Evolving Cooperation in Multi-Agent Systems

Author: Aglaia Elli GALATA
Supervisor: Dr. Mario MARTÍN

A thesis submitted in fulfillment of the requirements for the degree of Master in Artificial Intelligence

in the

April 19, 2021
Abstract

Evolving Cooperation in Multi-Agent Systems

by Aglaia Elli Galata

Applications of deep reinforcement learning in multi-agent systems is a rapidly developing scientific field that focuses on constructing computational frameworks and models for how various agents can interact efficiently to achieve their goals. In this thesis we tackle the challenging coordination problem in which the agents have to cooperate to achieve a common task.

First we evaluate the robustness of the state-of-the-art algorithms: VDN and MADDPG in more complex scenarios of higher agent cardinality. We will analyse if the agent populations are able to discover various physical and informational coordination strategies, since the variance of the models increases proportionally to the number of the agents. To overcome the limitations derived from their low cooperation and their slow convergence, we introduce MAGAT model. This new actor-critic model, incorporates graph neural layers to encourage a better representation of the agent relations. The MADDPG algorithm demonstrates a poor performance on the initial scenario, motivating us to modify the current rewarding scheme. Although our model yielded some promising preliminary results, it seemed more challenging to fine-tune the self-attention architecture and requires further analysis.

Designing optimal policies for coordination problems is a highly active and open research area. Through our insights, we hope that we will contribute to the evolution of the cooperation analysis and our work will be the building block for future studies.
Acknowledgements

Firstly, I would like to deeply thank my Supervisor, Dr. Mario Martin, for introducing me to the world of reinforcement learning. He has been one of the most inspiring, empowering, and caring people I have been fortunate to meet and has made me love the field in which I have worked. Secondly, I am really grateful to the High Performance Artificial Intelligence of the Barcelona Supercomputing Center for allowing me to use their infrastructure. I will like also to thank my family for continuously supporting me through anything I put my heart and soul in. Additionally, I will like to deeply thank my friends Alex, Federico, Daniel and Michael for believing in me and supporting me through this educational journey. Finally, I would like to thank all my MAI friends for all the laughs and support I received from them.
Contents

Abstract iii
Acknowledgements v

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Contributions ................................................ 2
  1.3 Overview .................................................... 3

2 Prerequisites 5
  2.1 Reinforcement Learning ........................................ 5
    2.1.1 The Reinforcement Learning Problem ..................... 5
    2.1.2 Optimality Concepts ..................................... 6
    2.1.3 Bellman Equation ....................................... 7
    2.1.4 Single-Agent RL Algorithms ............................. 8
    2.1.5 Replay Buffer ........................................... 11
    2.1.6 Exploration Policies .................................... 12
  2.2 Matrix Games .................................................. 13
    2.2.1 Definition and strategy ................................ 13
    2.2.2 Nash Equilibrium ....................................... 14
    2.2.3 Types of matrix games ................................ 14
    2.2.4 The coordination problem ............................... 15
  2.3 Multiagent Systems ............................................ 15
    2.3.1 Dynamics in Cooperative Multiagent Systems .......... 15
    2.3.2 The multiagent reinforcement learning problem ........ 16
    2.3.3 Challenges in multiagent RL ........................... 16
    2.3.4 Multiagent RL Algorithms ............................... 17
    2.3.5 Modeling other agent information ....................... 20
  2.4 Graph Neural Networks ........................................ 21
    2.4.1 Graph Representation Learning .......................... 22

3 Methods and Experiments 27
  3.1 Multiagent Particle Environment ............................. 27
  3.2 Mixing Strategies ............................................ 28
  3.3 Actor Critic Strategies ....................................... 30
    3.3.1 Experiments on the Multiagent Algorithm for Deep Deterministic Policy Gradients ........................................ 30
    3.3.2 Experiments on the Multiagent Algorithm for Deep Deterministic Policy Gradients with Graph Attention Layers ........ 30
    Hyperparameter Search ........................................ 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Results</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Original Environment</td>
<td>33</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Results on Mixing Strategy Algorithms</td>
<td>33</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Results on Multiagent Algorithm for Deep Deterministic Policy Gradients</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Updated Environment</td>
<td></td>
</tr>
<tr>
<td>4.2.1</td>
<td>Results on Multiagent Algorithm for Deep Deterministic Policy Gradients</td>
<td>37</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Results on Multiagent Algorithm for Deep Deterministic Policy Gradients with Graph Neural Networks (MAGAT)</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Discussion</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Conclusions and Contributions</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Work</td>
<td>50</td>
</tr>
<tr>
<td>A</td>
<td>Technicalities</td>
<td>51</td>
</tr>
<tr>
<td>A.1</td>
<td>Github repository</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td><strong>Bibliography</strong></td>
<td>53</td>
</tr>
</tbody>
</table>
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>GNN</td>
<td>Graph Neural Networks</td>
</tr>
<tr>
<td>MDP</td>
<td>Markov decision process</td>
</tr>
<tr>
<td>FCN</td>
<td>Fully connected network</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional neural network</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent neural network</td>
</tr>
<tr>
<td>GAT</td>
<td>Graph Attention Networks</td>
</tr>
<tr>
<td>GCN</td>
<td>Graph Convolutional Networks</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This master project aims to evaluate and extend reinforcement learning algorithms for cooperative multiagent tasks, i.e., in scenarios where agents share a common reward. It mainly aims at the coordination games that yield multiple pure-strategy Nash equilibrium. Different approaches aim to tackle the multiagent credit assignment problem and promote the cooperation of the agents.

1.1 Motivation

In the last decades, the scientific interest in robotic and software agents has increased since they can perform tasks that we would barely accomplish otherwise. For instance, autonomous driving involves various agents employed to control the road speed limits, the drivable zones, avoiding collisions, etc. Furthermore, in the industrial sector, learning-based robots perform dangerous for people tasks in the most efficient way.

Mainly, multiagent systems motivate distributed solutions that can be cheaper and more efficient than centralized single-agent ones. Furthermore, many real-life applications can be simulated via multiagent cooperative frameworks, where individual agents’ actions directly influence other agents and individual independent rewards are not provided.

In this context, reinforcement learning (RL) trains agents to compute optimal ways of performing the required tasks, with just a few instructions indicating if the task was or was not accomplished.

Despite its numerous applications that will be discussed in section 2.3, multiagent RL is considered an open scientific area. Innovative RL algorithms are primarily applied to relatively simple applications either in scenarios with reduced complexity or game theory examples with only a few (typically two) learning agents involved. Although applying RL algorithms in simplified scenarios benefits the analysis of its limitations and challenges, it struggles in generalizing to more complex environments. Thus, we should evaluate the multiagent algorithms on whether they can fulfill their expectations on potential real-life applications.

However, learning in multiagent systems arises additional learning problems due to the continually changing decision-making policies of agents throughout the training procedure. This non-stationarity stems from breaking the Markov assumption that governs most single-agent RL algorithms. The training procedure also sustains the multiagent credit assignment problem: i.e., the subset of agents that acted optimally and contributed to the cooperative reward should be elected, whereas the rest of the agents must be penalized. An additional obstacle constitutes the partial observability property that poses the agents in a short-sighted state, receiving only their local observations. This property enhances the stochasticity of the environment
and hinders the training performance. Finally, heterogeneous agents may have different sensing and acting capabilities, and thus their learning algorithms may differ.

Scientific research and practice in multiagent systems focus on constructing computational frameworks and models for how various agents can interact effectively to achieve their goals. When designing such frameworks, the most challenging problem is how to train the agents to cooperate to get the optimal policy. This spectrum of problems is called the cooperation problem, and a typical example is the coordination problem. This problem is particularly hard since the concept of individual rewards has been replaced by a joint reward function where the effectiveness of an agent’s action is inextricably linked to other agent actions. Furthermore, in some environments, a higher level of cooperation is required. For instance, the agents should select a sub-optimal policy for them because an altruistic behavior may yield a better joint policy for their common goals. One way to achieve this is to induce biases in our models that will promote the coordinated behavior of the agent. Nowadays, graph neural networks (GNNs) has been proven a powerful mathematical tool to encode interactions between different entities in various fields. Motivated by the success of these models, we intend to model the interaction between the agents as a graph in which each neuron is an agent, and the edges represent their interactions.

In this thesis, we will evaluate the robustness of the state-of-the-art algorithms in the multiagent particle environment, first published by Lowe et al. (2020) along with their multiagent actor-critic algorithm. Our primary goal is to generalize cooperative multiagent algorithms for coordination games to complex scenarios of higher agent cardinality. First, we will evaluate the dynamics of current multiagent algorithms in more complex settings. Then, we will present a new multiagent architecture, which incorporates graph neural networks (GNNs) during the decentralized training procedure to promote cooperation among the agents.

1.2 Contributions

This thesis applies and analyzes for the first time GNNs in a cooperation multiagent setting to the best of our knowledge. This network architecture is applied in value-based algorithms with centralized mixing strategies and actor-critic structures with various alterations. The Value Decomposition algorithm proposed by Sunehag et al. (2017) was utilized as the primer mixing strategy, whereas the q-utility functions were implemented with various algorithms, such as deep q-networks published by Mnih et al. (2013), dueling double deep q-networks by Wang et al. (2016), deep recurrent q-networks by Hausknecht and Stone (2017) and temporal attention q-networks by Vaswani et al. (2017). The GNN was employed as an initial encoder of each agent’s state information, correlating neighbor agents and inducing bias if they were located in close proximity. On the other hand, in actor-critic structures, the GNN is incorporated on the critics’ network, aligned with the centralized training and decentralized-execution framework since the agents are completely independent on testing time.

Furthermore, various multiagent state-of-the-art algorithms were training and tuned on a larger scale cooperation-navigation scenario from the multiagent particle environment without explicit communication between the agents. Multiagent deep deterministic policy gradient algorithm was tuned successfully up to five agent scenarios. Finally, alternative architectures with one critic for all the agents were built, considering the experiment’s agent homogeneity.
1.3 Overview

This thesis is split into five main parts. Chapter 2 covers the theoretical background related to later parts of this work. Section 2.1 sets the broader context of this work and describes the basics of reinforcement learning, which includes general notation and definitions, as well as the basic mathematical formulation of the implemented algorithms. In particular, in section 2.1.4 we analyse the value-based employed algorithms and in section 2.1.4 the actor-critic ones.

In section 3 we initially present the environment of our experiments along with its default and later configurations. Subsequently, in sections 3.2 and 3.3 we illustrate the architectures of the executed algorithms for this thesis and perform detailed explanations on the distinctive layers of the corresponding networks.

In chapter 4 we demonstrate the results of the aforementioned experiments. In section 4.1, we apply the analyzed algorithms on the default environment, whereas in section 4.2 we repeat them in a more interactive environment.

Finally, in chapter 5 the obtained results and the limitations of the used methods are discussed. To overcome these limitations, we propose potential future directions.
Chapter 2

Prerequisites

This thesis implemented various multiagent approaches either with their default settings, from their corresponding papers, or with hyper-parameter tuning and GNN enhancements. This chapter summarizes these methods and places them into the broader context of research on cooperative multiagent RL. In section 2.1.1 we give an overview of RL by introducing the necessary notations, analysing the single-agent framework and the fundamental algorithms used in this thesis. It should be mentioned that the terminology and definitions of this work stem from Sutton and Barto (2018). Then, in section 2.2 we define matrix games and Nash equilibrium, which constitutes the basis of section 2.3, where the multiagent setting is introduced. In the subsequent subsections, we provide significant challenges that multiagent RL is confronting along with the analysis of the multiagent algorithms used in this thesis. Our implemented algorithms can be separated into two broader categories: value-based algorithms analyzed in subsection 2.1.4 and actor-critic algorithms explained in section 2.1.4. Finally, in section 2.4 we introduce graph neural networks (GNNs), attention networks, as well as their relation with multiagent RL. Finally, in section 2.3.4 we provide a comprehensive review of cutting-edge multiagent RL algorithms.

