TÉCNICO LISBOA

Ad hoc teamwork using approximate representations

Filipe Miguel Gomes de Sousa

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Francisco António Chaves Saraiva de Melo
Prof. José Alberto Rodrigues Pereira Sardinha

November 2022

Acknowledgments

I would like to give my thanks to my supervisors, Francisco Melo and Alberto Sardinha, for proposing this interesting and innovative dissertation topic, helping me find the best sources of information and suggesting the best courses of action at each phase of the project. I would also like to thank them for supporting me during my learning curve working with the complex Half-Field Offense domain. I would also like to thank João Ribeiro for the time and attention he dedicated helping me better understand the algorithms I used and the model training and evaluation processes, and for providing me with his agents framework.

I would also like to acknowledge Matthew Hausknecht, and all those who contributed to the development of the Half-Field Offense environment – although not the easiest to learn, it is a very powerful tool! Additionally, I would like to thank Pedro Santos, Francisco Delgado, João Pirralha and Inês Loução Vieira for sharing their experience and tips on how to work with this environment, and for providing me with their code and other useful resources.

On a more personal side, I would like to thank all my friends and family members who contributed to shaping who I am today. I would like to give a special thank you to my brother, for always setting high standards and working hard to achieve them, and to my parents, for all the practical and emotional support they have always lent me.

Thank you, everyone.

Abstract

The mass production of technological systems around the world is both an economic and ecological issue we face today. It is critical that we find alternate solutions as soon as possible, to work towards a more sustainable society. An emerging field that can bring some advancements towards this goal is that of *ad hoc* teamwork, which studies how an agent can be integrated in a new team without prior knowledge of its new teammates. Such agents would be reusable in future tasks, reducing the need to create such a huge amount of agents. Recent advances in this field shown that it is possible to design agents capable of achieving high performance in this task. However, none of the existing approaches tackled this problem for large domains with partial observability.

In this paper, we present a new algorithm, Partially Observable Plastic Policy (POPP), that combines transfer learning with Deep Recurrent Q-Networks, by having an agent learn policies to play along with different types of teammates, and reusing that knowledge when faced with new teams. We chose the Half-Field Offense domain for evaluation. We experiment with different configurations, with and without partial observability, and with known and unknown teammates. Finally, we present and discuss our results, and compare them to non-recurrent approaches, namely Deep Q-Networks (DQN). We concluded that POPP was able to quickly identify most of the previously known teams, and surpassed the score rate of a DQN approach in partially observable scenarios.

Keywords

Ad Hoc Teamwork; Multi-agent Systems; Transfer Learning; Function Approximation; Recurrent Neural Networks

Resumo

A produção em massa de sistemas tecnológicos por todo o mundo é uma questão não só económica, mas também ecológica da nossa atualidade. É fundamental que encontremos soluções alternativas o mais rápido possível, para trabalhar em direção a uma sociedade mais sustentável. Uma área emergente que pode trazer alguns avanços nesse sentido é a do trabalho em equipa *ad hoc*, que estuda a integração de um agente numa nova equipa sem conhecer previamente os seus novos colegas de equipa. Este agente seria reutilizável em tarefas futuras, reduzindo assim a necessidade de produzir uma quantidade tão volumosa de agentes. Com os recentes avanços nesta área, é possível criar agentes capazes de alcançar um elevado desempenho nesta tarefa. No entanto, nenhuma das abordagens existentes tratou este problema em domínios de elevada dimensão com observabilidade parcial.

Neste artigo, apresentamos um novo algoritmo, Partially Observable Plastic Policy (POPP), que combina aprendizagem por transferência com Deep Recurrent Q-Networks, em que um agente aprende políticas para cooperar com diferentes tipos de colegas de equipa, e reutiliza esse conhecimento quando confrontado com novas equipas. Escolhemos o domínio Half-Field Offense para avaliação. Experimentámos diferentes configurações, com e sem observabilidade parcial, e com colegas de equipa conhecidos e desconhecidos. Finalmente, apresentamos e discutimos os nossos resultados e comparamo-los com abordagens não recorrentes, como Deep Q-Networks (DQN). Concluímos que o POPP foi capaz de identificar rapidamente a maioria das equipas conhecidas anteriormente e superou a percentagem de golos marcados de uma abordagem DQN em cenários parcialmente observáveis.

Palavras Chave

Trabalho em Equipa *Ad Hoc*; Sistemas Multi-agente; Aprendizagem por Transferência; Aproximação de Funções; Redes Neuronais Recorrentes

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AATEAM	Achieving the Ad-hoc Teamwork by Employing the Attention Mechanism
ATPO	Ad hoc Teamwork under Partial Observability
BOPA	Bayesian Online Prediction for Ad hoc teamwork
DQN	Deep Q-Network
DQN-PP	Deep Q-Network - PLASTIC-Policy
DRQN	Deep Recurrent Q-Network
FQI	Fitted Q-Iteration
HFO	Half-Field Offense
LSTM	Long Short-Term Memory
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Problem
MMDP	Multi-agent Markov Decision Problem
MPOMDP	Multi-agent Partially Observable Markov Decision Problem
MSE	Mean Squared Error
PLASTIC	Planning and Learning to Adapt Swiftly to Teammates to Improve Cooperation
POMDP	Partially Observable Markov Decision Problem
POPP	Partially Observable PLASTIC-Policy
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
UCT	Upper Confidence bound applied to Trees
VI	Value Iteration

1

Introduction

Contents

With the non-stopping progress in the world of computer science during the last few decades, a wide variety of systems were designed and deployed. Many of those systems were custom-tailored to perform a specific task in a specific environment. This means they cannot be redeployed to perform a different task in a different environment without a large cost and effort. Therefore, it would be useful to build a system that was able to dynamically adapt to different tasks, should its physical capabilities allow it to do so. However, the design of an agent with such a generalization capability is not trivial. This challenge can be somewhat simplified, though, if the task is cooperative, since a new agent has its new teammates' behaviour as an additional source of information.

Ad hoc teamwork is a field of study aiming to determine how an agent can be integrated in a group of unknown teammates "on the fly", i.e., without any prior coordination, and with poor or nonexistent communication protocols. In a teamwork task, the agent can observe its teammates' behaviour to determine and adapt to the task they are solving, without the need for humans to specify it manually.

However, a multi-agent scenario brings an additional layer of complexity to the task, since, as there is no pre-coordination between the agent and the new team, the teammates' behaviour is itself another source of uncertainty. A possible simplification of this problem occurs if the agent has worked previously in the same task, but with a different team, since it can try to extrapolate its prior knowledge when working with the new team.

Another hindrance which increases the complexity of the *ad hoc* teamwork problem is when the environment is only partially observable. This is the most realistic scenario, though, since most physical agents' sensors are imperfect (i.e., there can be noise in the data collection), and don't allow for a full representation of the world (e.g.: a 2D camera provides an incomplete representation of a 3D world).

With partial observability comes another problem: if the agent aims to perform optimally (that is, to determine the optimal policy) in a partial observable environment, it must choose how to act considering everything it saw since the beginning of the task, since the most recent observation is not enough to describe the world state. This phenomena is also called history-dependence of the optimal policy.

A final hazard that adds complexity to the *ad hoc* teamwork scenario is the sparsity of reward signals given to the agent. In many real-world tasks, only the final result matters (e.g., an agent exploring a maze), and therefore it is difficult for the agent to know whether it is performing well until the end of the task.

State of the art algorithms in the field of *ad hoc* teamwork have been developed to address some of the aforementioned issues, however, no algorithm has been created that can solve all of these problems simultaneously.

1.1 Research Question

With our work, we aim to address an *ad hoc* teamwork setting which is missing in previous literature, combining:

(i): complex domains, with a continuous, high dimensional observation space;

(ii): partial observability, and subsequent history-dependence of the optimal policy;

(iii): sparse reward signals from the environment.

With this in mind, our research question becomes:

Is it possible to develop an autonomous agent which performs near-optimally in the *ad hoc* teamwork problem, in a complex, partially observable environment with sparse rewards?

1.2 Contributions

We contribute to the *ad hoc* teamwork research field with the development of a new algorithm, Partially Observable PLASTIC-Policy (POPP), which uses the agent's experience working with past teams to adapt to previously unseen teammates in a continuous, high dimensional and partially observable environment with sparse rewards. In particular, the partial observability - and inherent history-dependence of the optimal policy - is dealt with using Recurrent Neural Networks (RNNs).

1.3 Document Outline

This thesis is is organized as follows:

Chapter 2: Overview of the theoretical background related to Reinforcement Learning (RL), and how deep learning techniques can help solving reinforcement learning problems (Deep Q-Learning).

Chapter 3: Discussion and analysis of related work regarding multi-agent RL and *ad hoc* teamwork.

Chapter 4: Description of POPP, our novel algorithm, and how it extends a state-of-the art algorithm to complex, partially observable domains using Deep Recurrent Q-Networks (DRQNs).

Chapter 5: Experimental evaluation of POPP in the Half-Field Offense (HFO) domain, and comparison of its results with non-recurrent architectures, in totally and partially observable settings.

Chapter 6: Discussion of our results, how they contributed to the field of *ad hoc* teamwork, and what room there is for further improvements or additional research.

2

Background

Contents

In this section, we will define some concepts and notation, and explain the lexicon we will use in the next sections, which will give us a better understanding of multi-agent systems, *ad hoc* teamwork and function approximation.

2.1 Probability Theory

Here, we define preliminary concepts that will be useful to formalize the concept of uncertainty that is so central in *RL* problems. We assume the reader is familiar with the concepts of *random variables*, *probability*, *probability distribution* and *conditional probability*.

2.1.1 Expected value

For a discrete, real-valued, random variable X with domain D_x, the *expected value* of X is defined as

$$\mathbb{E}[X] = \sum_{x \in D_x} xP(X = x)$$

2.1.2 Conditional expected value

For two discrete, real-valued, random variables X (with domain D_x) and Y, the *conditional expected value* of X given that Y is equal to y is defined as

$$\mathbb{E}[X \mid Y = y] = \sum_{x \in D} xP(X = x \mid Y = y)$$

Sometimes we want to express the conditional expected value of a random variable X, but the condition that changes its probability distribution not that a certain random variable Y takes a certain value y. In general, for a random Boolean variable B, we can define the *conditional expected value* of X given that B is true as

$$\mathbb{E}[X \mid B] = \sum_{x \in D} xP(X = x \mid B)$$

For instance, if we want to represent the expected value of X given that Y is greater than y, we can write

$$\mathbb{E}[X \mid Y > y] = \sum_{x \in D} xP(X = x \mid Y > y)$$

2.2 Reinforcement Learning

2.2.1 Decision-theoretic frameworks

Multiple frameworks exist to represent dynamical systems where an agent must sequentially decide on an action to take on each time step, and the result of such actions is bound by some form of uncertainty. The frameworks we we will focus our attention on are Markov Decision Problems (MDPs) and its variants, depending on whether there is one or multiple agents (in which case the process is prefixed with "Multi-agent") and on whether the agent can totally observe the environment states, or only partially (in which case the process is prefixed with "Partially Observable").

Thus, apart from MDPs, there are Multi-agent Markov Decision Problems (MMDPs), Partially Observable Markov Decision Problems (POMDPs) and Multi-agent Partially Observable Markov Decision Problems (MPOMDPs). Each framework's characteristics are described in Table 2.1.

	Total observability	Partial observability
1 agent	MDP	POMDP
>1 agent	MMDP	MPOMDP

Table 2.1: Frameworks for sequential decision processes in the face of uncertainty.

We will present the definition for POMDPs and MPOMDPs only, as they are the most relevant ones for this work, and the other two can be seen as specializations of these ones.

2.2.1.A POMDP

A POMDP is a framework to formalize a sequential decision process with a single autonomous agent under partial observability and uncertainty (i.e., given that an action is executed, there's more than one possible next state).

It can be described as a tuple $(X, A, P, Z, O, r, \mu_0)$, where:

- X is the set of environment states
- A is the set of actions
- $P : X \times A \times X \to \mathbb{R}$ is the transition probability, with $P(x, a, x')$ being the probability of the environment evolving to state x' when the agent executes action a on state x; $P(x, a, x')$ is more commonly written as $P(x' \mid x, a)$ to highlight its conditional nature
- Z is the set of possible observations
- $O : X \times A \times Z \to \mathbb{R}$ is the observation probability, with $O(x', a, z)$ being the probability of the agent seeing the observation z when the execution of action a resulted in a transition to state x'; $O(x', a, z)$ is more commonly written as $O(z \mid x', a)$ to highlight its conditional nature

- $r : X \times A \to \mathbb{R}$ is the immediate reward function, where $r(x, a)$ is the reward for taking action a on state x
- $\mu_0 : X \to [0, 1]$ is the probability distribution for the initial environment state x_0 (sometimes referred to as the initial belief)

The execution of a POMDP is carried out as follows: at each time step t (with the environment state being x_t, unbeknownst to the agent), the agent sees an observation z_t; then, the agent executes an action a_t, observes a reward r_t and the environment evolves to state x_{t+1}.

