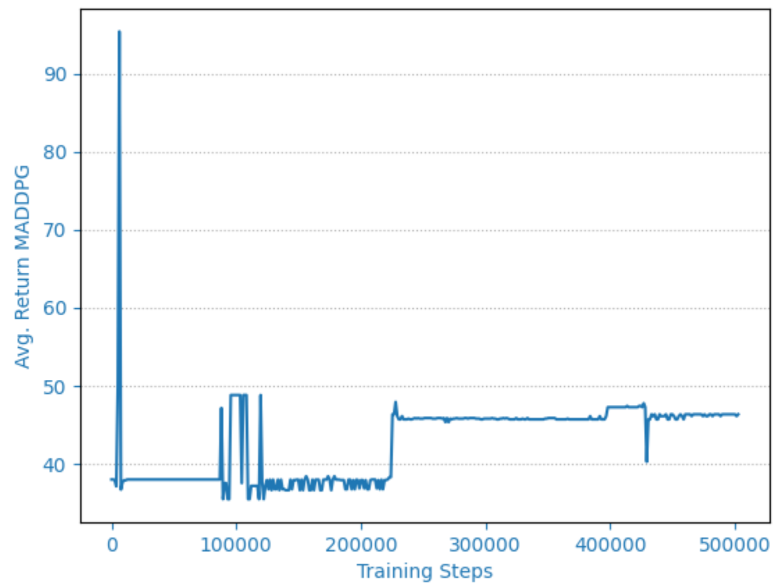


Figure 6.18: MADDPG 2 AVs and 1 HD vehicle scenario

6.4 Comments MADDPG

As clearly visible from Figures 6.17 and 6.18, MADDPG performances are a lot worse than those of IDQN. Indeed now we are not even close to asymptotic convergence, even if in both the graphs we can see that, at a very early stage of the learning process, the algorithm was capable of finding a set of networks parameters which solved the environments.

Regarding the 3 AVs and the 3 AVs plus HD vehicle scenarios, we do not show the learning curves because during all the training procedure there is no sign of learning; the environments are simply too complex to be solved with this algorithm.

The reasons behind these bad performances are multiple. First of all we must note that MADDPG is one of the first policy-based MARL algorithms, and it suffers from all the downsides of policy-based techniques, like high variance when evaluating a policy and especially attitude to converge to local rather than global optimum.

Moreover this algorithm was originally developed for environments with continuous action spaces [18], while we recall that in our particular problem the action space is discrete. Despite the modifications that we made to the algorithm in order to adapt it to the SMARTS environment, we believe that this is also one of the reasons for such bad performances.