2.1 Reinforcement Learning

Reinforcement learning is a branch of machine learning characterized by many researchers as the next massive Artificial Intelligence (AI) revolution. The learning problem corresponds to the mapping of situations to actions to maximize a numerical reward signal. The reward is a way of showing how good the action was at that particular instant/situation. An RL problem consists of an agent and an environment, where the first has to discover the actions that yield the most reward by a trial and error procedure. This essential characteristic differentiates RL from supervised learning. Additionally, in more exciting scenarios, RL deals with sequential decision-making. In these problems, the utility of the agent’s actions does not depend on single decisions, expressed with the state yielded from this decision, but instead on the whole sequence of the agent’s actions. Thus, actions may affect the immediate reward, the following situation and so all subsequent rewards. As stated from Sutton and Barto (2018), the two most important distinguishing features of reinforcement learning are the trial-and-error search and the delayed reward.

2.1.1 The Reinforcement Learning Problem

In a single-agent scenario reinforcement learning is typically formalized as solving a Markov decision process (Bellman, 1957a; Bellman, 1957b), which is a tuple \((S, A, R, T, \gamma)\) with the following components:
Chapter 2. Prerequisites

- $S$ is a finite set of states,
- $A$ is a finite set of actions,
- $T : S \times A \times S \rightarrow [0; 1]$ is a transition function that defines a probability distribution over next states. $T(s, a, s')$ is the probability of reaching $s'$ when the action $a$ is executed in $s$, also noted $P(s'|s, a)$,
- $R : S \times A \times S \rightarrow \mathbb{R}$ is a reward function giving the immediate reward or reinforcement received under each transition. Specifically, $R(s, a, s') = \mathbb{E}\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}$. In this context $r_{t+1}$ represents the immediate payoff of the environment to the agent at time $t + 1$.
- $\gamma \in [0, 1]$ is the discount factor that trades off the instantaneous and future rewards.

MDP has been widely adopted to characterize an agent’s decision-making with full observability of the system state $s$. At each time step, the agent chooses an action $a_t$ that leads the system from the state $s_t$ to the new state $s_{t+1}$ according to a transition probability $T(s_t, a_t, s_{t+1})$. Then it receives an instantaneous reward $r_{t+1} = R(s_t, a_t, s_{t+1})$. Solving MDPs consists of finding a mapping from states to actions, called a policy $\pi$ that maximizes the expected reward. Policies can either be deterministic ($\pi : S \rightarrow A$, i.e., transition probabilities and reward distributions do not change in time) or stochastic ($\pi : S \times A \rightarrow [0; 1]$), mapping from states to action probabilities. If not stated otherwise, we will assume the policy to be stochastic and denote the probability of picking action $a$ in-state $s$ with $\pi(a|s)$.

2.1.2 Optimality Concepts

The goal of an agent living in an environment that can be modeled as a Markov Decision Process is to maximize its expected reward over time. The most common ways of modelling a task are:

1. Finite-horizon model:
   In this scenario the agent tries to maximize the sum of rewards for the following $M$ steps:
   $$\mathbb{E}\left\{ \sum_{k=0}^{M} r_{t+k+1}|s_t = s \right\}$$ (2.1)
   The objective is to find the best action, considering there are only $M$ more steps in which to collect rewards.

2. Infinite-horizon discounted reward model:
   In this scenario the goal of the agent is to maximize reward at the long-run but favoring short term actions:
   $$\mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s \right\}, \gamma \in [0, 1]$$ (2.2)
   In this thesis, following the trends in the reinforcement learning bibliography, we adopt the Infinite-horizon discounted reward model.
2.1.3 Bellman Equation

Solving MDPs have always been closely tied to the idea of dynamic programming. There are different techniques for solving MDPs, assuming a complete description of all its elements. Much research has been done in that area, and extensive studies of the subject can be found in texts like Ross (1983), Bertsekas and Tsitsiklis (1996), and Bertsekas (2000). In this thesis, we apply model-free algorithms, i.e., we do not know the transition probability distribution (and the reward function) associated with the Markov decision process (MDP), opposed to the model-based approaches. A fundamental concept of MDPs is the state-value function, which provides the expected reward for a specific state $s$, given that the agent is following a policy $\pi$:

$$V^\pi(s) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, \pi \right\}$$ (2.3)

Similarly, the state-action value function or Q-function is defined as the expected reward given that the agent takes action $a$ in state $s$ and following policy $\pi$:

$$Q^\pi(s,a) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a, \pi \right\}$$ (2.4)

From the equation 2.3, Bellman (1957) derived a recursive relation, called Bellman equation, which is the basic concept behind dynamic programming and reinforcement learning algorithms.

$$V^\pi(s) = \mathbb{E}_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, \pi \right\} = \mathbb{E}_{\pi}\left[ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, \pi \right] = \mathbb{E}_{\pi}\left[ r_{t+1} + \gamma V^\pi(s_{t+1}) \right]$$ (2.5)

An optimal policy is the one that guarantees a maximal reward, in the sense that no other policy is expected to do better in any state. For value function is defined as follows:

$$V^*(s) = \max_{\pi} V^\pi(s), \forall s \in S$$ (2.6)

and for q-function:

$$Q^*(s,a) = \max_{\pi} Q^\pi(s,a), \forall s \in S, \forall a \in A$$ (2.7)

Considering the equations 2.3, 2.4 and 2.5, the Bellman optimality equations for state-values and q-values can be derived:

$$V^*(s) = \max_{a \in A} Q^\pi(s,a)$$ (2.8)

$$Q^*(s,a) = \mathbb{E}_{s' \sim T(s,a)} [R(s') + \gamma V^\pi(s')]$$ (2.9)
2.1.4 Single-Agent RL Algorithms

The single-agent RL algorithms are classified in three categories that will be analysed below:

1. Value-Based Methods
2. Policy-Gradient Methods
3. Actor-Critic Methods

Value-Based Methods

The first RL approach is value-based learning, which is considered implicit learning since the value function is initially learned, and the optimal policy is inferred as \( \pi(s) = \arg \max_a Q(s,a) \).

One of the most well known algorithms for RL is Q-learning (Watkins, 1989), which is devised for stationary, single-agent, fully observable environments with discrete actions. Considering the q-function 2.4 and the bellman equation 2.5, the update of each tabular state \( \hat{Q}(s,a) \) is an estimate of the corresponding optimal \( Q^* \) function, defined as:

\[
\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha [r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)]
\]  

(2.10)

with the learning rate \( \alpha \in [0,1] \). Q-learning is proven to converge to \( Q^* \) if state and action spaces are discrete and finite (Jaakkola, Jordan, and Singh, 1994; Szepesvári and Littman, 1998; Even-Dar and Mansour, 2004; Szepesvári, 2010; Kamihigashi and Le Van, 2015).

In this thesis, we implemented the value-based algorithms: Deep Q-Learning and Dueling Q-Learning with and without recurrency.

- **Deep Q-learning:**
  This algorithm was published from Mnih et al. (2013) and got a lot of attention due to its supper human-level performance on the Atari-2600 games (for details, see Bellemare et al. (2013)). It utilizes the replay memory buffer of Lin (1992) and a moving target network to stabilize the training. The replay memory holds the observation tuple \((s_t, a_t, r_t, s_{t+1}, d_t)\) in which \(s_t\) denotes the current state, \(s_{t+1}\) the next state of the agent after executing the action \(a_t\) and \(d_t\) determines if the episode ended with this observation. Then the approximator is trained by taking a random mini-batch from the replay memory. It includes a sequence of fully connected layers with weights \( \theta \) to approximate the Q-value (see equation 2.10) for each possible action, which is therefore trained by taking a mini-batch of size \( m \) to minimize loss function:

\[
L(\theta) = \frac{1}{m} \sum_{i=1}^{m} (y_i - Q(s_i, a_i; \theta))^2
\]  

(2.11)

\[
y_i = \begin{cases} 
  r_i & d_t = \text{True}, \\
  r_i + \gamma \max_{a'} Q(s_i', a', \theta^-) & d_t = \text{False}
\end{cases}
\]  

(2.12)

where, \( \theta^- \) is the weights of the target network which is updated by \( \theta \) every \( C \) iterations.
2.1. Reinforcement Learning

- **Dueling Deep Q-learning:**
  This algorithm extends the DQN, trying to solve its value overestimation flaw. Specifically, through the operation of the equation 2.10:

  $$\max_{a'} Q^* (s', a'),$$  

  (2.13)

  in the Q-function calculation, the estimate of the maximum value is retrieved. In a scenario where the true Q value for all actions are equal to 0, and the estimated Q values are distributed some above and below zero, taking the maximum of these estimates (which is obviously bigger than zero) to update the Q function leads to the overestimation of Q values. This systematic overestimation introduces a maximization bias in learning. And since Q-learning involves bootstrapping — learning estimates from estimates — such overestimation can be problematic.

  An initial solution to this problem was published from Hasselt, Guez, and Silver (2015) with the Double deep q-learning algorithm. It involves two separate Q-value estimators, each of which is used to update the other. The use of two independent estimators results to unbiased Q-value estimates of the actions selected using the opposite estimator.

  The Dueling DQN architecture also employs two different estimations for the Q-values. The motivation behind it is that for some games, it is unnecessary to know the value of each action at every timestep, but we should learn instead which states are (or are not) valuable. Thus, given the agent’s policy $\pi$, the action value $V^\pi$ and state value $Q^\pi$, we can define the advantage value subtracting the Q-value by the V-value:

  $$A^\pi (s, a) = Q^\pi (s, a) - V^\pi (s)$$  

  (2.14)

  Intuitively, it indicates how advantageous selecting an action is relative to the others at the given state. The DQN split into two separate streams, one for the state-value and one for state-dependent action advantages. After the two streams, the last module of the network aggregates the state-value and advantage outputs, which leads to the final q-values.

- **Deep Recurrent Q Network (RDQN):**
  Deep Recurrent Q-Network extends the capabilities of DQN, being capable of exploiting the past observations of the agent and successfully integrates this information to the current predictions. Hausknecht and Stone (2017) utilized a Long short-term memory (LSTM) layer in the network in order to evaluate past useful information. This network is suitable for stochastic games where the past actions influence the learning procedure and memory is important, i.e. common-pool resource games.

  An alternative of the LSTM layer on exploiting temporal information constitutes a self-attention layer, which relates different temporal observations of a single state to compute a new representation which encompass these past observation’s information. The self-attention architecture will be further discussed in section 2.4.1.
Policy-Gradient Methods

In contrast to value-based methods, which do not try to optimize directly over a policy space, policy gradient methods can learn parameterized policies without using intermediate value estimates. The objective function for policy gradients is defined as:

$$J(\theta) = \mathbb{E} \left[ \sum_{t=0}^{T-1} r_{t+1} \right]$$

thus, to learn a policy that maximizes the cumulative future reward to be received starting from any given time \( t \) until the terminal time \( T \).

Williams (1992) published REINFORCE algorithm which uses estimated return by Monte Carlo (MC) methods with full episode trajectories to learn policy parameters \( \theta \), with \( \pi(a; s, \theta) \approx \pi(a; s) \), as follows:

$$\theta_{t+1} = \theta_t + aG_t \frac{\nabla \pi(A_t; S_t, \theta_t)}{\pi(A_t; S_t, \theta_t)}$$

where \( G_t \) represents the return, \( a \) is the learning rate, and \( A_t \sim \pi \). Hence, the derivative of the objective function \( J \) that we try to optimize becomes as follows:

$$\nabla_{\theta} J(\theta) = \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi(\theta(a_t, s_t)) G_t$$

A main limitation is that policy gradient methods can have high variance (Konda and Tsitsiklis, 2001).

Actor-Critic Methods

It is a combination of the two aforementioned approaches. The policy gradient methods introduce high variability in log probabilities (log of the policy distribution) and cumulative reward values, since each trajectory during training can deviate from each other at great degrees. Consequently, they result in noisy gradients, and cause unstable learning, where the policy distribution is skewing to a non-optimal direction. Furthermore, in cases where trajectories have a cumulative reward of 0, both "goods" and "bad" actions will not be learned, since the essence of policy gradient is increasing the probabilities for "good" actions and decreasing those of "bad" actions in the policy distribution.