2.2.1.B MPOMDP

Similarly, an MPOMDP is also a framework to formalize a sequential decision process under partial observability and uncertainty, but targeted to a multi-agent setting. It can be described as a tuple $(I, X, A, P, Z, O, r, \mu_0)$, where:

- I is the index set of agents
- X is the set of environment states
- A^i is the set of actions available for agent i, and $A = \bigtimes_{i \in I} A^i$ is the set of joint actions
- $P : X \times A \times X \to \mathbb{R}$ is the transition probability, with $P(x, a, x')$ being the probability of the environment evolving to state x' when the team executes the joint action a on state x; $P(x, a, x')$ is more commonly written as $P(x' \mid x, a)$ to highlight its conditional nature
- Z^i is the set of possible observations for agent i and $Z = \bigtimes_{i \in I} Z^i$ is the set of joint observations
- $O : X \times A \times Z \to \mathbb{R}$ is the observation probability, with $O(x', a, z)$ being the probability of the team seeing the joint observation z when the execution of the joint action a resulted in a transition to state x'; $O(x', a, z)$ is more commonly written as $O(z \mid x', a)$ to highlight its conditional nature
- $r : X \times A \to \mathbb{R}$ is the immediate reward function, where $r(x, a)$ is the reward for taking the joint action a on state x
- $\mu_0 : X \to [0, 1]$ is the probability distribution for the initial environment state x_0 (sometimes referred to as the initial belief)

Note that the fact that the action space (A) and the observation space (Z) are indexed by agent reflects many real world scenarios, where different agents can have different actuators and sensors.

The execution of an MPOMDP is carried out as follows: at each time step t (with the environment state being x_t, unbeknownst to the team), each agent i sees an observation z_t^i; then each agent i selects

an action a_t^i, resulting in the joint action a_t; upon execution of a_t, the whole team observes a reward r_t and the environment evolves to state x_{t+1}.

In the cases where there is total observability (MDPs and MMDPs), the observation space (Z) and probabilities (O) are not necessary, nor is the probability distribution of the initial state (μ_0), as at each time step t, the agents know the current environment state x_t.

2.2.2 History

An important concept in the context of sequential decision frameworks is that of *history*. It represents the sequence of state-action pairs since the start of the decision process, up to the current time step t. So, the history at time step t is represented as $h_t = \{x_0, a_0, ..., x_{t-1}, a_{t-1}, x_t\}$. In a partially observable scenario, since the agents are unable to access the environment state, the history uses the observations instead ($h_t = \{z_0, a_0, ..., z_{t-1}, a_{t-1}, z_t\}$).

2.2.3 Belief

For partially observable environments, as the environment state is not accessible by the agent, it is useful to define a *belief*, which is a probability distribution over the environment states, which is updated every time step. It is represented as $\mu_t : X \to [0, 1]$ and can be computed using the expression

$$\mu_t(x) = P(x_t = x \mid h_t, x_0 \sim \mu_0)$$

, i.e., it is the probability of being in state x at time step t given the observation/action history h_t, and the probability distribution μ_0 of the initial state.

2.2.4 Policy

Another core concept of sequential decision frameworks is that of *policy*, which represents the criteria the agents use to decide on which action to choose. It is defined as $\pi : H \to \Delta(A)$, i.e., it maps a history to a probability distribution over joint actions, and $\pi(a_t = a \mid h_t)$ represents the probability of the agents choosing joint action $a \in A$ at time step t given the history up to that point, h_t.

If, at every time step t, the history h_t unequivocally determines the chosen action a_t, the policy π is said to be *deterministic*; otherwise, it is said to be *stochastic*. Additionally, if the current state x_t or the current observation z_t are enough to describe π, it is said to be *stationary*, and it can be simply written as $\pi(a \mid x)$ for totally observable scenarios, or $\pi(a \mid z)$ for partially observable ones; if we also allow π to access the size of the history up to that point ($|h_t|$) it is said to be a *Markov* policy; if further information about the history is required to compute π, it is said to be *history-dependent*.

In partially observable environments, it is common for policies to be history-dependent, and to receive a belief as input, instead of an observation, i.e., $\pi(a \mid \mu_t)$, since μ_t can summarize the agent's learning about its current state since the beginning of the interaction.

We also define the expected reward of a state x given that the agent follows a policy π as

$$r_\pi(x) = \mathbb{E}[r(x, a) \mid a \sim \pi]$$

and the probability of transitioning from state x to x' given that the agent follows a policy π as

$$P_\pi(x' \mid x) = P(x' \mid x, a \sim \pi)$$

2.2.5 Gain

In an RL scenario, we would ideally like to learn how to act in a way that yields the maximum possible reward. Specifically, we would like to maximize the total reward (or gain) G_T over the course of the whole interaction:

$$G_T = \sum_{t=1}^{T} \gamma^t r_t$$

where $r_t = r(x_{t-1}, a_{t-1})$ and T is the duration of the interaction.

2.2.6 Optimal policy

The agents should, therefore, learn a policy π^*, known as the *optimal policy* for the decision problem, that maximizes the expected gain. We can write

$$\mathbb{E}^*_\pi[G_T] = \sup_{\pi \in \Pi} \mathbb{E}_\pi[G_T]$$

where $\mathbb{E}_\pi[G_T]$ is the expected value of G_T if the agent follows policy π, and Π is the set of all possible policies for the decision problem.

2.2.7 Value functions

In order to simplify the notation for the upcoming sections, it is useful for us to define two functions, the *state-value function* $V_\pi(x)$ and the *action-value function* $Q_\pi(x, a)$.

2.2.7.A State-value function

The state-value function $V_\pi(x)$ represents the expected gain of the agent during its whole interaction with the environment given that its initial state is x, and that it follows policy π. It can be defined as

$$V_\pi(x) = E_\pi[G_T \mid x_0 = x]$$

2.2.7.B Action-value function

The action-value function (or *Q-function*) $Q_\pi(x, a)$ represents the expected gain of the agent during its whole interaction with the environment given that its initial state is x, its first action is a and that it follows policy π for the remainder of the interaction. It can be defined as

$$Q_\pi(x, a) = E_\pi[G_T \mid x_0 = x, a_0 = a]$$

2.2.7.C Computing value functions

To compute $V_\pi(x)$, a recursive solution exists:

$$V_\pi(x) = r_\pi(x) + \gamma \sum_{x' \in X} P_\pi(x' \mid x) V_\pi(x')$$

To compute $Q_\pi(x, a)$ first note that it can be obtained directly from $V_\pi(x)$:

$$Q_\pi(x, a) = r(x, a) + \gamma \sum_{x' \in X} P(x' \mid x, a) V_\pi(x') \tag{2.1}$$

Similarly, we can obtain $V_\pi(x)$ from $Q_\pi(x, a)$

$$V_\pi(x) = \sum_{a \in A} \pi(x, a) Q_\pi(x, a) \tag{2.2}$$

Putting together 2.1 and 2.2 we obtain a recursive solution for $Q_\pi(x, a)$:

$$Q_\pi(x, a) = r(x, a) + \gamma \sum_{x' \in X} P(x' \mid x, a) \sum_{a' \in A} \pi(x', a') Q_\pi(x', a') \tag{2.3}$$

2.2.8 MDP solution methods

In this section we will see how we can solve MDPs. We will discuss solutions for POMDPs later, in Section 2.3.5.

Since the goal of RL is to train an agent so it maximizes the expected gain G_T, we would like the agent to follow the optimal policy π^*. There are three main categories of methods for computing it:

Model-based methods , where the agent either builds an increasingly accurate model of the decision problem, estimating r and P, or has direct access to the model, and uses it to compute V or Q, and finally π^*;

Value-based methods , where the agent builds increasingly accurate estimates for V or Q and uses them to compute π^*, skipping the need to estimate r or P;

Policy-based methods , where the agent builds an increasingly accurate estimate for π^* skipping the need to estimate r, P, V and Q.

Value-based and policy-based methods are also included in a category commonly known as model free methods, since they neither require the agent to have access to the environment model, nor to estimate it. Now, we cover some of those methods, which will be used in works we will describe from chapter 3 onward. We will cover the variants of these algorithms designed for MDPs only since they have simpler notation.

2.2.8.A Value Iteration

Value Iteration (VI) is a model-based method for solving MDPs which requires the transition probabilities P and the reward function r to be known by the agent *a priori*. The model is used to compute increasingly accurate estimates of either Q or V. Since most algorithms we will address in this work focus on the action-value (Q) function, that is the variant of VI we will see here. The goal of this algorithm is, then, to compute the optimal Q-function, given by

$$Q^*(x,a) = Q_{\pi^*}(x,a) = \sup_{\pi \in \Pi} Q(x,a)$$

In order to do so, we use the *Bellman equation of optimality* for the action-value function, which states that

$$Q^*(x,a) = r(x,a) + \gamma \sum_{x' \in X} P(x' \mid x,a) \max_{a' \in A} Q^*(x',a') \tag{2.4}$$

With this, we can get increasingly accurate estimates for $Q^*(x,a)$ by setting every entry of $Q_0(x,a)$ to an arbitrary value (commonly zero) and iteratively applying the following update rule:

$$Q_{k+1}(x,a) = r(x,a) + \gamma \sum_{x' \in X} P(x' \mid x,a) \max_{a' \in A} Q_k(x',a') \tag{2.5}$$

The algorithm stops once the variation in every entry in the Q-function between two consecutive iterations is lesser than a given threshold ϵ, i.e., when the following condition is true:

$$\max_{x \in X, a \in A} |Q_{k+1}(x,a) - Q_k(x,a)| < \epsilon \tag{2.6}$$

The optimal policy can be then computed by

$$\pi^*(a \mid x) = \begin{cases} \frac{1}{n_a}, & if \;\; Q_{k+1}(x,a) = \max_{a' \in A} Q_{k+1}(x,a') \\ 0, & otherwise \end{cases} \tag{2.7}$$

where $n_a = |\operatorname{argmax}_{a \in A} Q_{k+1}(x,a)|$.

Note that the agent does not need to interact with the environment in order to run VI, since it has direct access to P and r. It solely computed a policy to be used in a future interaction. For this reason, VI is also classified as a *planning* algorithm.

2.2.8.B Q-Learning

Unlike VI, *Q-Learning* [8] is a value-based method, thus not requiring the agent to model the MDP. Instead, the agent relies on the data it gets from interacting with the environment to approximate Q^*.

The agent starts at time step 0 in state x_0, and initializes Q_0 with arbitrary values (commonly zero). Then, at every time step t, the agent executes an action a_t, observes a reward r_t and reaches a next state x_{t+1}. With that information, the agent then computes

$$Q_{t+1}(x_t, a_t) = Q_t(x_t, a_t) + \alpha_t [r_t + \gamma \max_{a' \in A} Q_t(x_{t+1}, a') - Q_t(x_t, a_t)]$$

and sets all the remaining entries (x,a) of Q_{t+1} to $Q_t(x,a)$. Computationally, this means we can store a single matrix Q, and update a single entry (x_t, a_t) of Q per iteration.

α_t is a positive value known as the *learning rate*, and has a great impact in the rate of convergence of Q-Learning. There are multiple approaches to choose a_t.

The algorithm stops when Equation (2.6) holds (treating the iteration k as the time step t), and Watkins [8] shown that for properly chosen values of α_t, Q-Learning is guaranteed to converge (i.e., for every positive value of ϵ there exists a time step T where Equation (2.6) holds).

2.2.9 Exploration vs. Exploitation

In RL we are often faced with the problem of choosing whether to choose the action that, according to our current knowledge, will yield a best long-term gain (i.e., to *exploit* that action), or to choose one that allows us to improve our knowledge about the environment (i.e. to *explore* it). This problem is often

known as the *exploration vs. exploitation* trade-off.

A common approach that tries to balance this trade-off is the ϵ-*greedy* policy, which selects an exploratory action with probability ϵ and exploits the best action with probability $1 - \epsilon$. For instance, for the Q-Learning algorithm, the ϵ-greedy policy at time step t would be

$$\pi_t^\epsilon(a \mid x) = \begin{cases} \frac{1-\epsilon}{n_a}, & if \;\; Q_t(x,a) = \max_{a' \in A} Q_t(x,a') \\ \frac{\epsilon}{|A|-n_a}, & otherwise \end{cases} \tag{2.8}$$

where $n_a = |\operatorname{argmax}_{a \in A} Q_t(x,a)|$.

2.2.10 Function Approximation

Suppose that a given MDP has a finite set of discrete actions A and a finite set of discrete states X. Our Q-function in the algorithms we have seen will be a $|X| \times |A|$ matrix. If $|X|$, $|A|$, or both are too large, the computations may become intractable. Even worse, if the states in X are, instead, continuous, we cannot write $Q(x,a)$ in matrix form, and thus we cannot use these methods.

A possible approach to solve this problem is to use *function approximation*, a field that studies how we can find functions from a simpler class that best approximate a more complex (target) function. An important result in this field is that equations of the form

$$E(F(x) - \hat{\theta}) = 0 \tag{2.9}$$

can be addressed using *stochastic approximation algorithms*, using the update rule

$$\hat{\theta}_{k+1} = \hat{\theta}_k + \alpha(F(x) - \hat{\theta}) \tag{2.10}$$

For instance, suppose we want to estimate the optimal Q-function for a given policy π, $Q^*(x,a)$. We know from Equation (2.4) that

$$Q^*(x,a) = r(x,a) + \gamma \sum_{x' \in X} P(x' \mid x,a) \max_{a' \in A} Q^*(x',a')$$

But, by the definition of conditional expected value (Section 2.1.2), we have

$$\sum_{x' \in X} P(x_{t+1} = x' \mid x_t = x, a_t = a) \max_{a' \in A} Q^*(x',a') = \mathbb{E}_{\pi^*}[\max_{a' \in A} Q^*(x_{t+1},a') \mid x_t = x, a_t = a]$$

Therefore

$$\begin{aligned} Q^*(x,a) &= r(x,a) + \gamma \mathbb{E}_{\pi^*}[\max_{a' \in A} Q^*(x_{t+1}, a') \mid x_t = x, a_t = a] \\ &= \mathbb{E}_{\pi^*}[r(x_t, a_t) + \gamma \max_{a' \in A} Q^*(x_{t+1}, a') \mid x_t = x, a_t = a] \end{aligned}$$

which is equivalent to

$$\mathbb{E}_{\pi^*}[r(x_t, a_t) + \gamma \max_{a' \in A} Q^*(x_{t+1}, a') - Q^*(x_t, a_t) \mid x_t = x, a_t = a] = 0 \tag{2.11}$$

Since Equation (2.11) is in the form 2.9, we can approximate it using the update rule 2.10.