6.5 Results VDN and QMIX

6.5.1 VDN

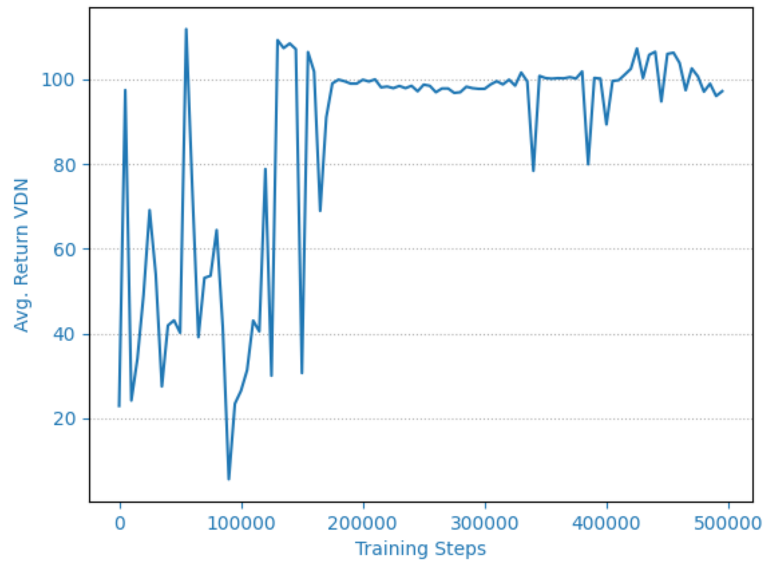


Figure 6.19: VDN 2 AVs scenario

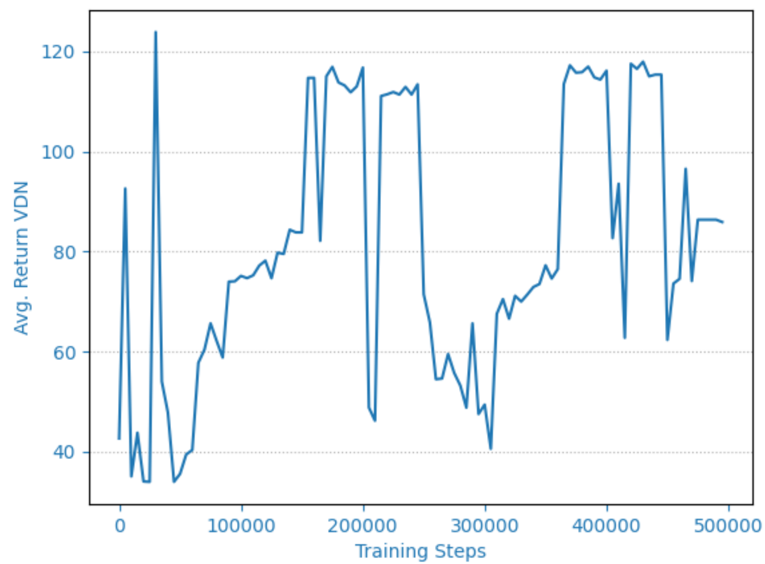


Figure 6.20: VDN 2 AVs and 1 HD vehicle scenario

6.5.2 QMIX

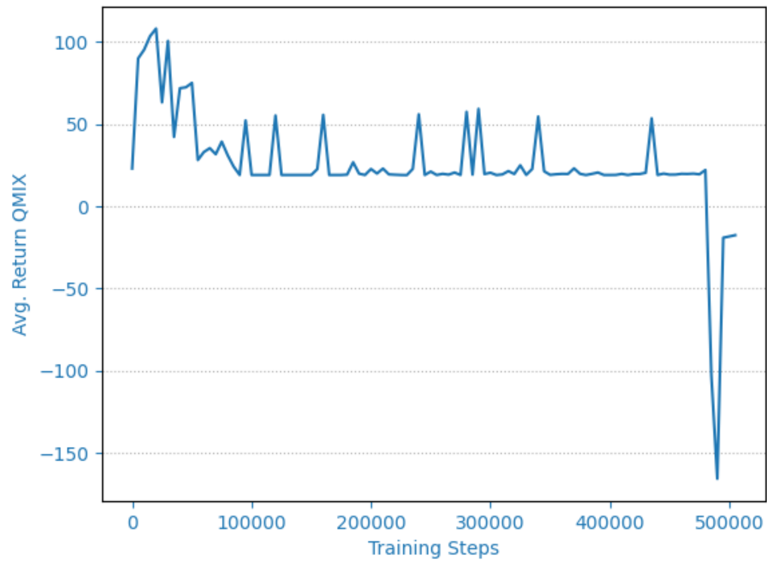


Figure 6.21: QMIX 2 AVs scenario

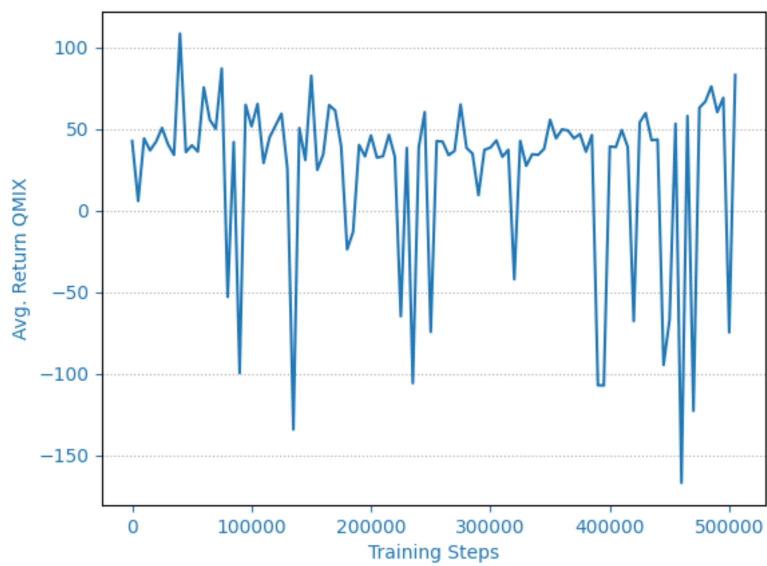


Figure 6.22: QMIX 2 AVs and 1 HD vehicle scenario

6.6 Comments VDN and QMIX

The first thing that we remark is that, like in the case of MADDPG, the plots with the results of the scenarios with 3 Autonomous Vehicles are not reported, because there were no sign of learning throughout the whole training processes.