Solving the occurring instability and slow convergence of vanilla policy gradient methods, demands reducing variance with a baseline \( b(s) \), such as:

$$\nabla_{\theta} J(\theta) = \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi(\theta(a_t, s_t)) (G_t - b(s_t))$$

Intuitively, making the cumulative reward smaller by subtracting it with a baseline will make smaller gradients, and thus smaller and more stable updates. Based on the choice of the baseline, multiple algorithms have derived, such as:

$$\nabla_{\theta} J(\theta) = \begin{cases} E_{\pi_\theta}[\nabla_{\theta} \log \pi(\theta(a_t, s_t))G_t] & \text{REINFORCE} \\ E_{\pi_\theta}[\nabla_{\theta} \log \pi(\theta(a_t, s_t))Q^w(s, a)] & \text{Q Actor-Critic} \\ E_{\pi_\theta}[\nabla_{\theta} \log \pi(\theta(a_t, s_t))A^w(s, a)] & \text{Advantage Actor-Critic} \\ E_{\pi_\theta}[\nabla_{\theta} \log \pi(\theta(a_t, s_t))\delta] & \text{TD Actor-Critic} \end{cases}$$
Hence, Actor-Critic Methods use two neural networks:

1. The Critic, which estimates the value functions and
2. The Actor, which updates the policy distribution in the direction suggested by the Critic (such as with policy gradients)

In this thesis, an extension of the original Deep Deterministic Policy Gradients (DDPG) algorithm is implemented in the multiagent setting (Lillicrap et al., 2019). It uses four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network.

The network resembles the Advantage Actor-Critic model, but in DDPG, the Actor directly maps states to actions instead of outputting the probability distribution across a discrete action space. Another variation of this algorithm is that the next-state Q values are calculated with the target value network and target policy network. The target network parameters for the target policy and q network are soft updated as shown below:

\[
\begin{align*}
\theta_Q' &\leftarrow \tau \theta_Q + (1 - \tau) \theta_Q' \\
\theta_\mu' &\leftarrow \tau \theta_\mu + (1 - \tau) \theta_\mu'
\end{align*}
\]

where \( \tau \ll 1 \).

**Bias Overestimation**

In Q-learning with discrete actions, the value estimate is updated with a greedy target \( y = r + \gamma \max_{a'} Q(s', a') \), which constitutes an estimation with error \( \epsilon \). Thus, in every learning loop, we calculate the maximum over the \( Q(a', s') \), which will always be greater than the true maximum (Thrun and Schwartz, 1993).

\[
\mathbb{E}[\max_{a'} Q(s', a') + \epsilon] \geq \max_{a'} Q(s', a')
\]

This continuous increase in the value estimation results in a consistent overestimation bias, which is further propagated through the Bellman’s equation. It should be stated that this erroneous behavior is not only present on the value-based methods, but also in the actor-critic algorithms. In the latter, the policy is updated via gradient descent with respect to the value estimates of the critic network, which is a value-based model. Fujimoto, Hoof, and Meger (2018) mathematically proves that the value estimate will be overestimated if the critic’s value estimate is at least as large as the true value with respect to the actor’s model parameters.

\[
\mathbb{E}[Q_\theta(s', \pi_{\text{approx}}(s))] \geq \mathbb{E}[Q^\pi(s', \pi_{\text{approx}}(s))]
\]

**2.1.5 Replay Buffer**

A critical component of DQN-based algorithms is the experience replay (Lin, 1992). It is a buffer of fixed-size that stores the most recent transitions collected in the environmental steps. The exploitation of the same trial various times (instead of throwing away the data immediately after collection) improves the sample efficiency of the algorithm.

A standard way of implementing the experience replay is as a circular buffer. First, the buffer receives new experiences until it fills its capacity, and subsequently,
it removes old experiences and replaces them with new ones. During the learning process, a number of experiences equal to the pre-defined batch size are sampled randomly from the buffer. The most basic sampling strategy is the uniform sampling, where each experience has an equal probability of getting elected for training.

A more sophisticated approach is the prioritized experience replay (Schaul et al., 2016). The observation that motivated this approach is that some experiences are more valuable than others. In a uniform sampling every experience is used approximately the same number of times at the end of the training. However, if we sample with weights, some more beneficial experiences will get sampled more times on average. One way of implementing these sampling weights is to employ a sampling probability, proportional to the loss obtained after the forward pass of the neural network. Thus, the highly weighted experiences are the ones that led to an important difference between the expected reward and the reward that the model actually got. Conceptually, the experiences that are kept on the buffer lead the neural network to learn a lot.

2.1.6 Exploration Policies

The exploration-exploitation trade-off is a well-known problem in scenarios where a learning system has to repeatedly make a choice with uncertain pay-offs. In a decision-making system, the agent has to decide whether to repeat decisions that have worked well so far (exploit) or make novel decisions, hoping to gain even greater rewards (explore). Once the model is learned, the agent can exploit its knowledge to plan the most rewarding actions given a task. This exploitation encourages model-based learning approaches to display generalization capabilities. However, initially, the agent has no prior knowledge of the environment, and needs to resort to exploration to search and learn within the environment before any exploitation can be applied.

Depends on the type of the environment’s action space (discrete or continuous), the exploration policy is changing. For discrete action spaces, the exploration is done via probabilistically selecting a random action, such as e-greedy algorithms (i.e., adaptive epsilon-greedy of Tokic (2010)) or Boltzmann exploration (see Cesa-Bianchi et al. (2017)). For continuous action spaces, exploration is done via adding noise to the action itself.

E-greedy

E-greedy exploration policy is applied only to discrete action problems. It uses a probability \( \epsilon \) for exploring the environment and a probability \( 1 - \epsilon \) for exploiting the best policy so far. Typically, most of the algorithms that employ \( \epsilon \)-greedy exploration policy, initially set a high \( \epsilon \) to encourage exploration and afterward reduce it, as they gather knowledge about the rewards. Depending on the environment, the \( \epsilon \) probability affects differently the learning process. This motivated the creation of various decay functions to control the reducing rate. In most discrete action tasks, either a linear or an exponentially decay strategy is chosen.

Gumbel-Softmax

Gumbel-Softmax is an exploration policy for discrete action problems, which was independently discovered from Jang, Gu, and Poole (2017) and Maddison, Mnih, and Teh (2017). It is a sampling process, which samples actions from a distribution
to encourage exploration. However, the sampling process of discrete data from a categorical distribution is not differentiable, meaning that backpropagation is not working. The Gumbel-Softmax distribution is a continuous distribution that approximates samples from a categorical distribution and also works with backpropagation.

**Ornstein-Uhlenbeck Process**

DDPG algorithm uses Ornstein-Uhlenbeck Process to add noise to the action output (see Finch (2004)). The Ornstein-Uhlenbeck Process (OU) evolves the agents’ velocity according to Langevin equation (Wikipedia contributors, 2021), where in the large time limit, the sampled velocities follow a Gaussian distribution. It generates noise correlated with the previous noise to prevent it from canceling out or “freezing” the overall dynamics. In the continuous setting, the OU process is defined as:

$$dX_t = -\beta (X_t - a)dt + \sigma dW_t,$$

where $X_t$ stands for an environmental state, $\alpha$ and $\beta$ are constants that decide certain movement characteristics of the agents, $W_t$ is a Weiner process and $\sigma$ is the weighting factor of the Weiner process, meaning the amount of noise being added to the process. The Weiner process starts with $W_0 = 0$ and then adds independent increments of $\mathcal{N}(\mu, \sigma)$ as $W_{t+1} = W_t + \mathcal{N}(\mu, \sigma)$. This is formulated as $W_t - W_s = \sqrt{t-s} \mathcal{N}(0,1)$. This is because of the fact, $W_t$ can be written recursively as $W_t = \mathcal{N}(0,1) + W_{t+1} = \mathcal{N}(0,1) + \mathcal{N}(0,1) + \ldots W_s$ and since the samplings are independent at each step the mean get added as $\mu_t = \mu_{t-1} \ldots$ and the variances as $\sigma^2_t = \sigma^2_{t-1} \ldots$. Since the means and the variance are 0 and 1 respectively, the final mean is: $\mu = 0$, and the variance is: $\sigma^2 = (t-s)$. Hence, occurs the initial Weiner process as $W_t - W_s = \sqrt{t-s} \mathcal{N}(0,1)$.

### 2.2 Matrix Games

It is essential to introduce the fundamental concepts of game theory, before proceeding to the multiagent framework employed on this thesis. Game theory is commonly used to model, as a game, strategic interactions between a set of players and thus to analyze multiagent decision making. A game is defined as a mathematical object, that describes the consequences of interactions between player strategies in terms of individual payoffs. In this section, we will define a matrix game framework, and we will classify the games based on their payoff function. Finally, we will analyze the category of cooperation coordination games, which constitutes the basic environment of the thesis.

#### 2.2.1 Definition and strategy

**Definition**

A matrix game or normal form game (Neumann, Morgenstern, and Rubinstein, 1944; Rubinstein and Osborne, 1994) is a multiple agent, single state framework. It is defined as a tuple $< m, A_1, \ldots, A_m, R_1, \ldots, R_m >$ where:

- $m$ is the number of players;
- $A_i$ is the set of choices available to player $i$ (and $A = A_1 \times \cdots \times A_m$ is the joint action space);

$$2.2. \text{ Matrix Games}$$
Chapter 2. Prerequisites

• \( R_i : A \mapsto \mathbb{R} \) is player’s \( i \)'s reward or payoff function.

The goal of a learning agent in a matrix game is to learn a strategy that maximizes its reward. With respect to the RL framework notations, a strategy (policy) \( \pi_i : A_i \mapsto [0,1] \) for an agent \( i \) specifies a probability distribution over actions, i.e. \( \forall a_i \in A_i, \pi_i(a_i) = P(a_i) \). Additionally, \( \pi \) is the joint strategy for all of the agents and \( \pi_{-i} \) is the joint strategy of all agents except agent \( i \). Similarly, \( A_{-i} \) is the set of actions for all of the agents except agent \( i \). The set of all of the strategies for the agent \( i \) is noted \( \Delta(A_i) \). The matrix game can be called a repeated games when it is repeatedly played by the same agents. The main difference from a one-shot game is that the agents can use some of the game iterations to gather information about the other agents or the reward functions, and make more informed decisions thereafter.

2.2.2 Nash Equilibrium

The Nash equilibrium is a decision-making theorem within game theory that states that a player can achieve the desired outcome by not deviating from its initial strategy. Formally, it is a joint strategy \([\pi_1^*, \ldots, \pi_n^*]^T\) such that each individual strategy \( \pi_i \) is a best-response to the others. It describes a status quo, where no agent can benefit by changing its strategy as long as all other agents keep their strategies unchanged. Every player wins because everyone gets the outcome they desire. A game may include multiple Nash equilibrium or none of them, but this does not guarantee that the most optimal strategy is selected. A standard game theory example that adequately showcases the effect of the Nash equilibrium is the prisoners’ dilemma: a two-agent game with prison sentence rewards, formalized by Albert W. Tucker. Namely, it is a game in which the optimal strategy for two completely rational individuals is not to cooperate, even if it appears that it is in their best interests to do so (see Neumann and Morgenstern (1944)).

2.2.3 Types of matrix games

A detailed survey on matrix game categories performed from Busoniu, Babuska, and De Schutter (2008). In general, matrix games can be classified according to the structure of their payoff function to three broad categories:

1. **Fully Competitive Game** or zero-sum game:
   In a two agent game, the rewards are defined as \( r_1 = -r_2 \). The minimax principle can be applied: maximize one’s benefit under the worst-case assumption that the opponent will always endeavor to minimize it (Littman, 2001). Did-digi, Kamanchi, and Bhatnagar (2020) published the minimax-Q algorithm that employs this principle.

2. **Fully Cooperative Game**:
   In a fully cooperative game, the agents have the same reward function \( (r_1 = \cdots = r_n) \), and the learning goal is to maximize the common discounted return. A non-trivial characteristic is the absence of a centralized controller, a fact that violates the convergence assumption of algorithms as Q-learning, since a coordination problem arises (will be further analyzed in section 2.2.4).

3. **Mixed games** or general-sum games: In this case, there is no constraint on the sum of the agents’ rewards. Cooperative agents may as well encounter situations where their immediate interests are in conflict, i.e. competing over
the same resource. The learning problem is non-stationary due to the dynamic behavior of the agents.