2.2.11 Fitted Q-iteration

We can, therefore, use the method outlined in Section 2.2.10 to estimate the optimal Q-function in complex domains. *Fitted Q-Iteration* is a *batch* RL method, since it stores the experience it collects while interacting with the environment in a buffer, and uses them later update its estimation of our usual functions. In contrast, Q-Learning is an *online* method, since it updates its estimations right after collecting the data, skipping the need to store it in a buffer.

However, one of the main problems of Q-Learning is that its updates are not independent of one another, since they come from consecutive time steps. In the more recent versions of Fitted Q-Iteration (which are the ones we will describe here), the updates are done in *mini-batches* of size N, by uniformly selecting N time steps from the replay buffer, and using the to update its estimation of the Q-function, which ensures independent updates, resulting in better convergence properties.

Fitted Q-Iteration also uses what is called a *target estimator*, apart from its current estimation of the Q-function. It does so because we do not have access to Q^* to compute our target ($\pi^r(x_t, a_t) + \gamma \max_{a' \in A} Q^*(x_{t+1}, a')$ in Equation (2.11)), since Q^* is what we are trying to estimate. If we always used our current estimate of Q^* to update our next estimation, we would be *chasing a moving target*, as we are trying to get closer and closer to an ever-changing target Q^*. The target estimator is then used to store an estimate for Q^* that is updated only every $P^{\odot}$ time steps, to minimize this problem.

Fitted Q-Iteration is described in Algorithm 2.1, with B being the mini-batch size, M the size of the weights' vector, P the update period of the estimator, and $P^{\odot}$ the update period for the target estimator. Note that, in the algorithm, the update

$$w_{k,t+1} = \underset{w \in \mathbb{R}^M}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} ||y_{k,t,n} - \hat{Q}_w(x_n, a_n)||^2$$

consists of minimizing the Mean Squared Error (MSE), which is a standard error measure for estimators,

given by

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(X_n - \hat{X_n})^2$$

where X is a data sample of size N and $\hat{X}$ is a vector with the predicted values.

Algorithm 2.1: Fitted Q-Iteration

begin
  $D \leftarrow \{\}$ `// The replay buffer`
  $w_{0,0} \leftarrow 0$ `// The weights` $w_{k,t}$ `for the estimator at time step` t `of episode` k
  $w^{\odot} \leftarrow w_{0,0}$ `// The weights for the target estimator`
  $k \leftarrow 0$ `// The current episode`
  repeat
    $t \leftarrow 0$ `// The current time step`
    do
      Interact with environment and get transition $T_{k,t} = (x_{k,t}, a_{k,t}, r_{k,t}, x_{k,t+1})$
      $D \leftarrow D \cup \{T_{k,t}\}$ **if** $t \mod P = 0$ **then**
        $N \leftarrow \max(B, |D|)$
        Randomly sample mini-batch $\{(x_n, a_n, r_n, x'_n), n = 1, ..., N\}$
        $w_{k,t+1} = \text{argmin}_{w \in \mathbb{R}^M} \frac{1}{N}\sum_{n=1}^{N} ||y_{k,t,n} - \hat{Q}_w(x_n, a_n)||^2$
        where
        $y_{k,t,n} = r_n + \gamma \max_{a \in A} \hat{Q}_{w^{\odot}(x'_n, a)}$ `// The target estimator`
      $t \leftarrow t + 1$
      if $t \mod P^{\odot} = 0$ **then**
        $w^{\odot} \leftarrow w_{k,t}$
    while $x_{k,t}$ is not terminal
    $w_{k+1,0} \leftarrow w_{k,t}$ `// The weights continue to the next episode`
    $k \leftarrow k + 1$
  until forever

2.3 Deep Learning

2.3.1 Artificial Neuron

An *artificial neuron* is a simple computational unit which has its origins in neurobiology, as its design intends to resemble the functioning of a biological neuron. It takes as input a real-valued vector ($\mathbf{x}$), and outputs a single real value (o). The computation done by the neuron consists in computing a weighted sum of the input values, using a configurable weights vector ($\mathbf{w}$) and applying an activation function (f) to the result, as illustrated by Fig. 2.1 (a). The weighted sum (net, as in "net value"), can be calculated as

$$net = \mathbf{w} \cdot \mathbf{x} = \sum_{k=1}^{n} w_k x_k$$

where n is the size of the input vector x. Regarding the activation function, many exist, but the simplest one is the *step function*

$$f(\text{net}) = \mathbb{I}[\text{net} \geq \theta] = \begin{cases} 1 & \text{if net} \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

where θ is the *threshold*. Alternatively, and also very often done both in the literature and in practice, we can add a bias term to the net value ($\text{net} = \mathbf{w} \cdot \mathbf{x} + b$) and set the activation function to $f(\text{net}) = \mathbb{I}[\text{net} \geq 0]$. Other commonly used activation functions include the *sigmoid* function, the *tanh* function, Rectified Linear Unit (ReLU) and the *softmax* function (the last one is used more often for the output layer).

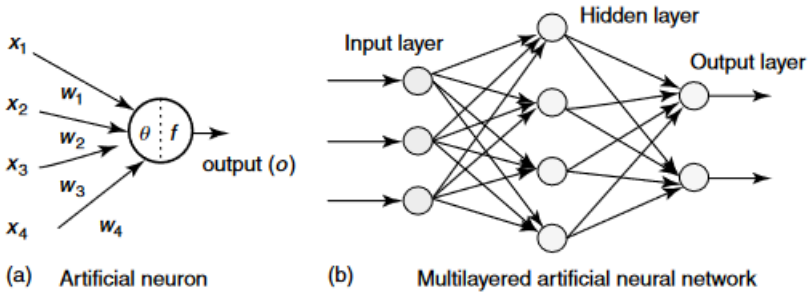


Figure 2.1: Simple representations of an artificial neuron (a) and an artificial neural network with one hidden layer (b). Source: Adapted from [1].

2.3.2 Artificial Neural Network

An *artificial neural network* is a network composed by several artificial neurons, where the outputs of some of them are connected to the inputs of others. One of the most basic architectures is the *multi-layered neural network*. It has an *input layer*, one or more *hidden layers*, and an *output layer* (Fig. 2.1 (b)).

To compute the output for a given input, each layer of neurons executes its own local computation process as described in the previous section, and outputs a value which can be used by the neurons on the next layer. This is known as *forward propagation*, and it can be parallelized, as neurons on the same layer do not depend on each other's computation to perform their own.

The weights of each neuron in the network can be trained, or manually adjusted, so that each input produces the desired output. One of the most standard algorithms used for the training process is *backpropagation*. In short, it consists in computing the gradient of the error/loss function in respect to the weights and subtracting from them that derivative multiplied by a constant (η, known as the learning rate) from the weights, resulting in the updated weights. One of the most commonly used loss functions

is the MSE

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{k=1}^{n}(y_k - o_k)^2$$

for a multi-layered network with n output neurons, with o_k being the output of the k^{th} neuron and y_k being the desired output for that same neuron. The derivative of this error function is then used to update the weights:

$$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} - \eta\nabla_{\mathbf{w}}L(\mathbf{w}, b) \tag{2.12}$$

$$b^{\text{new}} = b^{\text{old}} - \eta\nabla_{b}L(\mathbf{w}, b) \tag{2.13}$$

Note that this update rule is for the weights associated with a single neuron. To represent the weights of a full layer of the network (with m neurons), we can use a matrix (and a vector for the bias):

$$\mathbf{W} = [\mathbf{w}_1\ \mathbf{w}_2\ \ldots\ \mathbf{w}_m]^T$$

$$\mathbf{b} = [b_1\ b_2\ \ldots\ b_m]^T$$

Since the derivative of the error function depends on the outputs, the first layers depend on the derivatives computed in the last layers (hence the name "backpropagation"). However, as in forward propagation, neurons in the same layer can compute their own derivatives in parallel, since there are no dependencies between neurons in the same layer.

Backpropagation when used in the circumstances we described is also known as Stochastic Gradient Descent (SGD). However, more efficent methods than SGD were invented since it first appeared, namely one known as *Adam*, which is one of the most popular backpropagation algorithms (also known as *optimizers*) nowadays. Its implementation is out of the scope of this work.

2.3.3 Recurrent Neural Network

Another popular artificial neural network architecture is the RNN. This family of neural networks is designed to extract information from sequential data [2]. A simple representation of an RNN is shown in Fig. 2.2. The input data comes in the form $\mathbf{x}^{(1)}\mathbf{x}^{(2)}\ldots\mathbf{x}^{(t)}$, where each $\mathbf{x}^{(t)}$ is the input vector at time step t. Here, the index t could have a meaning other than time. For instance, it could represent the index of a letter in a word or the index of a word in a sentence, but, for our purposes, we will assume the inputs are indexed by their time step.

The main novelty about RNNs is that they feature a vector $\mathbf{h}^{(t)}$, called the *hidden state*, which stores information between time steps. This state includes information from all previous time steps, but the most recent ones contribute more to its value than the older ones, which is analogous to the idea of

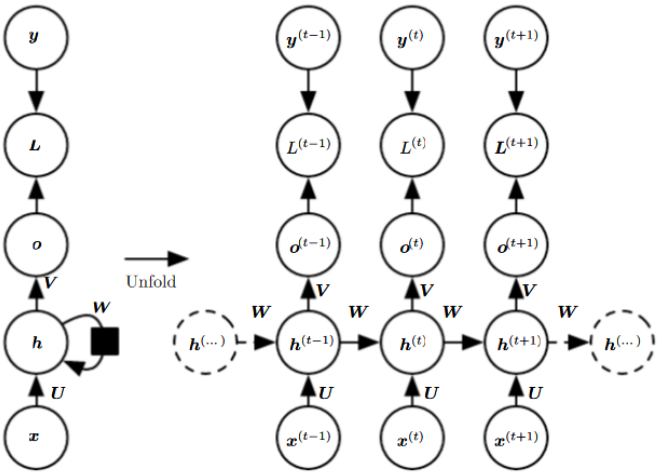


Figure 2.2: Compact (left) and unfolded (right) circuit diagrams of a simple RNN. Source: Adapted from [2].

short-term memory. Unlike regular neural networks, RNNs can get time-dependent information, like the direction an object is moving and its speed (by analyzing the changes to its position in the last few time steps).

Another difference between RNNs and regular neural networks is that they use a greater amount of weight matrices: $\mathbf{U}$, between the input layer and the hidden state, $\mathbf{V}$, between the hidden state and the output layer (or a hidden layer, in the case of multi-layered RNNs) and $\mathbf{W}$, between hidden states in two consecutive time steps.

The forward propagation algorithm has the following steps:

- The input at the current time step, $\mathbf{x}^{(t)}$, and the hidden state of the previous time step, $\mathbf{h}^{(t-1)}$, are multiplied with their weight matrices (U and W, respectively) and added to a bias vector, $\mathbf{b}$:

$$\mathbf{a}_1^{(t)} = \mathbf{U} \cdot \mathbf{x}^{(t)} + \mathbf{W} \cdot \mathbf{h}^{(t-1)} + \mathbf{b}$$

- An activation function, f_1 (e.g., the ReLU function), is applied to the result, yielding the new hidden state:

$$\mathbf{h}^{(t)} = f_1(\mathbf{a}_1^{(t)})$$

- The new hidden state is multiplied with another weight matrix, $\mathbf{V}$, and added to another bias vector, $\mathbf{c}$:

$$\mathbf{a}_2^{(t)} = \mathbf{V} \cdot \mathbf{h}^{(t)} + \mathbf{c}$$

- Another activation function, f_2 (e.g., the softmax function), is applied to the result, yielding the

network's output for this time step, $\mathbf{o}^{(t)}$:

$$\mathbf{o}^{(t)} = f_2(\mathbf{a}_2^{(t)})$$

As in the regular neural networks, learning can be done using the backpropagation algorithm. The procedure is the same as what we have seen before, by readjusting the weights in the opposite direction of the partial derivative of the loss function in respect to themselves. The only difference is that these steps must be repeated for the 3 weight matrices ($\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$) and for the 2 bias vectors ($\mathbf{b}$ and $\mathbf{c}$):

$$\begin{aligned}
\mathbf{u}^{\text{new}} &= \mathbf{u}^{\text{old}} - \eta \nabla_{\mathbf{u}} L(\mathbf{u}, \mathbf{v}, \mathbf{w} b, c) \\
\mathbf{v}^{\text{new}} &= \mathbf{v}^{\text{old}} - \eta \nabla_{\mathbf{v}} L(\mathbf{u}, \mathbf{v}, \mathbf{w} b, c) \\
\mathbf{w}^{\text{new}} &= \mathbf{w}^{\text{old}} - \eta \nabla_{\mathbf{w}} L(\mathbf{u}, \mathbf{v}, \mathbf{w} b, c) \\
b^{\text{new}} &= b^{\text{old}} - \eta \nabla_{b} L(\mathbf{u}, \mathbf{v}, \mathbf{w} b, c) \\
c^{\text{new}} &= c^{\text{old}} - \eta \nabla_{c} L(\mathbf{u}, \mathbf{v}, \mathbf{w} b, c)
\end{aligned}$$

The vanilla RNN implementation, although effective in theory has a practical problem. The information the RNN stores in a given iteration is almost totally forgotten after a certain number of iterations of the forward and backpropagation algorithms. This works well if we only need to memorize the last few data points, but if we want to remember something that happened a long time ago, we need to use a different approach. To solve that problem, a new kind of units to replace standard recurrent units was developed, called Long Short-Term Memory (LSTM), which show a much better capacity of keeping useful knowledge from past time steps. Its implementation details are out of the scope of this work.