This is mainly due to the nature of value decomposition algorithms like VDN and QMIX. As already said, these algorithms are most suited for fully cooperative problems, while our testing scenario is originally general-sum, even if we modified its reward structure to consider a global reward, as explained in Section 5.2.4.

Moving to the scenarios with 2 AVs, we immediately notice the big difference between VDN and QMIX. Indeed the former has extremely good performances with just 2 autonomous agents, while the latter performs quite poorly even in this simplest scenario. This behaviour is interesting, because QMIX is a sort of upgrade of VDN, and theoretically it should perform better. This does not happen in our problem because the higher complexity of QMIX makes the hyperparameters tuning process much harder, and it also gives rise to a very big number of parameters to optimize. VDN instead, despite its simpler mixing strategy, is able to solve the environment and also achieve asymptotic convergence.

Finally, in the 2 AVs plus HD vehicle scenario VDN does not achieve asymptotic convergence, even if it shows interesting results, obtaining very high scores many times throughout the training process.

6.7 Overall Comparison

	Environment solved	Asymptotic convergence	Speed of convergence
IDQN	✓	✓	100k
IDDQN	✓	✓	110k
IDuelDQN	✓	✓	100k
IDuelDDQN	✓	✓	90k
MADDPG	✓	×	/
VDN	✓	✓	180k
QMIX	✓	×	/

Table 6.1: 2 AVs scenario

	Environment solved	Asymptotic convergence	Speed of convergence
IDQN	✓	✓	160k
IDDQN	✓	✓	170k
IDuelDQN	✓	✓	140k
IDuelDDQN	✓	✓	140k
MADDPG	✓	×	/
VDN	✓	×	/
QMIX	✓	×	/

Table 6.2: 2 AVs and 1 HD vehicle scenario

	Environment solved	Asymptotic convergence	Speed of convergence
IDQN	✓	✓	100k
IDDQN	✓	✓	100k
IDuelDQN	✓	✓	80k
IDuelDDQN	✓	✓	90k
MADDPG	×	×	/
VDN	×	×	/
QMIX	×	×	/

Table 6.3: 3 AVs scenario

	Environment solved	Asymptotic convergence	Speed of convergence
IDQN	✓	✓	210k
IDDQN	✓	✓	200k
IDuelDQN	✓	✓	200k
IDuelDDQN	✓	✓	150k
MADDPG	×	×	/
VDN	×	×	/
QMIX	×	×	/

Table 6.4: 3 AVs and 1 HD vehicle scenario

In all scenarios, IDQN and its variants outperform the other algorithms. This is clearly visible in the tables above, where all the results of the different algorithms in the 4 scenarios have been compared, making use of the evaluation metrics introduced at the beginning of Chapter 6. We can see that, except for IDQN, only VDN is capable of reaching asymptotic convergence, but just in the 2 AVs environment. MADDPG and QMIX are instead able, at a certain point during training, to find a set of network parameters which solve the 2 AVs and 2 AVs plus HD vehicle scenarios, but they both are far from convergence and quite unstable. Finally, the environments with 3 AVs are asymptotically solved by the IDQN family of algorithms, while the other strategies completely fail.

We believe that the reasons for this huge performance gap are mainly two: the first aspect concerns the conditioning of the policies on the history of observations. Indeed there are three different options to do that [1], and the algorithms that we tested represented all these categories:

- IDQN and its variants stacked three consecutive frames in each observation.
- VDN and QMIX used a recurrent neural network structure.
- MADDPG assumed that the current observation carried all necessary information for deciding an action.

After seeing the results we can affirm that in a problem like ours it is crucial to consider the history of observations at each time step.

The second reason that explains these results regards the structure of our problem. As already said, we considered a general-sum game, where the agents did not compete nor explicitly cooperate between them. These kind of problems are extremely difficult to handle for a MARL algorithm, and many times resorting to an independent learning strategy can be the best choice. The situation would change if we consider a cooperative problem; in that case CTDE

algorithms would become of great help. A possible application of that will be briefly discussed in Section 7.1.

In conclusion, we remark that the obtained results are consistent with what we saw in Section 1.1: at the moment for Autonomous Driving applications in mixed traffic independent learning methods are often the best solution.