2.2.4 The coordination problem

In multiagent games crucial is the need for cooperation. The cooperation necessity will be analyzed through a game of two agents that need to maintaining formation. The Q-values on a specific state with the available actions: left \((a_0, b_0)\), right \((a_2, b_2)\), stay \((a_1, b_1)\) are illustrated on the table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>(a_0)</th>
<th>(a_1)</th>
<th>(a_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b_0)</td>
<td>10</td>
<td>0</td>
<td>-5</td>
</tr>
<tr>
<td>(b_1)</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>(b_2)</td>
<td>-5</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2.1: Common Q-values of thee agents with actions \(a_i\) and \(b_i\)

The payoffs to both players are identical, and wrong coordination results in a penalty. There are three pure equilibrium. While \(<a_0, b_0>\) and \(<a_2, b_2>\) are the optimal equilibrium, the symmetry of the game induces a coordination problem for the agents. With no means of breaking the symmetry, and the risk of incurring the penalty if they choose different optimal equilibrium, the agents might in fact focus on the suboptimal equilibrium \(<a_1, b_1>\).

If the agents choose their part of the equilibrium randomly, there is a 0.5 chance that they miscoordinate (one player choose to go right and the other to go left), thereby obtaining an expected immediate reward of \(-5\).

2.3 Multiagent Systems

In this section, we will analyze the dynamics of multiagent systems, real world application and challenges on the area.

2.3.1 Dynamics in Cooperative Multiagent Systems

Multiagent Reinforcement Learning is a system of agents (robots, machines, cars) interacting within a shared environment. Its contributions are notable in a wide variety of applications from robotics (walking, soccer, robotic rescue) and games (backgammon, Go from Silver et al. (2017)) to recommendation systems, finances (electronic markets), and electric power networks. Specifically, multiagent robot research seeks effective coordination among autonomous agents to perform a task to achieve a high overall performance (Ota, 2006). City problems such as traffic light control (also ground and air traffic control) have been confronted with multiagent RL algorithms. Zhang et al. (2019) proposed the CityFlow algorithm, which was used in many traffic signal control problems for large-scale city traffic scenarios. Wei et al. (2019) applied attention-based RL to tackle the traffic light cooperation task. OpenAI et al. (2019) proposed the openAI Five multiagent RL algorithm trained for the Dota mixed-strategy game, where five heterogeneous agents with partial observability need to cooperate and defeat their opponents.
Chapter 2. Prerequisites

Our framework evolves the application of RL algorithms to sequential decision problems in which multiple agents are controlling the system. Furthermore, the decision problem is cooperative since each agent’s reward is drawn from the same distribution, reflecting the utility assessment of all agents. The agents, heterogeneous or homogeneous, wish to choose actions that maximize the expected reward.

2.3.2 The multiagent reinforcement learning problem

The generalization of the Markov decision process to the multiagent case is the stochastic game. We denote a multiagent setting with tuple $< N, S, A, R, P, O, \gamma >$, in which:

- $N$ is the number of agents,
- $A = \{ A_1, \ldots, A_N \}$ is the set of actions for all agents,
- $P$ is the transition probability among the states,
- $R$ is the reward function and,
- $O = \{ O_1, \ldots, O_N \}$ is the set of observations for all agents.

Respectively with the normal form games presented in section 2.2, we use $a$ to denote the vector of actions for all agents and $a_{-i}$ is the vector of all agents except agent $i$.

2.3.3 Challenges in multiagent RL

The key challenges that are imposed on multiagent problems and will be analyzed in this section are the following:

- Non stationarity
- Exploration-Exploitation trade-off
- Curse of dimensionality
- Multiagent credit assignment: How much of the reward obtained corresponds to each of the agents?
- Modeling other agent information: An agent decides his action taking into account his predictions and reasoning on other agent’s actions.

Non stationarity

Controlling multiple agents poses several additional challenges as compared to a single agent setting. Similar to it, each agent is able to learn the optimal Q-value or the optimal stochastic policy. However, in the multiagent setting, the environment becomes non-stationary because all the agents in the system are learning simultaneously. Therefore, each agent is faced with a moving-target learning problem: the best policy changes as the other agents’ policies change. Consequently, the adopted Bellman equation for MARL (assuming the full observability) also does not hold for the multiagent system:

$$Q^*_i(s, a_i | \pi_{-i}) = \sum_{a_{-i}} \pi_{-i}(a_{-i} | s) \left[ r(s, a_i, a_{-i}) + \gamma \sum_{s'} P(s' | s, u_i, a_{-i}) \max_{a'_{-i}} Q^*_i(s', a'_{-i}) \right]$$

(2.24)
2.3. Multiagent Systems

where $\pi_{-i} = \Pi_{j \neq i} \pi_j(a_j|s)$. Due to the fact that $\pi_{-i}$ changes over time as the policy of other agents changes, one cannot obtain the optimal Q-value using the classic Bellman equation.

Furthermore, the changing of agents’ policies during training results to a non-valid experience replay, a fact that makes most of the value-based and actor-critic algorithms difficult to learn.

**Exploration-Exploitation trade-off**

In the single-agent setting, the agent has to decide in each action either to exploit of its current knowledge or to improve it by gathering more information. The exploration strategy is crucial for the efficiency of RL algorithms.

In the multiagent setting, the exploration-exploitation trade-off poses more complications, since the agents during exploration seek to obtain information not only about the environment, but also about the other agents. Thus, more exploration is required, which as a consequence leads to an increase in noise and variance, a fact that can destabilize the learning dynamics of the other agents.

**Curse of dimensionality**

This challenge encompasses the exponential growth of the discrete state-action space in the number of state and action variables (dimensions). In the basic RL algorithms, like Q-learning, we estimate values for each possible discrete state or state-action pair. On the multiagent RL, the growth on the number of agents leads directly to an exponential increase in the computational complexity. The complexity of MARL is exponential also in the number of agents, because each agent adds its variables to the joint state-action space.

**Multiagent credit assignment**

Multiagent credit assignment in the cooperative setting refers to the acquisition of one not decomposable cooperative reward for all the agents after the execution of the joint actions. This joint team reward make it difficult for a single agent to deduce his individual contribution to the team’s success or failure.

**Modelling other agents**

Tackling the non-stationarity problem arises the need of predicting the actions of the other agents. Thus, each agent will not face the aforementioned moving-target learning problem. For instance, Jaques et al. (2019) builds its own Model-Other-Agents architecture to compute the probability of another agent’s action.

2.3.4 Multiagent RL Algorithms

In this section, we will provide an overview of the state-of-the-art MARL algorithms, focusing on the Value Decomposition Networks and Deep Deterministic Policy Gradient algorithm that were used in the scope of this thesis. The following algorithms comply with the centralized training of decentralized policies framework in which each agent selects its action conditioned only on its local action-observation history.

Furthermore, this thesis works on the cooperation game framework. As explained in section 2.2.4, cooperation games require coordination, which needs predictability among the agents. It is crucial that the agents follow the strategy formed
on the learning process. During execution time, this strategy is conditioned on the current observation history.

**Independent Q-Learning**

Perhaps the most commonly applied method in multiagent learning is independent Q-learning (Tan, 1993), which decomposes a multiagent problem into a collection of simultaneous single-agent problems that share the same environment. Each agent learns its own q-values considering the rest of the agents as part of the static environment. This approach does not address the non-stationarity introduced due to the changing policies of the learning agents, and thus, unlike Q-learning, has no convergence guarantees even in the limit of infinite exploration. In practice, IQL usually serves as an experiment baseline of new approaches.

**Counterfactual Multiagent Policy Gradients (COMA)**

Foerster et al. (2017) is an Actor-Critic algorithm, which employs a centralized critic and a counterfactual baseline coping with the multiagent credit assignment problem. The learning is speeded up by sharing parameters among the agents, i.e., they learn only one critic and the actors are shared their parameters. The actors are trained with counterfactual(COMA) policy gradients, where the baseline is an advantage function, which compares the Q-value for the current action \( u_a \) to a counterfactual baseline that marginalises out \( u_a \), while keeping the other agents’ actions \( u_{-a} \) fixed. The drawbacks of COMA are listed below:

- Requires on-policy learning, which can be sample-inefficient
- The training of a fully centralized critic becomes impractical when there are more than a handful of agents.
- Shows poor reliability indicating high variance and noisy training with consequently large variation across multiple runs. This occurs due to high variance of its counterfactual advantage and consequently large variance critic losses, causing gradients for updates to be unstable

**Value Decomposition Networks (VDN)**

Sunehag et al. (2017) proposed to separate the action-value functions for multiple agents and to learn them by just one shared team reward signal. The joint action-value function is a linear summation of all agents’ action-value functions. An optimal linear value decomposition is learned by back-propagating the total Q gradient through deep neural networks representing the individual component value functions. Using a single shared reward signal, it tries to learn decomposed value functions for each agent and uses it for decentralized execution.

Each agent propagates its current observation through low-level linear layers. Then, a recurrent layer produces individual ‘values’, that are summed to a joint Q-function for training. Actions are produced decentralized from the individual outputs, while training is performed centrally from the joint Q-function.

The modeling of the individual Q-networks with the aggregation mixing strategy represents a completely disconnected factor graph with an extreme form of factorization; only one agent in each factor. However, this extreme factorization imposes the constraint of monotonicity on the individual q-function. This constraint can be better explained through the following mixing examples on the table 2.2.
2.3. Multiagent Systems

In the first q-value table, the monotonic mixing can decompose the mixing values to individual q-utilities since the relation between the individual q-values is monotonic. On the other hand, the monotonic mixing cannot capture the benefit of the coordination problem of the second table. In this case, the agent choice depends on which action the other agent will perform.

QMIX: Monotonic Value Function Factorisation for Deep multiagent Reinforcement Learning

Rashid et al. (2018) proposed the QMIX algorithm, based on which a full factorization of VDN is not necessary in order to be able to extract decentralized policies. In order to ensure that a global argmax performed on $Q_{tot}$ yields the same result as a set of individual argmax operations performed on each $Q_\alpha$, QMIX enforces a monotonicity constraint between $Q_{tot}$ and each $Q_\alpha$:

$$\frac{\partial Q_{tot}}{\partial Q_\alpha} \geq 0, \forall \alpha \in A$$

(2.25)

Each agent consists of an individual DRQN $Q_\alpha(\tau^\alpha, u^\alpha)$. The mixing network is a feed-forward neural network that takes as input the agent network outputs and mixes them monotonically (absolute activation function) in a non-linear way, producing the values of $Q_{tot}$.

The drawbacks on QMIX are presented below:

- The mixing network prevents the representations of value functions on which an agent’s ordering over its actions can depend on other agents’ actions (non monotonic action value functions). Thus, QMIX cannot solve tasks that require significant coordination within a given timestep as the problem on table 2.2.

- Poor exploration.

- QMIX ranks low in training stability compared to on-policy algorithms.

Multiagent for Deep Deterministic Policy Gradients (MADDPG)

MADDPG algorithm is an extension of the DDPG algorithm (see section 2.1.4) on the multiagent setting. Its structure is displayed on figure 2.1.

Each agent trains a DDPG algorithm such that the actor $\pi_i(o; \theta_i)$ with policy weights $\theta_i$ observes the local observations, while the critic $Q_{\mu_i}$ is allowed to access the observations, actions, and the target policies of all agents in the training time. Then, each critic concatenates all states-actions together as the input and using the
local reward obtains the corresponding Q-value. The critic is trained by minimizing a DQN-like loss function:

\[
L(\mu_i) = \mathbb{E}_{x_t,a_t,x_{t+1}} \left[ (Q_i(x'_t,a'_t,\ldots,a'_{N};\mu_i) - y)^2 \right],
\]

\[y = r_{it} + \gamma Q_i(x_t',a_{t+1}',\ldots,a_{N-1}')|a_{it} = \hat{\pi}(o_{it}')\]

(2.26)
in which \( \hat{\pi} \) is the target policy and \( \hat{\mu} \) is the target critic. Consequently, the critic of each agent deals with a stationary environment, and in the inference time, it only needs to access the local information.