2.3.4 Deep Q-Network

The Deep Q-Network (DQN) [9] is an RL method that takes advantage of the properties of deep neural networks to estimate the Q-values more efficiently. Essentially, it consists in Fitted Q-Iteration algorithm as we described in Section 2.2.11 using neural networks for both the estimator and the target estimator.

One of the main advantages of the DQN is that can be used in problems where the state space is continuous. Figure 2.3 provides an overview of its architecture, supposing there is one hidden layer. When the DQN is provided with an environment state $x \in X \subseteq \mathbb{R}^M$, it is multiplied by the first weights matrix $\mathbf{W}_1$ and added to the first bias vector $\mathbf{b}_1$, generating the first net value net_1. An activation function (ReLU, in our example) is applied to the net value, and the result is passed on to the next layer. The process repeats until the output layer is reached, where the net value corresponds directly to the estimated Q-values, $\hat{Q}$, without the need to apply any activation function.

Additionally, DQN does not compute the full optimization described in line 13 of Algorithm 2.1. In-

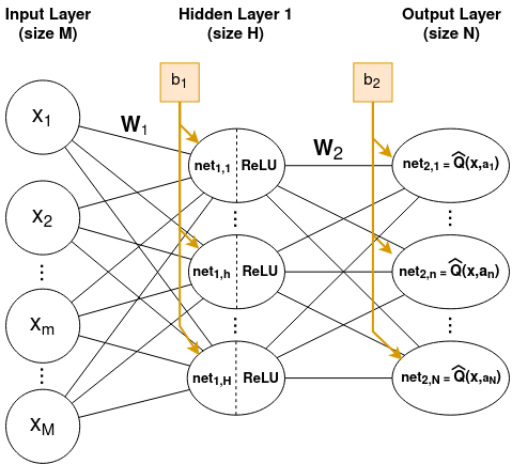


Figure 2.3: Simple representation of a DQN with one hidden layer. Source: Primary.

stead, it performs a single step of the backpropagation algorithm, as in Equation (2.12), where the loss function L used is the MSE:

$$\mathbf{W}_{1,k,t+1} = \mathbf{W}_{1,k,t} - \eta \nabla_{\mathbf{W}_1} L(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2)$$
$$\mathbf{b}_{1,k,t+1} = \mathbf{b}_{1,k,t} - \eta \nabla_{\mathbf{b}_1} L(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2)$$
$$\mathbf{W}_{2,k,t+1} = \mathbf{W}_{2,k,t} - \eta \nabla_{\mathbf{W}_2} L(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2)$$
$$\mathbf{b}_{2,k,t+1} = \mathbf{b}_{2,k,t} - \eta \nabla_{\mathbf{b}_2} L(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2)$$

2.3.5 Deep Recurrent Q-Network

Like the DQN, the aim of the DRQN [10] is also to solve RL problems using deep learning techniques. However, it features RNNs, instead of regular neural networks. This allows it to extract sequential information from the succession of environment states, as we discussed in Section 2.3.3. This property makes it a good candidate for solving POMDPs, since partial observability makes the optimal policy be history-dependent.

The DRQN algorithm is very similar to the DQN algorithm. Here we present the main changes when we introduce recurrence:

Both the estimator and target estimator for the DRQN are RNNs instead of regular neural networks;

The items pushed to the replay buffer in the DRQN are sequences of transitions of a fixed length S:

$$D \leftarrow D \cup \{T_{k,t'} = (x_{k,t'}, a_{k,t'}, r_{k,t'}, x_{k,t'+1}), t' = t, ..., t + S - 1\}$$

We discuss in more detail the parameterization of the DRQN, when we describe our novel algorithm, in Section 4.2.

3

Related Work

Contents

3.1 *Ad hoc* teamwork

In this section we will explore state-of-the-art algorithms used to solve the *ad hoc* teamwork task. We will start by exploring the first works in this field, and find out how they culminated in the creation of the most important architectures in this field - the Planning and Learning to Adapt Swiftly to Teammates to Improve Cooperation (PLASTIC) architecture - and its two main implementations - PLASTIC-Model and PLASTIC-Policy. Then, we will see how more recent approaches were designed to deal with the partial observability problem. We will finish by examining a competitive alternative to the PLASTIC architecture based on attention networks.

3.1.1 Early Work

The first work we will address in the field of *ad hoc* teamwork is by Stone et al. [11] and it was one of the pioneering texts to recognize the importance of deepening our research on *ad hoc* autonomous agent teams. The *ad hoc* teamwork problem is posed to the community as: "To create an autonomous agent that is able to efficiently and robustly collaborate with previously unknown teammates on tasks to which they are all individually capable of contributing as team members." The authors go even deeper, by stating that a robust *ad hoc* team agent should be able to:

1. "Identify the full range of possible teamwork situations that a complete ad hoc team player needs to be capable of addressing."
2. "For each such situation, find theoretically optimal and/or empirically effective algorithms for behavior."
3. "Develop methods for identifying and classifying which type of teamwork situation the agent is currently in, in an online fashion."

Furthermore, they propose a performance evaluation method for an *ad hoc* autonomous agent based on its capability to replace the role of a random existing teammate from a cohesive team, while trying to maximize a certain score measure.

After this, Barrett et al. [12] published a followup work, in which the first empirical evaluation of an *ad hoc* team agent was provided. The domain used was *Pursuit* (Fig. 3.1). This environment consists of a square grid where four agents (the predators) must coordinate to capture a fifth agent (the prey), which moves randomly. The four predators cannot communicate with each other, and they do not learn, which means that each one of them is known to have a fixed policy (i.e., one that does not change over the course of the simulation).

The authors had the *ad hoc* agent play as one of the four predators, while its three teammates can follow one from a wide range of possible behaviors - some known, and others unknown by the agent *a*

priori.

Two planning algorithms are tested: VI, which we have seen before, and Monte Carlo Tree Search (MCTS). MCTS is an online planning algorithm which works by successively performing simulations in a search tree, as shown in Fig. 3.2, to enhance its knowledge about the expected return of each possible sequence of actions. In order to perform those simulations, the algorithm needs to model the uncertainty in the environment, which, in this particular environment, exists both in the prey and the teammates' behavior. The prey, however, is known to act following a uniform distribution over the possible actions. To main challenge is to model the teammates behavior, which is done using Bayes' theorem (assuming the teammates' possible models are previously known by the agent). At each time step t, to estimate the posterior probability $P_t(m \mid a_t)$ of each model m given the joint action a_t, the likelihood of each joint action given a teammate model, $P_t(a_t \mid m)$ is multiplied by the prior distribution over the teammate models, $P(m)$, and divided by $P(a_t)$, which works as a normalization factor:

$$P_t(m \mid a_t) = \frac{P(a_t|m)\dot{P}(m)}{P(a_t)}$$

The evaluation method is adapted from the one in [11], with multiple experiments done for different combinations of the planning algorithm, the teammates' behavior and the grid size. The results shown that MCTS allows for efficient planning when compared to VI given that it has access to a known set of teammate models, even if these models are faulty, or the actual models being used by the teammates differ from the ones in the set.

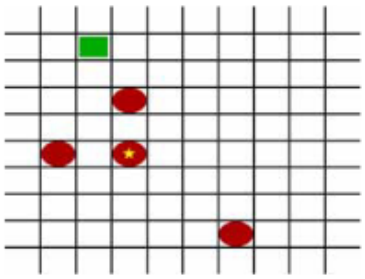


Figure 3.1: The "Pursuit" domain. In this scenario, four predators (red circles) must coordinate to surround the prey (green square). The agent being evaluated is marked with a star. Source: Adapted from [3].

Later, Barrett et al. [3] developed the first *ad hoc* team agent capable of autonomously learning its teammates' models. They describe a novel algorithm based on *transfer learning*, adapted to cases where the observations the agent has about its (potential) teammates are limited. Transfer learning techniques consist in having an agent store the knowledge it acquired in an RL task to reuse it in a different one.

The authors use, again, the "Pursuit" domain to test their algorithm, and the method defined in [11] to evaluate it. In the construction of the algorithm, they assume the *ad hoc* agent knows the representation

of both the environment and the prey, but not that of its teammates.

The planning algorithm used is Upper Confidence bound applied to Trees (UCT), which is an MCTS algorithm (Fig. 3.2) that has been shown to be effective in complex domains like large POMDPs and GO, where the branching factor is high. To perform each simulation, the agent must make an assumption about its teammates' models, and it does so using a Bayesian approach. To select the set of models, multiple approaches are followed, but the most relevant for our case is the one where transfer learning is used, as the models used in Bayesian Online Prediction for Ad hoc teamwork (BOPA) and Ad hoc Teamwork under Partial Observability (ATPO) (as we will see in Section 3.1.3) could be seen as a byproduct of interactions with former teammates. The authors empirically concluded that this transfer learning approach can enhance the *ad hoc* agent's performance even in cases where there is a small amount of data available.

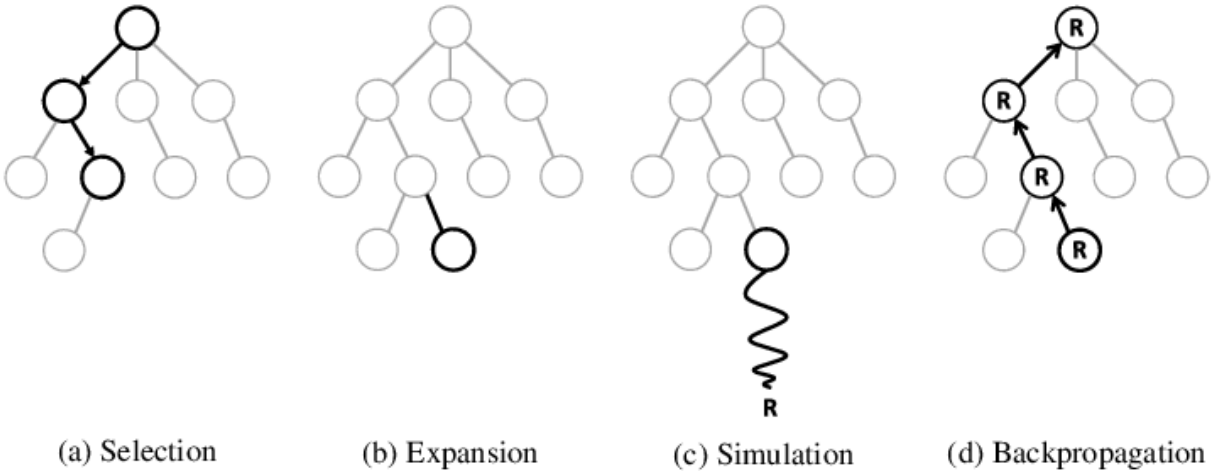


Figure 3.2: Phases of the Monte Carlo tree search algorithm. A search tree, rooted at the current state, is grown through repeated application of the above four phases. Source: Adapted from [4].

3.1.2 The PLASTIC architecture

The PLASTIC architecture, by Barrett [5], is one of the most popular methods for *ad hoc* teamwork, due to its robustness to task and teammate diversity, and its capability to efficiently adapt to those diverse situations. It is an algorithm which assumes the *ad hoc* agent has past experience cooperating with other teammates. When faced with new teammates, the agent tries to identify the past team with a highest similarity to the current one and reuses the knowledge it got from the most similar past team to act upon the domain.

In order to better leverage information from different data sources, the author introduced a new transfer learning algorithm, "Two-Stage Transfer" (used by the PLASTIC algorithm), which tries to identify the best weighing for the importance of each data source.

The author introduces two variants of the PLASTIC architecture, which we will proceed to describe, called PLASTIC-Model and PLASTIC-Policy.

3.1.2.A PLASTIC-Model

The first variant we will see is PLASTIC-Model, which functions as schematized in Figure 3.3.

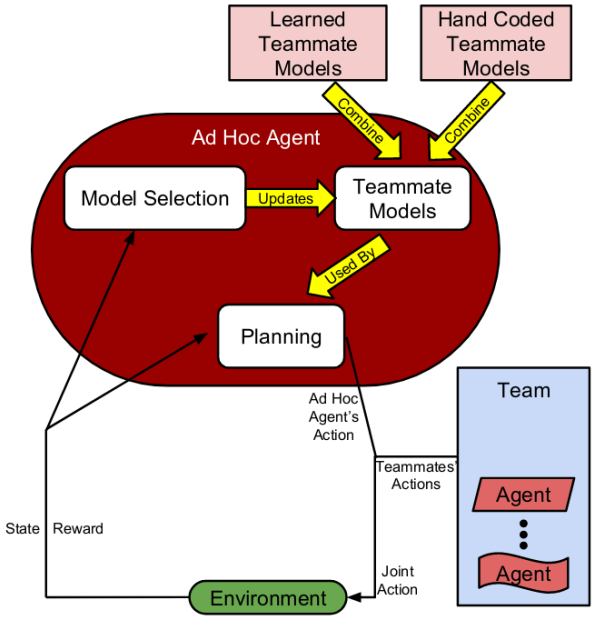


Figure 3.3: Overview of the PLASTIC-Model algorithm. Source: Adapted from [5].