Conclusions and Future Works

In this work we encountered and discussed an important number of different topics. Indeed the complexity of the Multi-Agent Reinforcement Learning field required a deep analysis of its foundations and core concepts, before diving into the specific Autonomous Driving application objective of this thesis.

We began by introducing the problem and analyzing some related works, to see the different areas of research and the current limitations of this field. Then we introduced the single agent Reinforcement Learning paradigm, first from the theoretical point of view, and then from the practical implementation perspective. Already in this case we saw the necessity, and at the same time the complexity of introducing non-linear approximations, through Neural Networks, in high-dimensional sequential decision processes. We also described the two macro-categories of RL algorithms: values-based and policy based strategies.

After that we gave a very brief overview of Game Theory, which gave us the theoretical foundations and definitions of a Multi-Agent problem. We first introduced the different types of game models, and then we defined the different solution concepts that we can exploit in order to solve them. Finally, we talked about tabular Multi-Agent Reinforcement Learning, and we saw its huge limitations when facing complex problems.

In Chapter 4 we entered the core part of the thesis, where we analyzed the challenges of Deep-MARL and also the three different paradigms under which the algorithms are classified: Centralized Training and Execution (CTCE), Decentralized Training and Execution (DTDE), Centralized Training and Decen-

tralized Execution (CTDE). Then we deeply analyzed the four algorithms that we decided to use for our experiments: IDQN and its variants, MADDPG, VDN and QMIX.

Finally, Chapters 5 and 6 were devoted to the experimental part of the thesis. At first we introduced the simulator that we decided to use for our experiments. After that we described the environments that we specifically designed for this project, along with the observation and action spaces of the agents, and their reward structure. We also discussed about the practical implementations of the algorithms, especially regarding the choice of the hyperparameters.

After that we were ready to finally show the experimental results of all the four algorithms. We began by analyzing them separately, and later on we compared them all together. We clearly saw that IDQN outperformed the other algorithms in all the scenarios, confirming the competitiveness of independent learning methods, especially in general-sum problems like this one.

7.1 Possible future directions

The major challenges of Multi-Agent Reinforcement Learning, and in particular the non-stationarity of the environment and the dimensions of the state and action spaces, are the reasons for which MARL algorithms require a tremendous number of samples for effective training [30].

A promising approach to address this issue consists in developing model-based strategies for MARL problems. This idea arises from the fact that model-based methods in single agent RL have shown their advantages in sample efficiency both practically and theoretically [30]. However, simply applying single agent model-based algorithms to the MARL setting would not work, due to the challenges mentioned above and discussed in Section 4.1.

This reason, along with the fact that the studies about model-based MARL strategies have just started very recently, makes this field of research extremely interesting and promising. Moreover it has also been shown that learning and then exploiting a model could be also useful to explore the environment, and thus improve the performances [5], [21].

Finally, model-based strategies could be a crucial tool to help predicting the behavior of human drivers.

Another interesting theme regarding smart mobility and traffic scenarios con-

cerns the deployment of a fleet of Autonomous Vehicles in a specific stretch of road, with the purpose of improving the traffic flow. This is particularly useful in highways, where it has been shown [11] that these vehicles could help the other road users to respect the security distances, thus avoiding traffic congestion and also many incidents. This problem would be cast as a completely cooperative task, and for this reason many of the techniques explored in this thesis could become useful, along with communication techniques between agents and again model-based approaches.

Finally it would be interesting to transpose these experiments in the physical world, to see the effectiveness of all this techniques outside of simulation, and also to handle the natural differences between virtual and real scenarios. Currently there exists a platform, called Duckietown [19], which seems particularly suitable to test all the proposed strategies in a scaled real world environment.

References

[1] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2023. URL: https://www.marl-book.com.

[2] Xi Chen and Xiaotie Deng. "Settling the Complexity of Two-Player Nash Equilibrium." In: *FOCS*. Vol. 6. 2006, pp. 261–272.

[3] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE].

[4] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. http://pybullet.org. 2016–2021.

[5] Sebastian Curi, Felix Berkenkamp, and Andreas Krause. "Efficient model-based reinforcement learning through optimistic policy search and planning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 14156–14170.

[6] Joris Dinneweth et al. "Multi-agent reinforcement learning for autonomous vehicles: A survey". In: *Autonomous Intelligent Systems* 2.1 (2022), p. 27.