Several variants of the MADDPG algorithm have been proposed. Ryu, Shin, and Park (2018) proposed MADDPG-GCPN algorithm which includes an extra actor network \( \mu_c \) to generate action samples of other agents. Lowe et al. (2017) presents another algorithm based on MADDPG for a cooperative game, called ATT-MADDPG which enhances the critic network by adding an attention layer. In this way, at agent \( i \), instead of just using \([o'_1,\ldots,o'_N] \) and \([a'_1,\ldots,a'_N] \) for timestep \( t \), ATT-MADDPG considers \( K \) combinations of possible action-vector \( a_{t-1} \), and obtains the corresponding \( K \) Q-values. Wang, Everett, and How (2020) proposed R-MADDPG, in which a recurrent neural network is used to remember the last observations in both actor and critic. In this order, they modified the replay memory such that each tuple includes \((o_{t-1},a_{t-1},o_t',r_t',h_{t-1},h_{t-1}')\), in which \( h_t \) the actor network. It was observed that recurrent actor (with fully connected critic) does not provide any better results than MADDPG.

### 2.3.5 Modeling other agent information

A different category of multiagent algorithms consists of the methods that tackle the challenge of modeling other agent informations. The first complete attempts of
reasoning on the other agents’ information have been performed from Su, Adams, and Beling (2020) and Jiang et al. (2020).

### Counterfactual multiagent Reinforcement Learning with Graph Convolution Communication

This algorithm extends the COMA architecture presented in section 2.3.4. Each agent is represented by a node in the graph, and edges are defined by metrics that measure the relationship (closeness) between agents. They employ a one-hop convolution process, which enables an agent to communicate with its first-order neighbors. Each agent employs a multi-head attention layer to extract relevant information of its neighbors. Also, they utilize recurrency, as a top layer in their architecture, to retrieve latent features from convolution layers as the input and outputs agent’s policy $\pi_n$.

### Graph Convolution Reinforcement Learning

This algorithm is instantiated based on deep Q-networks, and it shares weights among all agents, making it easy to scale. Initially, the observation map of each agent passes through a CNN to the first encoding layer, retrieving the first feature vector. To ensure cooperation, every $t$ timesteps, they concatenate the feature vectors into a Feature Matrix and save the current adjacency matrix that captures the neighbor agents. Thus, the Feature Matrix omits the information of the agents that are not located closely, and through a multi-head attention model, it checks which neighbors are going to cooperate. To keep the attention weights consistent for a short time, they apply KL divergence between the current and subsequent distribution. The experience replay is enhanced to keep the current adjacency matrix. They observed that the Q function is difficult to converge since the graph may change quickly while the agents move out further or closer to others. To overcome this learning difficulty, they maintained unchanged the adjacency matrix in two successive timesteps when they compute the Q-loss in training.

### 2.4 Graph Neural Networks

In coordination games, the relation of the agents is arbitrary and mainly depending on the position of the agents, which is dynamic during the training process. The previous years, the most common approaches for building learning blocks were the fully connected layers (Rosenblatt, F., 1961), the convolutional layers (Fukushima, K., 1980) and the recurrent layers (Elman, J. L., 1990) (see figure 2.2).

In the FCN, the relations between the entities (individual units) in the network are all-to-all (all units in layer i are connected to all units in layer j), and the rules are specified by the weights and biases. Thus, the implicit relational inductive bias in a fully connected layer is very weak: all input units can interact to determine any output unit’s value independently across outputs. Regarding CNN, the entities are usually individual units such as grid elements (pixels), but their relations are sparser. In contrast to fully connected layers, convolutional layers impose some important relational inductive biases: locality and translation invariance. These biases are very effective for processing natural image data because there is high covariance within local neighborhoods, decreasing with distance, and because the statistics are mostly stationary across an image. Finally, in RNNs, the inputs and the hidden states at each processing step are the entities, and there is a relation between two
consecutive hidden states. The rule for combining the entities takes a step’s inputs and the hidden state as arguments to update the hidden state. Thus, considering the recurrent layers, the imposed relational inductive bias is the temporal invariance (similar to a CNN’s translational invariance in space).

A multiagent system constitutes a complex real-world model, which requires high coordination between the individual agents in the scope of cooperation games. It reaps the benefits of arbitrary relations among them, in contrast with Convolutional Neural Networks that model spatial relations and Recurrent Neural Networks that model time relations. Thus, the introduction of graph learning blocks bridges the theoretical gap of modeling agents’ relations.

### 2.4.1 Graph Representation Learning

GNNs use neural message passing in which vector messages are exchanged between nodes and updated using neural networks (see figure 2.3). In this way, they
create more sophisticated hidden representations of nodes that exploit the structural
topology of their neighborhood and are capable of generalizing in unseen testing
nodes. Particularly, during each message-passing iteration in a GNN, a hidden em-
bedding $h_u$ corresponding to each node $u \in V$ is updated according to information
aggregated from $u$’s graph neighborhood $\mathcal{N}(u)$. This message-passing update can
be expressed as follows:

$$
\begin{align*}
    h_u^{(k+1)} &= \text{UPDATE}^{(k)}(h_u^{(k)}, \text{AGGREGATE}^{(k)}(h_u^{(k)}, \forall u \in \mathcal{N}(u))) \\
    &= \text{UPDATE}^{(k)}(h_u^{(k)}, m_{\mathcal{N}(u)}^{(k)})
\end{align*}
$$

(2.27)

The aggregate function has input the set of embeddings of the neighbor of node
$u$. This aggregation produces the message $m_{\mathcal{N}(u)}$. Subsequently, the Update function
receives as inputs the message $m_{\mathcal{N}(u)}$ and the previous embedding $h_u$ of node $u$ to
generate the final embedding. The latter holds structural information on the graph
since after $k$ iterations of GNN message passing, the embedding $h_u$ of node $u$ might
encode information from all the nodes in $u$’s $k$-hop neighborhood. Besides the ca-
pability of identifying different motifs, GNN node embeddings are feature-based in
the sense that they encode information about all features in a $k$-hop neighborhood.

The main differences in the various GNN algorithms are based on the different
implementations of the aforementioned Aggregate and Update function. In a simple
general form, the node embedding is calculated by:

$$
    h_u^{(k)} = \sigma\left(W_{\text{self}}^{(k)}h_u^{(k-1)} + W_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(u)} h_v^{(k-1)}\right)
$$

(2.28)

where $W_{\text{self}}^{(k)}, W_{\text{neigh}}^{(k)} \in \mathbb{R}^{d(k) \times d^{(k-1)}}$ are trainable parameter matrices and $\sigma$ denotes an
element-wise non-linearity. The algorithm first sums the messages incoming from
the neighbors, then combines the neighborhood information with the node’s pre-
vious embedding using a linear combination and finally, applies an element-wise
non-linear operation.

In this thesis, we implied two different types of GNNs:

1. Graph Convolutional Networks (Kipf and Welling, 2016)
2. Graph Attention Networks (Veličković et al., 2018)

**Graph Convolutional Networks**

GCN alters the aggregate function, adding self-loops in the message passing. The
previously hidden embedding of a specific node is considered one of the neighbors’
embeddings, and the final embedding is calculated with simply the aggregate func-
tions. Simplifying the message passing in this way can often alleviate overfitting.
However, it also severely limits the expressivity of the GNN, as the information
coming from the node’s neighbors cannot be differentiated from the information
from the node itself.

Furthermore, GCN performs a symmetric normalization to the adjacency ma-
trix (Kipf and Welling, 2016). In this way, the entries of the adjacency matrix, i.e.
the edges $(i,j)$ between the nodes/ agents $i$ and $j$, correspond to transitional prob-
abilities between them. Most of the aggregation functions perform a normalization
on the incoming messages (aggregate information from node’s neighborhood), presuming that nodes’ degrees vary and normalization is required to not add bias in the networks’ hubs.

Graph Attention Networks

Velisˇ covi´c et al. (2018) integrated the attention models in the GNN framework by integrating attention weights on each neighbor, thus varying the importance of the neighbors based on these values.

Self-attention is an attention mechanism relating different positions of a single sequence to compute a representation of the same sequence (for more details see Vaswani et al. (2017)). Every input must have three representations: key, query, and value, which are obtained via multiplication of the inputs with a set of weights for keys, queries, and values respectively. In a neural network setting, these weights are usually small numbers, initialized randomly using an appropriate random distribution like Gaussian (Wellmer, 1998), Xavier (Sirignano and Spiliopoulos, 2019) and Kaiming distributions (He et al., 2015) before training. Initially, using self-attention, we retrieve attention scores for each input, applying a dot product between the query of one input and the keys of the rest of the inputs, including itself. This operation is known as dot product attention, one of the several score functions that have been researched. Based on the number of representations that we have, we obtain an equal number of attention scores. Subsequently, we apply a softmax function on the attention scores, which are next multiplied with their corresponding values (of the input representation). The output value corresponds to the sum of the weighted values and constitutes the query representation of an input, interacting with all other keys, including itself.

Considering our multiagent setting with five agents, the self-attention mechanisms receives as input the observation of an agent and outputs a representation of this observation that include part of the observations of the interacting agents. Thus, the aggregation messages of the Graph Attention Network are calculated as:

$$m_{N(u)} = \sum_{v \in N(u)} \alpha_{u,v} h_v$$  \hspace{1cm} (2.29)

where $\alpha_{u,v}$ denotes the attention weights on neighbor $v \in N(u)$ and are defined as:

$$\alpha_{u,v} = \frac{\exp(a^T [Wh_u \oplus Wh_v])}{\sum_{v' \in N(u)} \exp(a^T [Wh_u \oplus Wh_{v'}])}$$  \hspace{1cm} (2.30)

where $a$ is a trainable attention vector, $W$ is a trainable matrix and $\oplus$ denotes the concatenation operation.

Skip Connections

A core issue in GNNs constitutes the over-smoothing, which occurs when node-specific information is lost after several iterations of GNN message passing. In these cases, the aggregated information from the node neighbors during message passing begins to dominate the updated node representations. A natural way to alleviate this issue is to use skip connections to directly preserve information from previous rounds of message passing during the update step. This enhancement proposed from the Hamilton, Ying, and Leskovec (2018) on his GraphSAGE framework.
These concatenation and skip-connection methods can be used in conjunction with the vast majority of other GNN update approaches. In this thesis, we use skip connections combined with the Graph Attention Network where the aforementioned update function is:

\[
\text{UPDATE}_{\text{concat}}(h_u, m_{N(u)}) = \left[ \text{UPDATE}_{\text{base}}(h_u, m_{N(u)}) \oplus h_u \right]
\] (2.31)

Thus, we perform a simple concatenation of the base update function with the node’s previous-layer representation to disentangle information during message passing by separating the information coming from the neighbors with the current representation of each node.
Chapter 3

Methods and Experiments

In this chapter, we describe the methods and algorithms used for the conducted experiments. In section 3.1 we explain the environment that we used along with minor modifications on the reward function that empirically enhanced the learning procedure. In section 3.2 we describe the mixing strategy multiagent algorithms that we implemented. Finally, in section 3.3 we present the actor-critic corresponding algorithms.

3.1 Multiagent Particle Environment

It is an archive of different multiagent environments (Lowe et al., 2020). In our experiments, we applied the scenario of Cooperation Navigation (or simple-spread), which is illustrated on figure 3.1. In this environment, \(N\) agents must cooperate through physical actions to reach a set of \(L\) landmarks. Agents observe the relative positions of other agents and landmarks and are collectively rewarded based on the proximity of any agent to each landmark. In other words, the agents have to ‘cover’ all of the landmarks. Further, the agents occupy significant physical space and are penalized when colliding with each other. Agents learn to infer the landmark they must cover and move there while avoiding other agents. This environment constitutes a cooperation game that, despite its simplistic structure, remains one of the most challenging tasks due to the problem of agent coordination (for details, see section 2.2.4).

\[\text{Figure 3.1: Cooperation Navigation Environment}\]
Furthermore, the environment structure becomes flexible on the type of the action space. The existing configurations permit both a discrete and continuous action space. In the first case, the agents have five available actions (left, right, up, down, stay still), and the actual movement consists of the performed action multiplied by the agent’s current velocity. The velocity is affected by the proximity of the agent to its neighbors. In the continuous action space, the agents output two actions \( a_i \in [0; 1] \), which corresponds to their movement in the \( x \) and \( y \) axis. The chosen action is also multiplied by the current agents’ velocity.