The PLASTIC-Model algorithm starts by learning transition models for a set of teammates using a supervised learning approach, i.e., an approach where the aim is to predict a target output variable given an input, after being presented with a sequence of (input, output) tuples. In PLASTIC-Model's case, the inputs are the features extracted from environment states, and the outputs are the teammates' actions. The agent performs this learning offline, thus representing its past knowledge about these teammates.

Then, these learnt transition models are combined with hand-coded models from other data sources using their novel "Two-Stage Transfer" algorithm. The resulting models are then used by the *ad hoc* agent's online planner to plan the best course of action, using the UCT algorithm we mentioned before.

To select which model is used by the planner, the agent stores a belief distribution over which of the teammate models is more likely to be the one the agent is currently working with. Since, in the presence of previously unseen teammates, this belief distribution represents, instead, the similarity between the currently observed model and the known ones, they named it *behavior distribution*, which covers both the cases of seen, and unseen teammates.

The behavior distribution for each model m is then updated using polynomial weights, as follows:

$$L(m, x, a, x') = 1 - P(a \mid x, m, x')P(m \mid x, a, x') = \frac{(1 - \eta * L(m, x, a, x'))}{c} \tag{3.1}$$

where $L(m, x, a, x')$ is the loss of model m for observing the transition (x, a, x'), $\eta \leq 0.5$ is a parameter bounding the maximum value for the loss, and c is a normalization constant.

Empirical evaluation shown that PLASTIC-Model can quickly adapt to previously unseen teammates in simple domains. The two main weaknesses of this algorithm are (i) that it assumes the agent has access to its teammates' actions (since it observes the joint actions a), and (ii) that UCT, being a stochastic algorithm, is prone to learning inaccurate environment models. On the other hand, more accurate planning algorithms would most likely be way too ineffective to be used online in larger domains.

3.1.2.B PLASTIC-Policy

The second, and last variant we will see is PLASTIC-Policy, which functions as schematized in Figure 3.4.

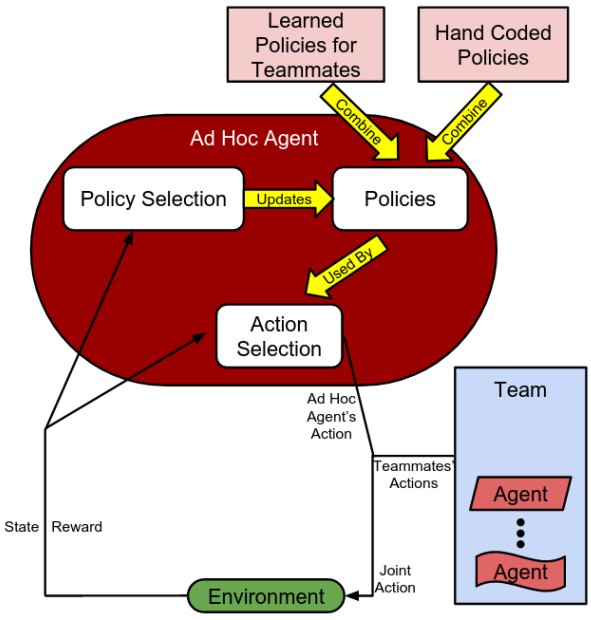


Figure 3.4: Overview of the PLASTIC-Policy algorithm. Source: Adapted from [5].

Instead of learning teammate models, PLASTIC-Policy starts by directly learning policies to work with a given set of teammates, removing the hurdle of online planning. PLASTIC-Policy represents these policies in the form of a matrix of Q-values which is learnt using an Fitted Q-Iteration (FQI) implementation, similar to the one we described in Section 2.2.11. The estimator used by the author consists

in a weighed sum of a set of binary features f_i extracted from the environment state:

$$\hat{Q}(x, a) = \sum_{i=1}^{M} w_i f_i$$

where the weights w_i are learnt by FQI and M is the number of features f_i. To build the replay buffer for each teammate in the known set, the agent performs exploratory actions in the domain, and stores in it the resulting transitions of the form (x, a, r, x'). Note that, since the agent does not need to predict its teammates' actions, a can simply be the agent's own action.

In parallel to learning the policies, the agent also learns a nearest neighbor model for each past teammate, which maps states to the next state whose previous state is the closest to the state to be mapped. We explain this in more detail in Section 4.1.

The agent then combines this knowledge with that from other sources using, again, the "Two-Stage Transfer" algorithm. The resulting policies are then used to act in the environment, and, to update which policy the agent follows, an analogous approach to that of PLASTIC-Model is followed, where the agent keeps a behavior distribution over the known policies, and updates it using Equation (3.1), where m is now a nearest-neighbor model m_{NN}. Since the agent now does not have access to its teammates' behavioral models, it instead estimates $P(a \mid x, m, x') = P(a \mid x, m_{\text{NN}}, x')$ by finding the state $\hat{x}$ in m_{NN} closest to x, and computing the similarity between x' and $\hat{x}$ (the next state for $\hat{x}$). The similarity can be computed by $\frac{|x' - \hat{x}'|}{D}$, assuming that all states x have bounded values and are normalized, being D a normalization constant.

PLASTIC-Policy was tested by Barrett et al. [7] in the HFO domain - the same we used for our experiments - making this work a very important baseline for us to follow.

After experimenting in scenarios with 2 attacking versus 2 defending agents and with 4 attacking versus 5 defending agents, the authors concluded that the algorithm quickly converged to the correct policy in both scenarios. Due to its scalability, its capability for quick adaptation to new teammates, and its portability for different *ad hoc* teamwork scenarios, PLASTIC-Policy has become one of the most popular approaches in the field.

3.1.3 Introducing partial observability in *ad hoc* teamwork settings

The approaches we have seen until now assume the agent can totally observe the state of the environment, which is usually not true in practice. We will now see two examples, one with total and other with partial observability, and learn how that difference impacts the choice of architecture.

First, we have the one from Ribeiro et al. [13] where the algorithm BOPA is presented. This algorithm is designed for simple, totally observable domains with a random reward function, where an *ad hoc* agent must cooperate with its teammates, who know the optimal policy for the underlying MMDP. The *ad hoc*

agent, however, does not know that policy, and must therefore extract information from its teammates' behaviour.

The algorithm uses a Bayesian approach to select the action with the highest likelihood, by computing the sum of the optimal policies for each task (defined by its reward function), weighed by the probability of each task:

$$\pi_t(a^\alpha \mid x_t) = P[A_t^\alpha = a^\alpha \mid X_t = x_t] = \sum_{m=1}^{M} \pi_m(a^\alpha \mid x_t)\; p_t(m)$$

where $p_t(m)$ is computed iteratively by using the state transition probabilities $\mathbf{P}_m$:

$$p_t(m) = P[R = r_m \mid H_t = h_t] = \frac{1}{Z}\, \mathbf{P}_m(x_t \mid x_{t-1}, a_{t-1}^\alpha)\; p_{t-1}(m)$$

with Z being a normalization constant.

The authors tested their algorithm both in a simulated and in a real-world scenario, and it was able to effectively identify and solve the correct task in both scenarios. Since in the simulated environment the algorithm performed well for different problem sizes, the authors claim the algorithm is also scalable. However, they only experimented with relatively small problem sizes, so, this might not be the case for much larger domains.

Following the work of Ribeiro et al., Martinho [14] presented an extension of BOPA for environments with partial observability, named ATPO. Like BOPA, ATPO also follows a Bayesian approach. The author assumes there is a random task M that an *ad hoc* agent has to perform in cooperation with other teammates, and defines

$$p_t(m_k) = P[M = m_k \mid H_t = h_t]$$

as the probability that the task M being performed is m_k given the action-observation history h_t. At each time step, the algorithm selects the action with the highest likelihood, computing the sum of the optimal policies for each task, weighed by the probability of each task:

$$\pi_t(a^\alpha \mid h_t) = P[A_t^\alpha = a^\alpha \mid H_t = h_t] = \sum_{k=1}^{K} \hat{\pi}_k(a^\alpha \mid b_{k,t})\; p_t(m_k)$$

The core of the ATPO algorithm is in the update of the probabilities of each task and of the beliefs over the states, where it takes into account not only the observation ($\mathbf{O}_k$) and state transition ($\mathbf{P}_k$) probabilities, but also the policy π_t computed earlier:

$$p_{t+1}(m_k) = \frac{1}{\rho} \sum_{x,y \in \chi_k} \mathbf{O}_k(z_{t+1} \mid y, a_t^\alpha)\; \mathbf{P}_k(y \mid x, a_t^\alpha)\; b_{k,t}(x)\; \pi_t(a^\alpha \mid h_t)\; p_t(m_k)$$

$$b_{k,t+1}(y) = \frac{1}{\rho} \sum_{x \in \chi_k} b_{k,t}(x)\; \mathbf{P}_k(y \mid x, a_t^\alpha)\; \mathbf{O}_k(z_{t+1} \mid y, a_t^\alpha)$$

where ρ is a normalization constant.

The algorithm is tested in the "Pursuit" domain, and compared against agents that use optimal or near-optimal policies. Multiple experiments were conducted, namely for cases where the agent does not know its teammates' behaviour, and for cases where the agent does not know the correct capture task. The author concluded that ATPO performed:

- Near-optimally when trying to identify the teammates' behaviour, "being 9.67% slower than the optimal policy"
- "39% slower than the optimal policy" when trying to identify the capture task
- "57% slower than the optimal policy" when trying to identify both the teammates' behaviour and the correct capture task.

Although the results for the case where the agent does not know the correct capture task are not perfect, the ones for the case where it does not know its teammates' behaviour are very close to being optimal, which is very encouraging, since this problem's formulation is very similar to ours, with the most striking difference being that we will consider a larger, continuous state space.

3.1.4 Other architectures for the *ad hoc* teamwork problem

Chen et al. [6] developed an attention network algorithm that surpassed PLASTIC-Policy's performance in the HFO domain. The algorithm was named by the authors Achieving the Ad-hoc Teamwork by Employing the Attention Mechanism (AATEAM) and, like POPP uses RNNs. However, unlike the algorithms we previously seen, AATEAM is designed to adapt to the *ad hoc* agent's new teammates in real-time, with the aid of attention-based RNNs. The architecture consists of multiple networks like the one in Fig. 3.5, one for each previously known teammate type, and each network consists, mainly, of two parts, called an "encoder" and a "decoder".

The encoder, at each time step, receives a sequence with the most recent environment states and outputs an encoded value for each state. Each of these values has in consideration the input from the few previous states, since the encoder has a layer with a hidden state. The decoder uses the information outputted by the encoder, together with its hidden state, to output an action. This description is only a simplified version of the actual algorithm used in the paper. Its complete details are out of the scope of this work.

The results shown that AATEAM clearly over-performed PLASTIC-Policy both with known and unknown teammates. As we can see in Fig. 3.6, AATEAM (in blue) always achieved a higher scoring rate than PLASTIC-Policy (in yellow). The difference between the two was always higher than 1%, except for two teammate types, "gliders" (known) and "ubc" (unknown), for which AATEAM only slightly surpassed PLASTIC-Policy.

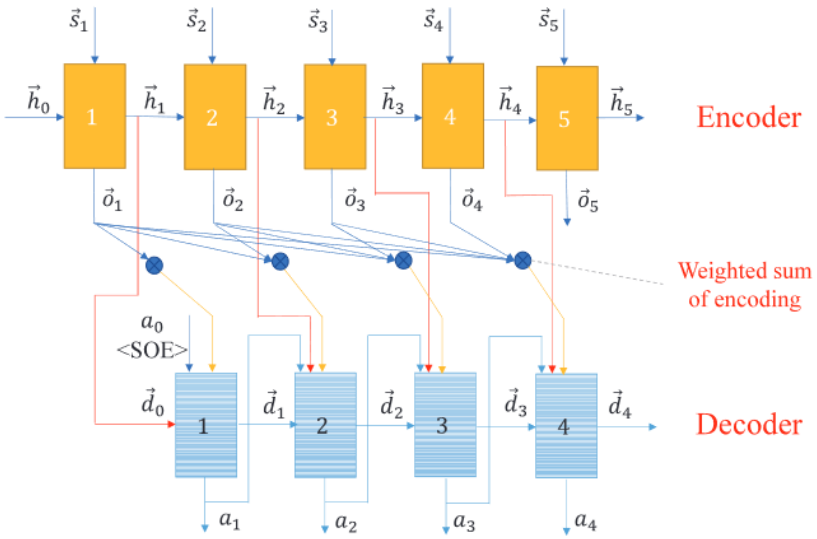


Figure 3.5: Simplified description of the attention network used by AATEAM. Source: Adapted from [6].

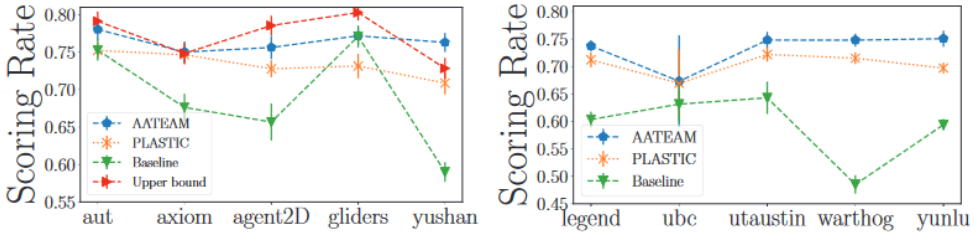


Figure 3.6: Performance of AATEAM with known teammates (on the left) and unknown teammates (on the right). Source: Adapted from [6].