[7] Jiqian Dong et al. "A DRL-based multiagent cooperative control framework for CAV networks: A graphic convolution Q network". In: *arXiv preprint arXiv:2010.05437* (2020).

[8] Mohamed Elsayed et al. *ULTRA: A reinforcement learning generalization benchmark for autonomous driving.* 2020. URL: https://ml4ad.github.io/files/papers2020/ULTRA:%20A%20reinforcement%20learning%20generalization%20benchmark%20for%20autonomous%20driving.pdf.

[9] David Ha, Andrew M Dai, and Quoc V Le. "Hypernetworks, in 5th International Conference on Learning Representations, ICLR 2017". In: *Conference Track Proceedings, OpenReview. net*. 2017, pp. 24–26.

[10] Songyang Han et al. "Stable and efficient Shapley value-based reward reallocation for multi-agent reinforcement learning of autonomous vehicles". In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, pp. 8765–8771.

[11] Nicholas Hyldmar, Yijun He, and Amanda Prorok. "A fleet of miniature cars for experiments in cooperative driving". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 3238–3244.

[12] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[13] Daniel Krajzewicz et al. "SUMO (Simulation of Urban MObility)-an open-source traffic simulation". In: *Proceedings of the 4th middle East Symposium on Simulation and Modelling (MESM20002)*. 2002, pp. 183–187.

[14] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[15] Ryan Lowe et al. "Multi-agent actor-critic for mixed cooperative-competitive environments". In: *Advances in neural information processing systems* 30 (2017).

[16] Alexey A Melnikov, Adi Makmal, and Hans J Briegel. "Projective simulation applied to the grid-world and the mountain-car problem". In: *arXiv preprint arXiv:1405.5459* (2014).

[17] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[18] Igor Mordatch and Pieter Abbeel. "Emergence of grounded compositional language in multi-agent populations". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[19] Liam Paull et al. "Duckietown: an open, inexpensive and flexible platform for autonomy education and research". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 1497–1504.

[20] Tabish Rashid et al. “Monotonic value function factorisation for deep multi-agent reinforcement learning”. In: *The Journal of Machine Learning Research* 21.1 (2020), pp. 7234–7284.

[21] Pier Giuseppe Sessa, Maryam Kamgarpour, and Andreas Krause. “Efficient Model-based Multi-agent Reinforcement Learning via Optimistic Equilibrium Computation”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 19580–19597.

[22] Lloyd S Shapley. “Stochastic games”. In: *Proceedings of the national academy of sciences* 39.10 (1953), pp. 1095–1100.

[23] Peter Sunehag et al. “Value-decomposition networks for cooperative multi-agent learning”. In: *arXiv preprint arXiv:1706.05296* (2017).

[24] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[25] Tijmen Tieleman, Geoffrey Hinton, et al. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[26] Behrad Toghi et al. “Altruistic maneuver planning for cooperative autonomous vehicles using multi-agent advantage actor-critic”. In: *arXiv preprint arXiv:2107.05664* (2021).

[27] Behrad Toghi et al. “Cooperative autonomous vehicles that sympathize with human drivers”. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2021, pp. 4517–4524.

[28] Behrad Toghi et al. “Social coordination and altruism in autonomous driving”. In: *IEEE Transactions on Intelligent Transportation Systems* 23.12 (2022), pp. 24791–24804.

[29] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[30] Xihuai Wang, Zhicheng Zhang, and Weinan Zhang. “Model-based multi-agent reinforcement learning: Recent progress and prospects”. In: *arXiv preprint arXiv:2203.10603* (2022).

[31] Ziyu Wang et al. “Dueling network architectures for deep reinforcement learning”. In: *International conference on machine learning*. PMLR. 2016, pp. 1995–2003.

[32] Ming Zhou et al. “Smarts: Scalable multi-agent reinforcement learning training school for autonomous driving”. In: *arXiv preprint arXiv:2010.09776* (2020).

Acknowledgments

First of all I would like to sincerely thank my supervisor, Professor Gian Antonio Susto. Following his advice and learning from his experience has been extremely important both for the development of this thesis, but also for my personal growth. I also want to really thank him for his availability and kindness during all these months.

Another person that I want to thank is my co-supervisor, Professor Mattia Bruschetta. His suggestions and his help have been really important for me and for this thesis.

Finally, I want to thank my family and all my friends for their presence and support throughout all these years that have brought me to this moment.