To boost the learning process, we perform a modified version of the current environment:

1. The reward obtained from the proximity of an agent to a landmark augmented to \(+100\)
2. An extra penalty is imposed when an agent moves far away from the landmarks
3. The episode finishes when an agent gets away from the landmark space by updating the donecallback of the Gym library where the environment is built

It should be mentioned that the training reward of the models applied to this task depends substantially on the instantiation of the agents on the map. The task complexity is aligned with the initial positions of the agents and may need a higher level of coordination. The map creation is a randomized process, and we expect high variance between subsequent episodes in all the models. For instance, if the same landmark is close to two different agents, they have to coordinate and decide which agent will approach this landmark and which agent will move towards the next closer free landmark.

### 3.2 Mixing Strategies

The VDN mixing algorithm is built upon value-based algorithms. In this thesis, we implemented alterations of the deep q-learning method as depicted on figure 3.2.

The simplistic independent q-learning model (see section 2.3.4) may achieve competitive performance in non-cooperative games but cannot decompose the dynamics of a coordination task. As mentioned in section 2.2.4, these tasks require modeling of other agent information to avoid the moving target problem of the non-stationary environment (for more details, see section 2.3.3). Motivated by the limitations of IQL architecture, we incorporate a GNN as an initial encoder of the agents’ observation maps. In this way, the latent outputs of the graph will include the relational dynamics between the agents in a hidden \( k \)-dimensional vector for each node. Subsequently, we use a Lambda layer to propagate each node’s \( k \)-dimensional vector in \( N \) individual DQN architectures. The model is differentiable and, thus, is trained altogether using back-propagation. The actions are retrieved from the individual Q-utility functions combined based on the VDN mixing strategy (see section 2.3.4).

In these experiments, we consider that each agent interacts only with the two closest agents to him. Thus, in every forward pass of the model, the adjacency matrix contains in each row zero entries, except two that corresponded to the two agents closer to this node/agent. It should be mentioned that the number of agents in the environment was set to \( four \).

Furthermore, the Cooperation Navigation scenario is configured as originally published, omitting the extra penalty of “leaving the scene” and restricting the reward to \(+10\) when the agents step on a landmark.
3.2. Mixing Strategies

The implemented variants of this backbone architecture are the following:

- **IQL**: Agent networks consist of independent DQNs, and the final Q-function is combined with the VDN mixing strategy.

- **R-IQL**: Agent networks consist of independent R-DQNs, implemented either with an LSTM layer or with self-attention. Then, the final Q-function is combined with the VDN mixing strategy.

- **GCN-DQN**: A Graph Convolutional Layer encodes the individual agents’ observation. Subsequently, the latent variables are propagated through $N$ independent DQNs, and, as previously mentioned, the final Q-function is combined with the VDN mixing strategy.

- **GAT-DQN**: Similarly to the previous architecture, this model is composed of $N$ independent DQN and the VDN mixing strategy. However, the initial GCN is replaced with a graph attention network.

- **GAT-DDQN**: In the last architecture of the mixing strategy experiments, the previously independent DQN networks were replaced with Dueling DQN. This architecture is illustrated in the right figure of 3.2.

On the GAT-DQN architecture, we try two different variants: one with entirely independent DQNs, and a second one where all agent networks share parameters. Parameter sharing has the restriction that can only be applied to cooperative environments, and that can only work for "homogeneous" sets of agents. Since none of
these restrictions are applied in our environment, we could perform this enhancement.

3.3 Actor Critic Strategies

In these experiments, we adopt the centralized training - decentralized execution framework of the MADDPG algorithm (see section 2.3.4). We implemented two different actor-critic algorithms that will be analyzed in the following subsections.

3.3.1 Experiments on the Multiagent Algorithm for Deep Deterministic Policy Gradients

First, the dynamics of the algorithm were tested on the original environment increasing the cardinality of the agents. Then, we implement the modifications mentioned in section 3.1 and analyze the obtained results. Namely, we examine the training and evaluation reward with different hyperparameters, such as exploration rate, learning rate, and the number of neurons.

3.3.2 Experiments on the Multiagent Algorithm for Deep Deterministic Policy Gradients with Graph Attention Layers

To overcome some of the limitations of the MADDPG algorithm, we implemented the Multiagent Algorithm for Deep Deterministic Policy Gradients with Graph Attention Layers (in the rest of this thesis, we will call it MAGAT for simplicity), which is a modification of the MADDPG. The problem of coordination motivated us to incorporate the graph network to encourage the relations between the agents. First, two consecutive Graph Attention Layers are used to encode the critic’s input. Thus, each node/agent of the graph has as input its observation, and the action performed conditioned on this observation. We represent the previous inputs in a new $k$-dimensional space through a forward pass, where the proximity of the agents is exhibited through the similarity of their vector representations. The final network is depicted in figure 3.3.

Thus, we experimented with the multiagent GAT architecture (as mentioned earlier) on the modified environment of the Cooperation Navigation scenario. Specifically, we analyzed if a graph neural network can encourage the coordination between the agents by inducing relational bias on the network. In this thesis, we will discuss the results obtained, applying various hyperparameters, such as exploration rate, learning rate, number of neurons, and application of skip connections between the graph layers.

Hyperparameter Search

In both the algorithms mentioned above, we observed that the models’ performances were highly influenced by the exploration strategy implemented. As mentioned in section 2.1.6, the DDPG algorithm uses the Ornstein-Uhlenbeck Process to encourage exploration, where noise is added to the actor’s continuous policy. The parameters that we experimented with were the initial noise added, the noise decay, the minimum allowed noise, and the episode on which the noise would reach its minimum rate (i.e., initial noise values: 0.1, 0.2, 0.5, noise decays: 0.999 and 0.99999, minimum noise: 0.01 and 0.05, episode with minimum noise: 10K and 20K).
Then we applied different learning rates to both the actor and the critic. Since the critic’s model has more parameters than the actor’s, the critic’s learning rate was always equal or smaller than the actor’s. The different values applied were: \(1e^{-2}\), \(1e^{-3}\), \(1e^{-4}\) and \(1e^{-5}\).

Finally, we experimented on different numbers of neurons on the actor and critic model. Independently of the graph neural network, each model consists of two hidden layers. The last layer has always had the neurons of its previous one. We experimented with values: critic: 256 & actor: 128, critic: 512 & actor: 256 and critic: 700 & actor: 500.
Chapter 4

Results

In this chapter, we present the results of the conducted experiments based on the methods described in chapter 3. The performance of the original environment with mixing-strategy algorithms is presented in section 4.1.1 and the corresponding behavior of the actor-critic algorithms in section 4.1.2. The robustness of these approaches in a varying number of agents is discussed. It leads to the section 4.2, where the actor-critic methods on the updated environment are analyzed. Section 4.2.1 presents the efficiency of the MADDPG algorithm, examining different hyperparameters (i.e., exploration, learning rate, and the number of neurons on the actors’ and critics’ models). Finally, in section 4.2.2 we conducted a critical analysis on our new approach of the enhanced critic structure with the graph neural network.

4.1 Original Environment

Initially, we configured the environment as originally published in Lowe et al. (2020) with a $+1$ reward scalar and a $-1$ collision penalty. Then, we set the number of agents equal to four and used a seed equal to three in all the experiments to obtain a comparable performance.

4.1.1 Results on Mixing Strategy Algorithms

In this section, we will present the asymptotic performance of the algorithms presented in section 3.2. These algorithms are using the VDN mixing algorithm to combine the individual action-value utility functions. In table 4.1, we sum up the testing reward of the agents after 60 thousand training episodes during 25 steps. The models implemented are described in section 3.2. Since all the network backbones are implemented with a deep q-network and VDN mixing policy, the action input space is configured to be discrete. The DQNs used prioritized experience replay with a capacity equal to $1e6$, which was augmented to include the adjacency matrix on the algorithms that include graph layers.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>IQL</th>
<th>GCN-DQN</th>
<th>GAT-DQN</th>
<th>IGAT-DQN</th>
<th>GAT-DDQN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Episode Reward</td>
<td>-57288.049</td>
<td>-58951.223</td>
<td>-62996.899</td>
<td>-52952.133</td>
<td>-57955.201</td>
</tr>
</tbody>
</table>

Table 4.1: Asymptotic performance of value-based algorithms with VDN mixing strategy in the original environment.

Considering the results on table 4.1, it is evident that neither of the models could learn to accomplish the task. The Value decomposition network was not capable of decomposing the total action-value function since (as mentioned in section 2.3.4) it requires a monotonically mixing of the individual action-value utility functions. The approach of one agent to a landmark increases its utility action-value function
but does not increase the utility functions of the rest of the agents. In particular, we observed that the monotonicity constraint on the relationship between $Q_{tot}$ and each $Q_a$ agent-utility function was not fulfilled:

$$\frac{\partial Q_{tot}}{\partial Q_a} \not\geq 0, \forall a \quad (4.1)$$

It is evident that we could not detect any comparable results on the algorithm’s performance. The differences in the obtained asymptotic rewards are due to the different structures of the models and the corresponding layer initializations.

**Recurrent Independent Q-learning**

Acknowledging the monotonicity restriction of the VDN algorithm, we attempted to enhance the dynamics of the model by producing more complex action-value functions that could be composed and back-propagated successfully in the individual agents. Thus, we provided temporal experiences to the agents, meaning a history of four subsequent transitions instead of exploiting only one. This modification does not guarantee convergence to an optimal solution since it breaks the constraints of the coordination problem. However, we explore if this additional information may yield a better solution. Practically, we used the R-IQL model described in section 3.2 and decreased to two the number of agents in the environment.

Figure 4.1 illustrates the training and evaluation reward of the independent q-learning algorithm, enhanced with a recurrent layer and aggregated with the VDN mixing strategy. In this experiment, we trained our model with only two agents. In contrast to the previous models, the training reward on the y-axis corresponds only to one agent; thus, the joint reward is the double value. Also, we depict the corresponding exploration rate applied to the E-greedy algorithm.

![Figure 4.1: Analysis of the training and evaluation reward (Left panel, y-axis) and of the e-greedy exploration rate (Right panel, y-axis), for two agents, of the recurrent independent q-learning algorithm with the VDN mixing strategy](image)

Regarding the training rewards, shown in figure 4.1, the model could not decompose the total q values to the individual utility functions. Specifically, the training reward persisted the same performance during the whole learning procedure. Strangely, the high exploration rate of the e-greedy strategy did not change the obtained results. This indicates that the current experiences of the agents could not be adequately exploited. Thus, the performed random actions added transitions that
the model could not use to learn, resulting in poor policies that could not solve the coordination task. Thus, the addition of the recurrent layers did not improve the learning procedure.

4.1.2 Results on Multiagent Algorithm for Deep Deterministic Policy Gradients

The MADDPG algorithm was tested initially in the original environment with two and five agents. In these experiments, the action space is still discrete, but the exploration strategy applied is the Gumbel Softmax (for details see 2.1.6) instead of the E-greedy method used on the value-based algorithms. In the two agents’ experiment, they were successfully trained to recognize the landmarks and move towards them (see left figure 4.2). However, when the complexity of the environment increases to five agents, the algorithm’s performance notably decreases (see right figure 4.2).

Regarding the training rewards with five agents, as shown in the right figure 4.2, it is evident that the actor and policy networks did not converge to a good policy. Despite a throughout analysis of different model parameters, the agents’ performances were poor after 60000 episodes.

This led us to conclude that the low rewarding of the agents when they stepped on a landmark was an obstructing factor in the training procedure. The model could not correlate the observations of each agent, which consist of the relative positions of both the landmarks and the other agents, to an optimal action. We observed that the agents, trained to avoid any collision between them, also learned to distance themselves from other objects (both landmarks and other agents).