3.1.5 Conclusion

We have started by exploring how the research in the *ad hoc* teamwork field began, and how it became so relevant nowadays, that it prompted the development of innovative and effective approaches to tackle the *ad hoc* teamwork task.

We have seen that Bayesian approaches are able to identify the correct task for an *ad hoc* teamwork scenario in low-dimensional, discrete domains, under total (BOPA) or partial (ATPO) observability. We have also seen how transfer learning techniques can be very effective in generalizing teamwork experiences, both in simple (Barrett et al. [3]) and complex (Barrett [5] and Chen et al. [6]) domains.

In particular, the PLASTIC-Policy architecture has become the *de facto* standard for *ad hoc* teamwork, despite having been surpassed by AATEAM, a more custom-tailored approached for the HFO domain, due to the former being able to adapt to a great variety of tasks and teammates, and especially for doing so very efficiently.

4

POPP

Contents

4.1 The PLASTIC-Policy Architecture

In this section, we will present the PLASTIC-Policy architecture in greater detail, since it will be the basis for our novel architecture. An overview of PLASTIC-Policy can be found in the previous chapter (Figure 3.4), but here we present a more thorough explanation.

Algorithm 4.1: The PLASTIC-Policy algorithm. Source: Adapted from [7].

Procedure `PLASTIC-Policy(`KnownTeammates`)`**:**
  $\Pi \leftarrow$ `LearnPolicies` (KnownTeammates)
  $M \leftarrow$ `LearnNNModels` (KnownTeammates)
  `ActInDomain` (KnownTeammates, Π, M)

Function `LearnPolicies(`KnownTeammates`)`**:**
  $\Pi \leftarrow \{\}$
  foreach teammate $\beta \in$ KnownTeammates **do**
    Learn policy π to cooperate with β
    $\Pi \leftarrow \Pi \cup \{\pi\}$
  return Π

Function `LearnNNModels(`KnownTeammates`)`**:**
  $M \leftarrow \{\}$
  foreach teammate $\beta \in$ KnownTeammates **do**
    Learn nearest neighbor model m_{NN} of β
    $M \leftarrow M \cup \{m_{\text{NN}}\}$
  return M

Procedure `ActInDomain(`KnownTeammates, Π, M`)`**:**
  $\mu =$ UniformDistribution(KnownTeammates)
  Initialize state x
  while x is not terminal **do**
    $\beta = \operatorname{argmax} \mu$
    Take action $a = \Pi_\beta(x)$ and observe r, x'
    $\mu =$ `UpdateBehaviorDistribution` (KnownTeammates, Π, M, μ, x, a, x')

Function `UpdateBehaviorDistribution(`KnownTeammates, M, μ, x, a, x'`)`**:**
  foreach teammate $\beta \in$ KnownTeammates **do**
    $L(M_\beta, x, a, x') = 1 - P(a \mid x, M_\beta, x')$
    $\mu_\beta = \mu_\beta(1 - \eta L(M_\beta, x, a, x'))$
  Normalize μ
  return μ

Algorithm 4.1 is divided in three parts: LearnPolicies, LearnNNModels and ActInDomain. We will proceed to explaining each one of them.

4.1.1 LearnPolicies

First, the agent learns a set of policies Π to work with the previously encountered KnownTeammates. It does so using the FQI algorithm, as explained in Section 2.2.11. We will specify the type of estimator used in Section 4.1.4.

4.1.2 LearnNNModels

Then, the agent will learn a set of nearest neighbor models M for each teammate in KnownTeammates. Learning a model consists in interacting with the environment whilst cooperating with a known teammate β, storing tuples of the form (x_t, x_{t+1}) in M_β.

4.1.3 ActInDomain

This is the core of the PLASTIC-Policy algorithm. Here, the agent is placed in an *ad hoc* scenario, and must cooperate with the new teammate (which might be known or unknown), leveraging the previously acquired knowledge. The agent starts by initializing a behavior distribution vector μ to a uniform distribution over the known teammates (since in our work, for simplicity, we assume the teammates follow a uniform distribution). Each entry in this vector represents how similar the current teammate's behavior is to each previously seen teammate. The agent then starts interacting with the environment, and, at each iteration, starting in state x, takes the action a prescribed by the policy corresponding to the previously encountered teammate with the highest belief distribution (in case of a tie, it selects one at random among the tied ones), and observes r and x'. With this information, it will then proceed to updating the behavior distribution μ, and it does so, using the polynomial weights update we described in Equation (3.1). The term $P(a \mid x, M_\beta, x')$ is computed as we described in Section 3.1.2.B.

4.1.4 PLASTIC-Policy with a DQN

In order to compare our performance with a similar, but non-recurrent approach, we implemented a version of PLASTIC-Policy using a DQN, and called it Deep Q-Network - PLASTIC-Policy (DQN-PP). The DQN algorithm is explained in Section 2.3.4.

4.2 Introducing Recurrence in PLASTIC-Policy: POPP

We finally reach the point were we explain our novel algorithm, POPP. In essence, it consists in an implementation of PLASTIC-Policy using a DRQN (as explained in Section 2.3.5) as the estimator. But there are some implementation details that are worthy to note.

Firstly, to choose the action during the LearnPolicies part of the algorithm, we used an ϵ-greedy policy (explained in Equation (2.8)), with ϵ being an experimental parameter.

Secondly, we used the replay buffer to store the transitions that the agent would then use to learn the nearest neighbor models (in function LearnNNModels). We could only do this, since we realized the replay buffer was never totally filled up after the end of the LearnPolicies function. In this way, we saved memory space, by avoiding the use of an additional buffer to store the $(x_t, x_{t+1}$ tuples.

Thirdly, since the environment we chose (HFO) has different valid actions for different environment states (e.g., an agent can only pass the ball to a teammate, or shoot the ball if it is currently controlling it), we changed the agent's action selection mechanism. When the agent is selecting a random (exploratory) action, it selects instead a random action among the valid ones for that state. When the agent is following the greedy action, it selects the one with the highest Q-value from among the valid ones. In this way, we prevent the agent to select invalid actions, which would hinder the learning process.

Last, but not least, since with partial observability we cannot access the full state of the environment, we use the observations $z \in Z$, instead of the states $x \in X$ to create the nearest neighbor model for each teammate.

5

Experimental evaluation

Contents

In this chapter, we describe our evaluation procedure and results. We start by describing the chosen environment, HFO, in Section 5.1. After that, we proceed to analyzing how well DQN and DRQN agents perform with total (Section 5.4.1.A) and partial (Section 5.4.1.B) observability. We then describe the evaluation procedure for our *ad hoc* team agent, the POPP agent, and its performance with different levels of environment observability and of similarity between past and present teammates. Finally, based on our results, we provide answers to the questions:

1. Can DRQN surpass a non-recurrent (DQN) architecture's performance in a complex, totally observable scenario with sparse rewards?

2. Can DRQN surpass a non-recurrent (DQN) architecture's performance in a complex, partially observable scenario with sparse rewards?

3. How accurately can POPP identify teams it previously encountered?

4. Can POPP surpass a non-recurrent (DQN-PP) architecture in a complex, partially observable scenario with sparse rewards?

5. How does POPP's performance change with the level of similarity between the current and past teammates?

6. How does POPP's performance compare with that of the original teams?

5.1 Half-Field Offense

The environment we chose to evaluate our agent was HFO, which is a task that plays out in the offense half of a soccer field, where a offense team must coordinate to score a goal before time runs out, and a defense team must coordinate to prevent the offense team to score. We used the HFO implementation by Hausknecht et al. [15]. Figure 5.1 presents a snapshot of an HFO match.

Figure 5.1: Snapshot of a Half-Field Offense ongoing match. The yellow (offense team) is trying to score against the red and purple (defense) team. Source: Primary.

An HFO match ends in one of the following terminal states:

- **Goal**, if the ball enters the defense team's goal;
- **Captured by Defense**, if an agent of the defense team manages to get control of the ball;
- **Out of Bounds**, if the ball leaves the playing field through one of the four lines limiting it;
- **Out of Time**, if a set amount of time steps have passed without any of the previous terminal states being reached.

If the terminal state is "Goal", the offense team is considered the winner; otherwise, victory goes to the defense team.

We chose this environment for multiple reasons:

- It is a multi-agent system, including both cooperating (teammates) and adversarial (opponents) agents;
- It is a complex domain, with a continuous state space;
- It supports total and partial observability;
- It has sparse rewards;
- There is a wide range of teammate implementations available to use;
- It was used on previous works, allowing us to better compare results.

5.1.1 Environment Parameterization

In HFO there are multiple parameters that impact the way the game plays out. In Table 5.1, we listed some of those parameters along with the values we chose for each of them.

Table 5.1: Description and chosen values for the parameters of the HFO environment

Parameter	Value	Description
--frames-per-trial	500	Number of frames (time steps) the offense team has for scoring, before time runs out and they lose the match.
--untouched-time	100	If the offense team does not touch the ball for this amount of (consecutive) frames, they lose the match.
--no-sync	True	If True, there are no restrictions to how long agents take to choose their actions. If False, the match is played in real-time, and agents have a fixed amount of time to choose their actions; if they take more than the time limit, no action is performed in that time step.

5.1.2 Environment Model

The HFO environment can be modeled as an MPOMDP $(I, X, A, P, Z, O, r, \mu_0)$. We will provide a high-level description for each of its components in the next few sections.

5.1.2.A Agents

The index set of agents I contains indices representing all agents from the offense and defense teams. In all experiments we will present in this work, we considered 2 agents in each team, so, we have always that $|I| = 4$. We can classify the agents in I regarding their role, or their policy.

Regarding each agent's role, we can split I in two subsets, such that $I = I^\omega \cup I^\delta$, where I^ω contains the offense agents and I^δ the defense agents.

Regarding their policies, each agent is from one of the following types:

- Learning agents (DQN/DRQN), which we will call γ;
- *Ad hoc* agents, which we will call α;
- Agents for which the binary files were made available, which include those created for the RoboCup 2D Simulation League 2013 ("aut", "axiom", "cyrus", "gliders" and "helios"), and the type "base" (also known as "Agent2D"), that comes with the HFO simulator. We will call them *binary agents*, and index them as $\beta_k, k \in \mathbb{N}$, since in all our experiments there are multiple binary agents.

We made three types of experiments:

- **Learning + Binary:** $I^\omega = \{\gamma, \beta_2\}$;
- ***Ad hoc* + Binary:** $I^\omega = \{\alpha, \beta_2\}$;
- **Binary + Binary**: $I^\omega = \{\beta_1, \beta_2\}$;

In all our experiments, the 2 defense agents ($I^\delta = \{\beta_3, \beta_4\}$) were of the binary type "base".

5.1.2.B State

The HFO simulator makes available two list of features: the *Low Level State Feature Set* and the *High Level State Feature Set*. Those feature sets represent what information each agent will get about the state on each time step, so, they represent the agents' observations, and not the state itself. Therefore, in this section, we try to estimate the number of features that define a state of a HFO match.

The Low Level State Feature Set contains a total of $50 + 9|I| = 86$ features. However, some of the features in that set are redundant for defining the environment state since they either:

- **(i)** do not change throughout the whole match (uniform numbers) - a total of $|I| - 1 = 3$ features;

- **(ii)** simply indicate whether an agent's perception of another feature is valid - a total of 4 features;
- **(iii)** can be deduced from other features - a total of 33 features.

There are also $7(|I|-1)=21$ features that contribute to defining the environment state, but are unavailable in the feature set of a given agent (e.g. other agents' stamina or whether they are able to kick the ball).

Therefore, we conjecture that a state of a HFO match can be fully described by $|X|=15|I|+7=67$ features. The features are all either Boolean (in $\{-1,1\}$), or real (floating point) numbers (in $[-1,1]$). These features represent the position, velocity and angle of each entity (agents or the ball), and other minor features like whether each agent is colliding with a landmark (e.g. a goal post).

We will describe the High Level State Feature Set in greater detail (in Section 5.1.2.E), since it is the one we chose for agents to have access to.

5.1.2.C Actions

HFO features 3 types of actions:

- Low Level actions, which represent simple, primitive movements (e.g. Turn(angle), Kick(power, angle)), and have continuous parameters;
- Mid Level actions, which implement higher level behavior (e.g. Intercept(), DribbleTo(x, y)), having almost all of them continuous parameters;
- High Level actions, which implement strategic behavior (e.g. Move(), DefendGoal()), and are discrete.

We chose to use the High Level actions and the Mid Level action Intercept(), mainly due to being discrete, allowing us to directly apply the DQN/DRQN algorithm. The full list of actions we considered for our custom agents (learning and *ad hoc* agents) is:

Intercept() : Moves the agent so as to intercept the ball's movement, taking its speed into account.

Move() : Moves the agent in order to implement a strategy (that of the "helios" team).

Shoot() : Takes a shot facing the direction with the highest possible opening angle (i.e., the angle between two obstacles, which can be opponents or goal posts).

Pass(n) : Passes the ball to the teammate with uniform number n, if the agent can see teammate n. Otherwise, it does nothing.

Dribble() : Executes a chain of short kicks and moves in order to advance the ball towards the goal.

The actions Intercept() and Move() can only be executed when the agent in unable to kick the ball, whilst Shoot(), Pass(n) and Dribble() can only be executed when the agent is able to kick the ball. Whenever

an agent selects an action that cannot be executed given the current environment state, that action has no effect, being equivalent to the agent choosing the action NoOp(), which does nothing.