To ease the training process, we increased the reward that the agent obtained stepping on the landmark from $+1$ to $+100$ and repeated the experiments with four agents. Furthermore, we modified the environment action-space configuration from discrete to continuous action space. Thus, the allowed actions were limited to two (continuous movement to the x-axis and y-axis, respectively). Additionally, we used the Ornstein-Uhlenbeck Process exploration policy (for details, see 2.1.6) to be aligned with the continuous policy distributions. The training reward of the MADDPG algorithm in the environment with the updated reward is illustrated in figure 4.3. The corresponding individual policy and critic losses are shown in the figure 4.4.
In figure 4.3, the model with four agents has a volatile performance, which can be partially explained from the initial agent initializations. It should be mentioned that we used a rolling window equal to 100 to smooth the final figure illustration. In this case, the agents’ critic and action models have learned significantly divergent policies, which leads to the heterogeneous performance of the agents.

In figure 4.4, it is clearly shown that the actor and policy networks of each agent display a dissimilar course. This behavior was also verified during execution time, where only one agent has learned to act optimally, and the rest of the agents moved outside of the frame. Thus, the positive training reward corresponds to the optimal performance of only one agent, whereas the rest did not learn to approach the landmarks. Also, it should be noted that the learning procedure of some agents is highly coarse and does not smoothly converge to a specific policy.

Taking a closer look at the loss functions, we associate the spikes to discovering the wrong estimated states. This behavior is expected and necessary for the agents to learn. However, the agents continuously change their policy in different timesteps, which complicates the adjustment to the non-stationary environment. The sampled gradients correspond to erroneous policies, and the learning procedure is extremely noisy. In particular, the right figure illustrates that most of the agents are learning
better estimations for these newly discovered states after a few episodes (see blue lines). However, others do not converge to a good policy (see pink line).

The MADDPG algorithm consists of $N$ actor-critic networks that do not share any weight parameters. This characteristic makes the model flexible since it can be applied in environments with heterogeneous agents. However, our coordination task leads to miscellaneous agent policies.

Furthermore, we analyzed the performance of the MADDPG algorithm with a different number of neurons on the actor and policy networks. In particular, we experimented with neurons equal to 64, 128 and 256. The models were tested on the same environment (i.e., seed equal to three), and the average reward of 50 consecutive episodes is depicted on table 4.2.

<table>
<thead>
<tr>
<th># Neurons</th>
<th>Rewards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critic: 128, Actor: 128</td>
<td>-992.869</td>
</tr>
<tr>
<td>Critic: 256, Actor: 64</td>
<td>369.367</td>
</tr>
<tr>
<td>Critic: 256, Actor: 128</td>
<td>490.624</td>
</tr>
</tbody>
</table>

Table 4.2: Asymptotic performance on variants of MADDPG model configuration with agent rewarding equal to +100

The number of neurons in the actor and critic networks substantially affect the performance of the model. The best results are obtained with 256 neurons on the critic network and 128 neurons on the actor’s one. Empirically, although the cooperative reward was positive, the behavior of only one agent was optimal. Regarding the models with fewer network parameters (i.e. simple models with fewer neurons), they are bound to underperform in the coordination task. This is happening because these models are not complex enough (i.e. exhibit high bias) to capture the dynamics of this task.

### 4.2 Updated Environment

The fact that a few agents were moving out of the frame even after 60K episodes motivated us to add a supplement penalty on the agents, as implemented in this open source project), and finish the episodes every time that this condition met. Furthermore, the maximum number of steps per episode increased from 25 to 35. If not stated otherwise, the number of agents in all the following experiments is set to five. In this modified final environment, we performed an evaluation and hyperparameter analysis on the actor-critic based algorithms.

#### 4.2.1 Results on Multiagent Algorithm for Deep Deterministic Policy Gradients

There is a variety of different modifications and choices of hyperparameters to evaluate. In this section, we will present an analysis on the noise initialization applied for the Ornstein-Uhlenbeck Process, on the number of neurons on the actor and critic model, and for their corresponding learning rate.

**Analysis on Noise Exploration**

The different noise initializations of the Ornstein-Uhlenbeck Process that we explored are depicted in figure 4.5. The current noise is multiplied with a constant noise decay in every environmental step. Specifically, we applied initial noises equal to 0.1, 0.2, and 0.5. Also, we configured the noise decay to 0.999, making the exploration approximately negligible after 4545, 5286, and 6012 episodes, respectively.
Chapter 4. Results

The learning procedure exhibits a robust performance in all the different tested parameters. The higher noise initialization of 0.5 encourages a robust state-action exploration on the first episodes, which justifies the lower initial training rewards on the yellow line. Also, the excessive exploration stores into the buffer many negligible transitions that do not essentially contribute to the learning process. Thus, the model is delayed to exploit the "good" and "bad" actions. Based on our results, the noise initialization of 0.2 accomplished an optimal exploration-exploitation trade-off since it resulted in a higher slow convergence rate. Furthermore, the experiment with initial noise equal to 0.1 performs slightly worse. Finally, the estimated q-values are converged to the true values after 36000 episodes in the blue and red lines (lower exploration) and after approximately 45000 in the yellow line (higher exploration). It is verified during execution time that the convergence is not an artifact of low exploration but due to the discovery of optimal actions.

Analysis on Number of Neurons

The performance of the MADDPG algorithm in the original environment was notably affected by the number of neurons of the actor and policy networks. This motivated us to perform a new hyperparameter search in this more robust environment. In particular, we experimented with larger networks, where the number of neurons varies from 256 to 700 as illustrated in figures 4.6. In these experiments, the learning rate for the critic was set to $10^{-3}$ and for the actor to $10^{-2}$. Finally, the initial exploration noise was set equal to 0.1.

In contrast to the corresponding experiments on the table 4.2, in which we had a varying number of neurons, the updated environment demonstrated a more robust performance towards the number of neurons per layer on the critic and actor networks. Even if the larger models (i.e., models with a higher number of neurons) presented a slight degradation at the beginning of the training, they converged faster to a good policy (see the yellow line on left figure 4.6). This initial low performance is expected since a more complex model needs more time to be tuned. On the other hand, the smaller model with 256 critic neurons and 128 actor ones (see blue line on the right panel of figure 4.6) demonstrates higher variance (i.e., noisy gradients) and slightly worse performance. We used the network configuration with 700 critic neurons and 500 actor neurons in the rest of the experiments.
4.2. Updated Environment

Analysis on Different Learning Rates

The most crucial hyperparameter to tune for a neural network to achieve good performance is the learning rate. Thus, the estimation of an adequate rate is a crucial factor of the training procedure. In all the experiments, we use the Adam optimizer (for more details, see (Kingma and Ba, 2017)). In contrast to the Stochastic Gradient Descent, it maintains adaptive learning rates for different parameters from estimations of first and second moments of the gradients.

In the subplots of the figure 4.7, we show the training reward and the corresponding critic loss function of the models with different learning rates. In particular, we experimented with values: $1 \times 10^{-2}$, $1 \times 10^{-3}$, $1 \times 10^{-4}$ and $1 \times 10^{-5}$ on the actor and critic networks. Furthermore, the number of neurons in all the experiments is set equal to 700 for the critic network and 500 for the actor’s one. Also, the initial Ornstein-Uhlenbeck noise was set to 0.1.

Even though Adam’s optimizer is applied on both actor and critic models, the learning rates substantially influence the model’s performance.

Empirically, updating the gradients in parallel with the loss functions updates result in too noisy directions and difficulties in convergence. In particular, the sampled batches will calculate different gradient directions, and the loss would significantly increase. Thus, a high learning rate will not help the model to converge.
On the other hand, setting the learning rate lower will lead to a more conservative direction, preventing the erroneous gradients from critically affecting the entire optimization problem. Finally, the learning process will be slower, but we will manage to converge in the end.

Observing the figure 4.7, this hypothesis is verified since the lower learning rates performed better, producing fewer variations and leading to smoother convergence. The configuration with $1e^{-4}$ on both networks resulted in the fastest convergence. It should be mentioned that the experiment with the higher learning rates (see blue line) were previously analyzed with different noise initialization. However, the learning rate exploration led us to conclude that a lower rate performs better on this coordination task. The rewards obtained with a higher learning rate value (red and blue line) illustrate a slightly worse performance.

**Analysis on Collision Rate**

In the updated environment, the collision rate penalty is stable and equal to $-1$, in contrast, to the high agent rewarding ($+100$). In this analysis, we are trying to explore if the agents can learn to avoid collisions with each other even with such a low penalty. Thus, to capture the collision rate dynamics, we group the aggregated number of collisions in bins of 100 episodes and calculate the average rate. In figure 4.8, we illustrate the total number of collisions (i.e., the sum of collisions for all the agents) per episode.

![Total Collision Rate](image)

**Figure 4.8:** Number of collisions between the agents after 60000 episodes averaged in bins of 100 episodes

The number of collisions between the agents essentially drops after approximately 20000 episodes. Although the acknowledgment of the collision penalty occurs noticeably after many episodes, we can argue that the small penalty scalar ($-1$) does not impede the agents from learning not to collide. We justify the remaining number of collisions to the non-trivial landmark initialization. In particular, two
landmarks can be partially covered, leading to agent collisions when they intend to approach them simultaneously.

**Robustness**

The robustness of the MADDPG algorithm on the updated environment was assessed through the training of the model with three different seed values as illustrated on figure 4.9.

**Figure 4.9:** Episode reward with three different seeds (Left panel, y-axis) and the corresponding critic loss (Right panel, y-axis) after 60000 episodes (x-axis)

DDPG algorithm is characterized by an inherent instability due to its sensitivity to hyper-parameters and propensity to converge to poor solutions or even diverge (Matheron, Perrin, and Sigaud, 2019). Although using only three seed values does not result in statistically significant conclusions, our experiments demonstrate a robust performance through the learning procedure. Namely, in all the experiments, we use a scalar value equal to three since it demonstrates a slightly faster convergence to the optimal joint policy.

**Comparable Analysis on the Number of Agents**

In this subsection, we illustrate the performance of the MADDPG algorithm in the cooperation navigation scenario configured with two to seven agents (see figure 4.10).

**Figure 4.10:** Episode reward of the final environment with 2-7 agents (Left panel, y-axis) and the corresponding critic loss (Right panel, y-axis) after 60000 episodes (x-axis)
Regarding the training rewards, in all the experiments, the agents learned a good policy. Thus, we successfully tackle the additional complexity imposed from the higher agent cardinality through the new environment configuration. The differences in the training rewards correspond to the accumulated distances of the agents from the landmarks. Since the agents are instantiated in random places away from the landmarks, it is expected that a higher number of agents correspond to a higher accumulated penalty. Hence, the training rewards related to the number of agents are expected. Empirically, the number of training episodes was not sufficient for this task. The agent policies of the settings with more agents are still improving and are not optimal. However, the computational limitations did not allow us to proceed with the further training of the models.

4.2.2 Results on Multiagent Algorithm for Deep Deterministic Policy Gradients with Graph Neural Networks (MAGAT)

In this section, we will analyze the performance of the MAGAT model that we introduced in section 3.3.2. We will examine whether the addition of the graph attention layers results in a better representation of the relational dynamics between the agents and leads to an optimal joint policy.

Comparable performance of Actor-Critic algorithms

In this subsection, we compare the training reward obtained from the MADDPG algorithm and our MAGAT model. The number of the parameters of the original MADDPG algorithm and the MAGAT variation is shown in table 4.3. We observe that the two graph attention layers approximately doubles the number of parameters of the model. Thus, the optimization of the enhanced algorithm constitutes a more challenging problem.

<table>
<thead>
<tr>
<th>models</th>
<th>MADDPG</th>
<th></th>
<th>MAGAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>no params</td>
<td>89802</td>
<td>195097</td>
<td>89802</td>
</tr>
</tbody>
</table>

*Table 4.3: Number of actor and critic model parameters for the MADDPG and MAGAT algorithms*

In figure 4.11 are illustrated the training rewards and the corresponding critic losses for the MADDPG (blue line) and the MAGAT (red line) respectively.

In our model, the initial degradation of the training reward is effectively omitted (i.e., the agents learn fast to remain (be detained) in the mainframe of the environment, thus near the landmarks). This can be partially attributed to the use of the graph layers. However, the coordination problem is not adequately tackled since the network’s behavior remains unchanged and slowly decreases after the first 10K episodes. We argue that the critics’ networks converged to sub-optimal training parameters, a fact that is related to the hyperparameter analysis that we performed.

We observed that training attention models could be unstable, and careful tuning of the initial learning rate, warm-up, and normalization is crucial. We hypothesize that the unsuccessful performance of the MAGAT model is due to the more difficult optimization problem, in combination with the proven instability of the DDPG algorithm. The tuning of the environment configurations aimed to encourage the
4.2. Updated Environment

**Figure 4.11**: Comparison of the performance between the original MADDPG algorithm (blue line) and our MAGAT variation (red line) after 60000 episodes.

The learning procedure of the MADDPG algorithm, a fact that did not happen with the MAGAT variation.

We hypothesize that the over-smoothing of the hidden variables restricts the highly divergent actions of the agents in the training process. This step is crucial for the exploration of the environment. Thus, the experience replay holds similar transitions and continuously exploits experiences that reproduce the same knowledge.

**Influence of Exploration Strategy**

We investigated different decay exploration rates, besides the various noise initialization trials, to ease the training instability from the attention layers. After the first $10^5$ episodes, the noise rate remained constant to a minimum value equal to 0.001 in all the experiments. The most stable experiments were obtained with the parameters stated in Table 4.4.

<table>
<thead>
<tr>
<th></th>
<th>Exp 1</th>
<th>Exp 2</th>
<th>Exp 3</th>
<th>Exp 4</th>
<th>Exp 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial noise</td>
<td>0.5</td>
<td>0.1</td>
<td>0.1</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>noise decay</td>
<td>0.999</td>
<td>0.999</td>
<td>0.9999</td>
<td>0.9995</td>
<td>0.999</td>
</tr>
</tbody>
</table>

**Table 4.4**: Exploration Parameters that used for the analysis of different exploration strategies on page 4.12.

The performance of the above-configured experiments is shown in Figure 4.12. In these experiments, the learning rate is set equal to 0.001 in both actor and critic networks. The critic uses two graph layers, and the initial observations are concatenated to the graph outputs, using skip connections. Furthermore, the number of neurons is set to 64, 700, 500 for the graph network, the critic’s network, and the actor’s network, respectively.
Chapter 4. Results

The highest training rewards were obtained with an initial noise percentage equal to 0.2. Although the models showcase an initial improvement on their training rewards, they did not manage to tackle the agents’ coordination problem. The enormous loss function in the right panel led us to conclude that the model cannot exploit the batched samples successfully. Thus, the error between the expected q-values and the sampled batches does not decrease.

The error that we try to minimize is the Bellman residual (i.e., the difference between the two sides of the equation 2.3), that is, the difference between the expected value of our network compared to the sampled batches of the replay buffer. In the case of RL, the gradients of this loss function are even noisier and more stochastic than in a Supervised learning setup since the error function that we try to optimize is not stable. To address this issue, state-of-the-art algorithms include two q-value estimators to prevent the instability of the loss function. In this way, the stochastic gradient will follow the same loss and be iteratively modified until convergence.

The behavior of the error function can be partially attributed to the high learning rate that we applied to these experiments. This rate leads the sampled batches to calculate different gradient directions and significantly increases the critic’s loss. Thus, we can not present a practical assessment of the exploration’s influence on the training process since the stored transitions are not exploited accurately.

Influence of Skip Connections

A common obstacle on the learning procedure with GNNs constitutes the loss of the individual information of the nodes (for more details, see section 2.4.1). This experiment evaluates the MADDPG algorithm’s performance with two consecutive graph attention layers incorporated into the critic’s network. In the blue line of figure 4.13 is depicted the training reward when no skip connection is applied, and in the red line, is shown the standard model, with skip connections, explained in section 3.3.2.

The mixing of the individual networks with the rest of the agent information homogenizes their observations, limiting the divergence of their actions. Consequently, the buffer will be filled with similar and not valuable experiences. In our case, the learning process got stacked into a fixed point corresponding to a poor solution (i.e., agents remain in the primary environment frame). We demonstrate that even if the skip connections work towards the solution of the high over smoothing, their application augments the state space significantly. This has as a drawback that the
4.2. Updated Environment

complexity of the input space explodes, and in comparison to the attention, layers result in a hard optimization problem.

Furthermore, removing the skip connection architecture led to the explosion of the critic loss (see blue line on figure 4.13). We hypothesize that we overestimated the Q-values, which leads to the increase of the loss value. As explained in section 2.1.4, the optimistic action value estimations are propagated through Bellman’s equation and equally affected the policy networks.

The slightly better performance of the architecture without the skip-connection contradicts its critic loss behavior. However, neither of the models learns a good policy. Thus, we conclude that the reward differences are due to the stochasticity of the initial model configurations. This led the two models to be stacked in sub-optimal policies.

Analysis on the Self-Attention Layers

The motivation behind this experiment was the irregular loss function, depicted in figure 4.13, in the previous experiment. To understand the reasons behind the poor performance of the enhanced model with the graph attention layer, we calculated the percentage of negative weight parameters that are set to zero after the non-linear activation layer (see figure 4.14).

All of the agent’s graph layers propagate most of the receiving information since only a 5% of the neurons produce a negative weight, which is discarded through the RELU activation. A slight increase in the negative weights is observed after the first 24000 steps when the buffer is full. However, we could not extract any meaningful insight into the slowly degrading performance of the model.

It is difficult to comprehend whether attention suffices as a holistic explanation for the model’s poor joint policy. Understanding the intermediate representation of the latent variables is a non-trivial task, and frequently the semantic representation of the transformed space is problematic.

Robustness

The variation of the current results is evaluated through training with different seed scalar values as illustrated in the figures 4.15.
Chapter 4. Results

Figure 4.14: Percentage of negative weight values in the last graph attention layer. Episodes are scaled down by a factor of 60.

Figure 4.15: Episode reward with three different seeds (Left panel, y-axis) and the corresponding critic loss (Right panel, y-axis) after 60000 episodes (x-axis).

Although the training rewards of the three different experiments demonstrate similar behavior, the corresponding loss functions in the blue and red lines are exploding. This indicates that the model is not robust in the different learning parameters. Multiple reasons may explain why this is happening. First, the high learning rates lead to extremely noise gradients that unstabilize the learning process since the actors update their policies too fast. As a result, the critics cannot cope with the continuously moving target estimations. Second, the fine-tuning of the self-attention layers is a challenging task, especially in an RL setting. We believe that a larger number of episodes may lead to more insights into the exploding loss. A reason for this behavior may be the exploration of new states that initially were erroneously estimated. Furthermore, the DDPG algorithm can lead to a deadlock situation in cases that the "good" and "bad" actions are not discovered early enough. In this case, neither the actor nor the critic can evolve anymore, a situation that persists even when the agents are subsequently trained with rewarded samples (Matheron, Perrin, and Sigaud, 2019). In our opinion, the agents discover late adequate coordination policies, which delays the learning procedure and results in the deadlock.
4.2. Updated Environment

situation, where the loss function continues to increase because of the erroneous estimations of the new awarding states.
Chapter 5

Discussion

In this chapter, the results of this thesis are reviewed and are placed into the context of the general field of multiagent coordination problems. This is followed by a discussion of potential future extensions to this research framework.

5.1 Conclusions and Contributions

The coordination task is categorized as the most challenging problem of the cooperation field. Even in human beings, the coordination among them is not an inherent characteristic and is not always succeeded. In game theory, it is demonstrated that the agents frequently ended up to sub-optimal joint actions in the absence of an accurate foresight of other agent movements. Thus, the Nash Equilibrium is attained with less rewarding outcomes.

The value-based algorithms implemented in this thesis (i.e., applying the VDN mixing strategy) have achieved satisfactory performance on cooperation scenarios, such as the StarCraft minigames (Vinyals et al., 2017). Even though this environment consists of a larger state and action space, it does not occult the difficulties imposed from the coordination tasks. In particular, the divergent actions of one agent do not affect the good policies of the rest even if the environmental reward is the same for all. Hence, the variations of the mixing strategy algorithm based on the monotonic mixing of the individual q-functions of the agents did not solve the Navigation Cooperation task.

Our following approach involved the actor-critic state-of-the-art MADDPG algorithm. The original configurations of the Navigation Cooperation scenario could not lead to optimal policies when the agent cardinality increased to four agents. This motivated us to modify the environmental reward function. In these experiments, the agents’ performance trained with the MADDPG algorithm was heterogeneous and highly oscillated.

Thus a most careful look was taken into the definition of the problem to discover a setting that encourages a faster convergence with balanced rewards and penalties. In particular, we encompassed a functionality to terminate the failing trials early and increased the maximum number of steps per episode. We successfully implemented the MADDPG algorithm on the Navigation Cooperation task and gradually incremented the problem’s complexity by increasing the agent cardinality.

In the direction of faster convergence and better coordination, we implemented our MADDPG variation, which incorporated the graph attention layers. From a theoretical perspective, the incorporation of other agent information in the value estimations induced bias to the model. This helps an agent take into consideration the rest before evaluating the value of a state. However, the obtained results were not as good as we initially expected, which is probably due to the complex optimization
problem of the attention layers combined with the instability of the original DDPG algorithm.

5.2 Future Work

We strongly believe that the graph incorporation on the centralized training of a multiagent system can yield a higher degree of coordination between the individual agents. However, the training of such a complex architecture in the multiagent framework is a challenging task. The major complications involve the fine-tuning of the attention model in combination with the additional noisy estimations of the action-value functions.

As future work, we would like to analyze our model’s behavior, applying more robust backbone algorithms that would help focus on the extension of the model with graph attention layers. For instance, recent studies have managed to stabilize the bias overestimation of the DDPG algorithm, yielding higher performance and convergence speed (i.e., Twin-Delayed DDPG algorithm from Dankwa and Zheng (2019)).

Finally, transfer learning techniques could be applied to take advantage of the optimal behavior of some agents and replicate their actor networks to the rest of the agents. This modification is feasible since the agents on our task are homogeneous. It would be interesting to evaluate the accumulated training reward when we utilize pre-trained networks in low complexity tasks (i.e., environments with only three agents) to test the agents’ behavior in more complex ones.
Appendix A

Technicalities

A.1 Github repository

All the code necessary to run this project can be found in the following Github repositories:

Multi-Agent Cooperation

https://github.com/LouGalata/Multiagent-Cooperation.git

This repository is implemented in TENSORFLOW 2.4 and contains the following experiments:

1. train-iql.py: IQL with VDN mixing strategy
2. train-gcn.py: IQL enhanced with Graph Convolutional Network with VDN
3. train-gat.py: IQL enhanced with Graph Attentional Network with VDN
4. train-gat-ind.py: IQL enhanced with Graph Attentional Network with VDN without shared weights
5. train-rnn-ind.py: IQL with recurrence and VDN mixing strategy
6. train-dueling-dqn.py: Independent Dueling DQN with VDN mixing strategy
7. train-maddpg.py: MADDPG algorithm
8. train-centr-maddpg: MADDPG with one centralized critic

The configuring parameters are located on the top of each execution file. The different experience replay buffers are located on the following files:

1. replay_buffer.py: The transitions saved are: state, action, adjacency matrix, next action, reward and done. Here we use random sampling.
2. replay_buffer_iql.py: The original experience replay without the storage of the adjacency matrix
3. prioritized_replay_buffer.py: We use prioritized experience replay

Furthermore, in the commons package, we have utility functions used by all the experiments. In the models package, there is the self-attention implementation for the recurrency layer. Additionally, in the agents package, there are the critic and action networks for the MADDPG algorithm. Finally, in execution-testing.py we tested our trained models.

It should be mentioned that part of the source code was retrieved from this repository. The library Spektral was used for the graph neural network implementation.
Appendix A. Technicalities

MAGAT model

https://github.com/LouGalata/MAGAT.git

This repository is implemented in PYTORCH and contains the following experiments:

1. main.py: Execution of the MAGAT or MADDPG algorithm. All the hyperparameters are defined here.
2. maddpg.py: Training procedure of the MADDPG or the MAGAT variation
3. networkforall.py: Definition of the critic and actor networks for the MADDPG
4. networkforgat.py: Definition of the critic and actor networks for the MAGAT variation
5. layers.py: Self-Attention model for the Graph Architecture

It should also be mentioned that part of the source code was retrieved from this repository.
Bibliography


Finch, Steven (2004). *Ornstein-Uhlenbeck Process*


Bibliography