Due to the sparsity of rewards (only in terminal states), we realized the agents γ were having trouble learning a policy. Therefore, we developed a way to reduce the number of time steps the learning agents see during an episode. Since, most of the time, the agent does not control the ball, most of the time steps the agent is just executing the actions "Intercept()" or "Move()", and sometimes alternating between both, resulting in a poor performance. So, we decided to introduce *repeated actions*, which consist in actions that, when selected by an agent, are repeatedly executed for a fixed number of time steps, or when the agent regains control of the ball. Moreover, during the execution of this action, the agent does not get any observation or reward - that only happens when the repeated action ends. Thus, we were able to increase our agent's performance by changing the agent's actions Intercept() and Move() to Repeat[Intercept(), 5] and Repeat[Move(), 15], respectively, where the number of repeats of each action was chosen empirically.

With this said, the action sets for the learning (A^{γ}) and *ad hoc* (A^{α}) agents are always Repeat[Intercept(), 5], Repeat[Move(), 15], Shoot(), Pass(n^{β_2}), Dribble(), where n^{β_2} is the uniform number of the agent's only binary teammate β_2. So, $|A^{\gamma}| = |A^{\alpha}| = 5$. We do not have access to the implementation of the binary agents, so we assume A^{β_i} can contain every Low, Mid or High Level action (except for the actions that are exclusive to defensive agents).

The joint action set is therefore $A = \bigtimes_{i \in I} A^i$, with each set A^i as described above.

5.1.2.D Transition Probabilities

The transition probability function P results from the combination of the physics simulated by HFO with the join actions chosen by the agents.

We consider that a state transition begins right after an agent starts executing action, and ends when the agent reaches the next state after finishing executing its action, or when it reaches a terminal state. Therefore, given a transition (x, a, r, x'), starting at time step t_1 and ending at time step t_2 we have that:

- $x = x_{t_1}$ is the agent's original state in time step t_1;
- $a = a_{t_1}$ is the action the agent chose at time step t_1, and that lasted $t_2 - t_1$ time steps;
- $r = r_{t_2 - 1}$ is the reward the agent get for transitioning to state x_{t_2};
- $x' = x_{t_2}$ is the agent's new state in time step t_2.

5.1.2.E Observations

As we discussed in Section 5.1.2.C, we chose to use the High Level State Feature Set as the agents' observations. We made this choice mainly due to its features being, in general more straightforward for the agent to use in its decision process, especially if used in conjunction with the High/Mid Level actions. For instance, one of the High Level features is the *goal opening angle*, which indicates the largest possible angle the agent has to shoot the ball towards the goal, taking into account obstacles (opponents and goal posts). The High Level action Shoot() shoots the ball towards the direction with the highest possible goal opening angle. Therefore, the agent could use the goal opening angle feature to decide when to use the action Shoot().

The High Level State Feature Set has a total of $6 + 6|I^\omega| + 3|I^\delta| = 24$ features, but $I^\omega - 1 + I^\delta = 3$ of them are uniform numbers. Therefore, there are $7 + 5|I^\omega| + 2|I^\delta| = 21$ features which can aid the agents in their decision process. These features include not only simpler values, like each entity's position in the field (namely that of all agents and the ball), but also some complex to compute, but meaningful values, like the goal opening angle we described before. A detailed list of all High Level features can be found in Table A.1.

For our learning and *ad hoc* agents, we empirically chose to use the feature set containing all $7 + 5|I^\omega| + 2|I^\delta| = 21$ useful features, plus one Boolean validity feature per entity (including the ball and excluding the agent itself), indicating whether or not all of the other entities' features are valid. If all features that describe an entity are valid, its corresponding validity feature takes the value 1, otherwise it takes the value -1. By default, HFO sets the value of invalid features to -2, but since that could damage our learning agent's learning process (since it could interpret -2 as a valid, but very low value for that variable), we decided to set invalid features to the value 0, and introduce the validity features we just described.

With this said, we have that $|Z^\gamma| = |Z^\alpha| = 7 + 6|I^\omega| + 3|I^\delta| = 25$. The binary agents (including the defense agents) have access to all $|Z^{\beta_i}| = 24$ default features, but, in practice, they only use the 21 useful ones.

The joint observation set is therefore $Z = \bigtimes_{i \in I} Z^i$, with each set Z^i as described above.

5.1.2.F Observation Probabilities

Like P, the observation probability function O also results from the combination of the physics simulated by HFO with the joint actions chosen by the agents. For instance, if an opponent is blocking our agent's view of the ball or of other agent, the obstructed entity's features in our agent's observation will be invalid. However, there is an option in the HFO simulator (--fullstate), that makes agents able to access the correct values for all features. Yet, since our feature set does not contain the entities' velocities, there is still some degree of partial observability in the experiments we label as "totally observable".

5.1.2.G Reward Function

We empirically chose the following reward function r:

- 0 on all time steps where the state is not terminal;
- 10 on goal;
- -10 otherwise, i.e., if the defending team catches the ball, the ball goes out of bounds or time runs out.

5.1.2.H Distribution of the Initial State

Regarding μ_0, the offense agents and the ball start in random positions in the left half of the playable field; the defender starts in a random position in the right half of the field; the goalkeeper starts in the center of the goal.

5.2 Learning agent configuration

In Table 5.2, we present the parameters we have empirically chosen for both the DQN and the DRQN agents.

Table 5.2: Parameters chosen for the DQN and the DRQN

Parameter	DQN value	DRQN value
Input Size	25	25
Output Size	5	5
Number of Hidden Layers	1	1
Number of Units Per Layer	12	12
Type of Units	Linear	LSTM
Activation Function	ReLU	ReLU
Optimizer	Adam	Adam
Learning Rate	0.00025	0.00025
Initial Exploration Rate	0.5	0.5
Final Exploration Rate	0.05	0.05
Initial Exploration Rate	0.5	0.5
Discount Factor	0.995	0.995
Estimator Update Period	4	4
Target Estimator Update Period	500	500
Replay Buffer Sequence Length	–	10

5.3 Procedure and Metrics

In this section, we present our evaluation procedure and chosen metrics for our three types of experiments: Learning + Binary, *Ad hoc* + Binary and Binary + Binary.

5.3.1 Learning + Binary

In this type of experiments, where $I^{\omega} = \{\gamma, \beta_1\}$, we aimed to evaluate the learning agent γ's capacity to learn a policy to cooperate with a binary agent β_1. For each configuration (learning agent type, teammate type and observability), we ran the agent for 80 rollouts of 500 training episodes and 50 test episodes each, adding up to a total of 40,000 train episodes and 4,000 test episodes per experiment.

In the training episodes, γ learned using the DQN and DRQN algorithms we explained in Section 2.3.4 and Section 2.3.5, respectively.

In the test episodes, the agents did not learn, and always chose greedy actions (the valid actions with the highest Q-values).

At the end of each rollout, we saved the agent's current neural network weights and replay buffer (to reuse in the *ad hoc* experiments, and measured the offense team's score rate in the test episodes, i.e., the fraction of test episodes from the 50 of each rollout that ended in a goal. The results for these experiments can be found in Section 5.4.1.A and Section 5.4.1.B.

5.3.2 *Ad hoc* + Binary

In this type of experiments, where $I^{\omega} = \{\alpha, \beta_1\}$, we aimed to evaluate our novel POPP algorithm. For each configuration (type of learning agent, type of the teammates the agent will find, observability), we followed the POPP algorithm as described in Section 4.2:

- we started by the policies learnt when cooperating with each known teammate (which were generated using the procedure defined in Section 5.3.1). Since, during each experiment, we save the agent's network and replay buffer, we had to choose which state to use. In order to achieve a balance between the policy's performance and the number of observed time steps, we chose the state with the highest score rate among the last 5 states;
- we then learnt a nearest-neighbor model for each known teammate, using the transitions stored in the replay buffer (which are all of the transitions the agent has seen, since the replay buffer never exceeded its maximum capacity);
- finally, we tested the agent in an *ad hoc* scenario, with either known, unknown, or both types of teammates. For each configuration, we ran 1000 trials of 25 episodes each.

The results for these experiments can be found in Section 5.4.2.

5.3.3 Binary + Binary

In this type of experiments, where $I^{\omega} = \{\beta_1, \beta_2\}$, we aimed to determine each binary team's average score rate, to compare their performance with that of POPP. We only considered cases where β_1 and

β_2 were of the same type, since we wanted to determine the performance of the original teams.

We ran each team during 1,000 episodes, and calculated their average score rate over the course of all 1,000 episodes. We also calculated the average score rate for the set of known teams, for the set of unknown teams and for the set of all teams. The results for these experiments can be found in Section 5.4.2, to be compared with those for POPP.

5.4 Results

In this section, we describe the results for the experiments defined in Section 5.3. To represent the standard deviation σ of a sample in the plots, we used 95% confidence intervals ($[\mu - \frac{1.96\sigma}{\sqrt{n}}, \mu + \frac{1.96\sigma}{\sqrt{n}}]$, with n being the sample size, and μ the mean value of the sample).

5.4.1 Learning a Policy in HFO

In this section we present the results for the experiments of the type "Learning + Binary". In each plot, each point corresponds to the average of 3 experiments as defined in Section 5.3.1, one for each of the following teammate types: "aut", "base" and "helios".

5.4.1.A Total Observability

In Figure 5.2, we present the experimental results for the DQN and the DRQN with the --fullstate option of the HFO set to **True**.

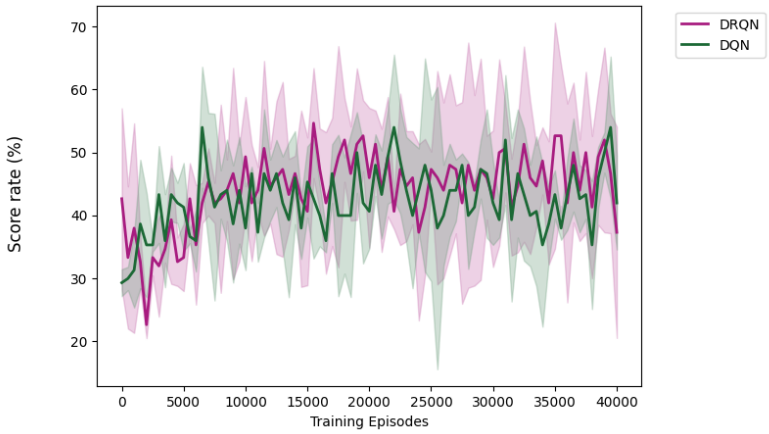


Figure 5.2: Score rate for the DQN and DRQN learning agents playing with total observability. Source: Primary.

5.4.1.B Partial Observability

In Figure 5.3, we present the experimental results for the DQN and the DRQN with the --fullstate option of the HFO set to **False**.

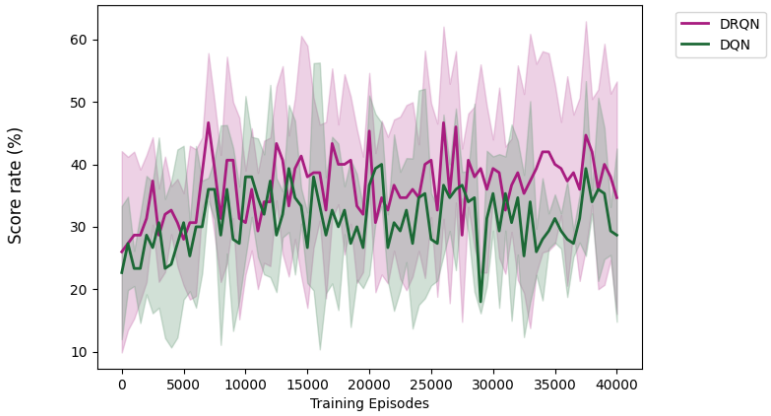


Figure 5.3: Score rate for the DQN and DRQN learning agents playing with partial observability. Source: Primary.

5.4.2 *Ad hoc* teamwork in HFO

In this section, we present the results for our agents playing in an *ad hoc* scenario (experiments of the type "*Ad hoc* + Binary", as explained in Section 5.3.2), and compare them to the binary agents using their original policy (experiments of the type "Binary + Binary" Section 5.3.3). In all experiments, the teams the *ad hoc* agents previously knew were "aut", "base" and "helios".

In Section 5.4.2.A and Section 5.4.1.B, the score rates presented for DQN-PP and POPP are the average of the score rates among 1000 trials. In each trial, the agent's teammate was uniformly chosen from the following types: "aut", "base", "helios", "axiom", "cyrus" and "gliders". The score rate for the binary agents is the average of the score rates of the original teams (teams with two identical agents) for the 6 types the *ad hoc* agent can encounter, where each team was run for 1000 trials.

5.4.2.A Total Observability

In Figure 5.4, we present the experimental results for DQN-PP, POPP and binary agents with the --fullstate option of the HFO set to **True**.

5.4.2.B Partial Observability

In Figure 5.5, we present the experimental results for DQN-PP, POPP and binary agents with the --fullstate option of the HFO set to **False**.

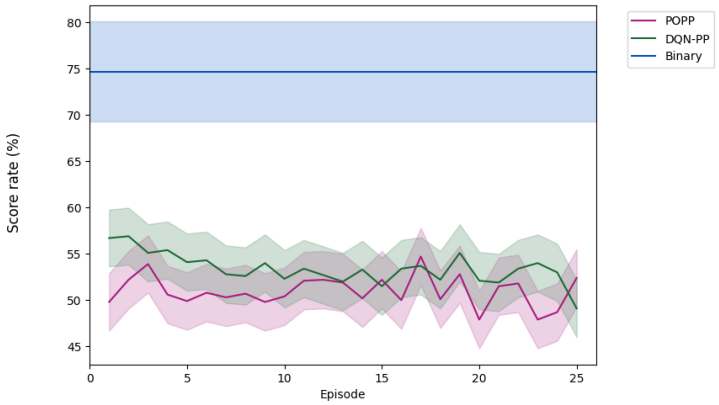


Figure 5.4: Score rate for DQN-PP, POPP and binary agents playing with total observability. Source: Primary.

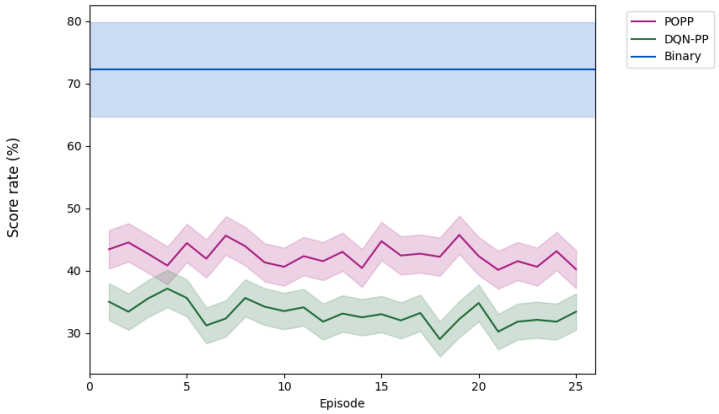


Figure 5.5: Score rate for DQN-PP, POPP and binary agents playing with partial observability. Source: Primary.

5.4.2.C Varying *Ad Hoc* Teammates with Partial Observability

In Figure 5.6, we present and compare the results for POPP while playing with known, and with unknown agents. with the --fullstate option of the HFO set to **False**. Each plot corresponds to the average of 1000 trials. During each trial, the agent paired up with an "aut", "base" or "helios" teammate in the "POPP (known teams)" experiment, and with an "axiom", "cyrus" or "gliders" teammate in the "POPP (unknown teams)" one. In each trial, the teammates were chosen uniformly at random from their sets.

The plots for the "Binary (known teams)" and "Binary (unknown teams)" represent the average over 1000 trials for the score rates of the original teams for the 3 types the *ad hoc* agent encountered in the "POPP (known teams)" and "POPP (unknown teams)" experiments, respectively.

We also present Figure 5.7, which shows how POPP's behavior distribution evolved over the course of 25 episodes, averaged over 1000 trials.

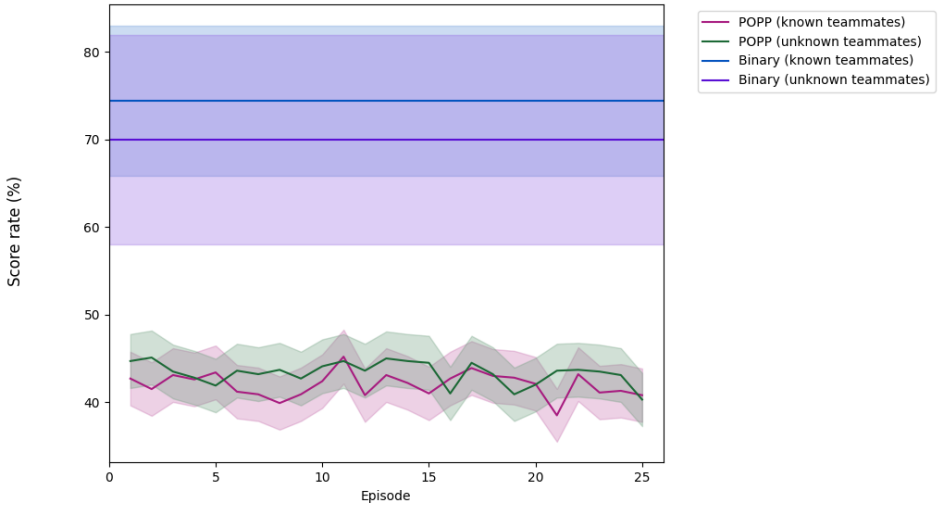


Figure 5.6: Score rate for POPP playing with known and unknown teammates, and for the original binary teams. Source: Primary.

5.5 Discussion

We will discuss our experimental results by answering the questions we posed at the beginning of this chapter.

Question 1. *Can DRQN surpass a non-recurrent (DQN) architecture's performance in a complex, totally observable scenario with sparse rewards?*

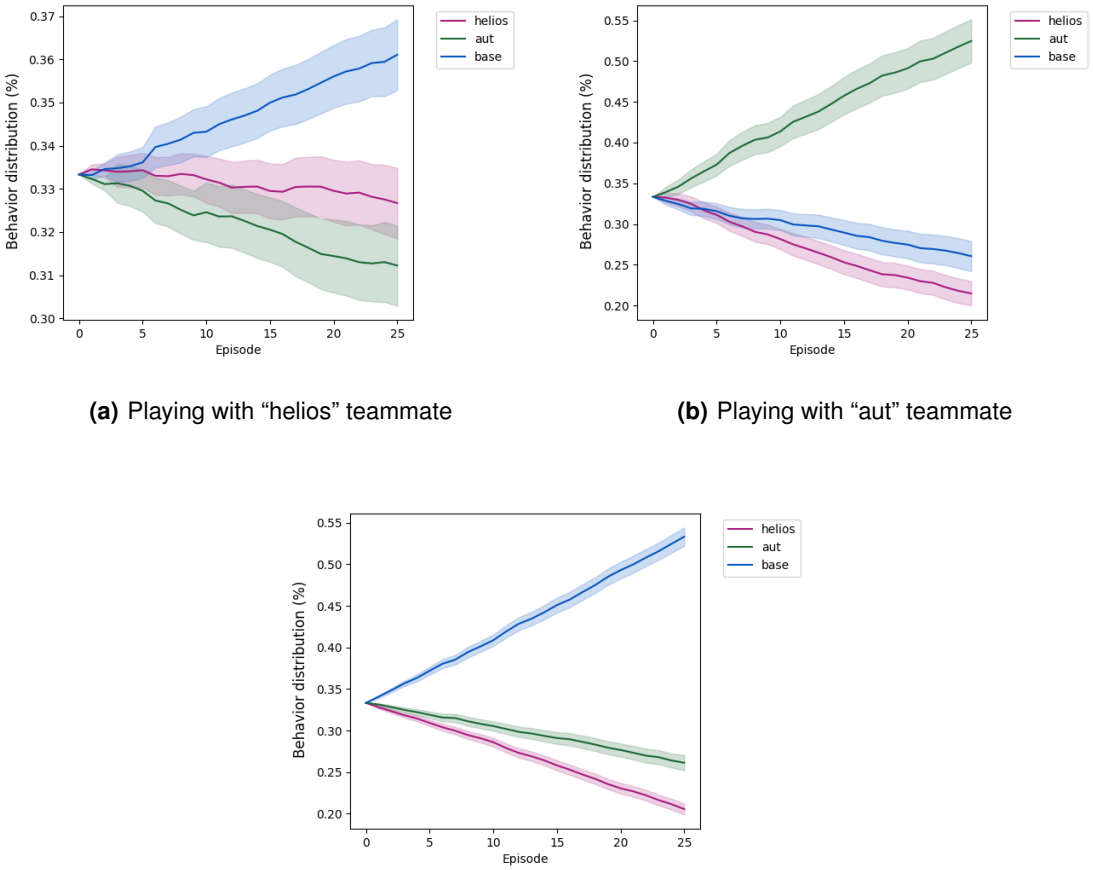
According to the results in Figure 5.2, the DRQN does not have a clear advantage over the DQN with full observability, which was expected, since the agents' observations contain almost all information that could be useful to decide how to act. Still, the DRQN had more frequently performance peaks than the DQN, and we conjecture that might have happened mainly due to two factors: (i) the DRQN being able to extract information about entities' speed, and (ii) the the teammate and opponents' policies being potentially non-stationary.

Question 2. *Can DRQN surpass a non-recurrent (DQN) architecture's performance in a complex, partially observable scenario with sparse rewards?*

According to the results in Figure 5.3, the DRQN surpassed the DQN with partial observability, which was also expected, as, in this case, knowing the history of the last few states constitutes a great advantage for the DRQN.

Question 3. *How accurately can POPP identify teams it previously encountered?*

According to Figure 5.7, there is some variability on how accurately POPP can identify previously encountered teammates. In Figure 5.7(b) and Figure 5.7(c), after 25 episodes, POPP identifies the



(a) Playing with “helios” teammate

(b) Playing with “aut” teammate

(c) Playing with “base” teammate

Figure 5.7: Behavior distribution for POPP when playing with known teammates. Source: Primary.

correct team more than half of the times. However, by observing Figure 5.7(a), we can see that, when playing with team “helios”, the agent has a nearly uniform distribution over the three policies even after the 25 episodes. Moreover, it considers team “base” to be the most likely teammate it is playing with.

We observed the POPP agent playing with the three teams, and realized that (i) the “base” teammate is very unpredictable, when compared with the other two, and (ii) the “base” teammate has a strategy that is more similar to that of the “helios” teammate than to that of the “aut” teammate. Our realization (ii) might explain why “base” has a higher behavior distribution than “àut” when playing with “helios”, but it does not explain why “helios” does not have the highest behavior distribution. However, due to (i), the “base” teammate must have spanned a much wider range of environment transitions during the policy learning process, when compared with the other two teammates. Thus, when cooperating with “helios”, the POPP agent has a higher chance of finding a transition previously encountered when cooperating with “base”, due to its wider range of transitions.

Question 4. *Can POPP surpass a non-recurrent (DQN-PP) architecture in a complex, partially observable scenario with sparse rewards?*

By observing Figure 5.3, we can see that POPP clearly surpassed DQN-PP's performance. Moreover, their 95% confidence intervals do not intersect after episode 4, which strongly endorses our claim.

Question 5. *How does POPP's performance change with the level of similarity between the current and past teammates?*

POPP has shown to be resilient to the lack of knowledge about the teammates it encounters, since, according to Figure 5.6, its performance in the presence of known and unknown teammates is very similar. In fact, its performance in the presence of unknown teammates seems to be slightly higher than in the presence of known teammates, even though the known teammates' original policies had a slightly higher score rate than that of the unknown teammates. This seems to indicate that the POPP successfully transferred its learning from previous tasks to new, unseen teamwork situations.

Question 6. *How does POPP's performance compare with that of the original teams?*

POPP performed poorly when compared with the original teammates' policies. We believe that, in part, this is due to original teammates having complex policies that use more than the 5 actions we allowed the POPP agent to use. Adding to that, some teams use communication protocols (that are available in the HFO simulator), that render useless when used with our agent, since it does not support them.

We also noticed, by observing the agents playing in real-time, that POPP's learnt policies are usually conservative, consisting in passing the ball to the teammate whenever possible. In this way, it is difficult for the team to have effective offense strategies, other than the cases where POPP's teammate is well positioned to score a goal. We believe that this behavior is due to, under partial observability, some High Level actions (especially Shoot()) fail very often. This makes sense, since they implement complex behavior that depends on many features in the observation, which, when unavailable, prevent the agent from executing the action.

With this, we finished answering all 6 questions we posed at the beginning of this chapter.

We will now comment on the performance of POPP and DQN-PP under total observability when compared to an approach with a very similar environment setup as ours - AATEAM, by Chen et al. [6]. Comparing our results to those of to Barrett et al. [7] would be unreliable, since they used a different defense team (they used "helios", whilst we used "base"). Under total observability, POPP achieved a maximum score rate of about 54.2%, and DQN-PP about 56.5%. AATEAM achieved a much higher performance of 76.5%, even surpassing the performance of the "axiom" and "yushan" original teams (i.e., an AATEAM playing with an "axiom"/"yushan" teammate achieves better performance than two "axiom"/"yushan" agents playing together). We believe this discrepancy is mainly due to the following two factors:

- AATEAM features a much more complex, custom-tailored and fine-tuned network than that of PLASTIC-Policy (used by POPP and DQN-PP), which is designed to being reusable for many different environments;
- Chen et al. define a simpler state transition model than ours, considering only transitions where, in the start state, the agent has control over the ball, and, consequently, only actions that are usable in that situation (Dribble(), Pass() and Shoot()). Whenever the agent does not control the ball, it does not observe state transitions, and it always chooses the action Move(), which implements the moving strategy of the high-performance "helios" agent. We also tried this approach when learning the policies for the DQN and DRQN and, in fact, it usually resulted in higher score rates than our current approach. However, that approach would not allow us to test the DRQN (and thus POPP) in a situation that could put it into an advantage over non-recurrent architectures, since the agent would not observe intermediate states that could contain hidden, important information for its decision process.

We will now answer our main research question.

Is it possible to develop an autonomous agent which performs near-optimally in the *ad hoc* teamwork problem, in a complex, partially observable environment with sparse rewards?

Considering the most realistic situation, where the agent can find either known or unknown teammates (Figure 5.5, POPP achieved a performance of about 42% of goals scored, while, in average, the original teams scored goals in 72% of episodes. This is a large gap, so we cannot say that POPP's performance was near optimal. Yet, since we argued that there was still plenty of room for improvement, namely due to the performance gap between POPP and a more custom-tailored architecture (AATEAM), the issue with some actions failing very often, and the difficulty in identifying the correct teammate when one of the known teammates spans a wide range of transitions, we strongly believe it is possible to achieve a near-optimal performance in this scenario.

6

Conclusion

Contents

We have seen the importance of the study and development of *ad hoc* team agents in our current world, and how this field of knowledge has experienced great progress recently, from BOPA to ATPO, and from PLASTIC algorithms to AATEAM. We have also seen how PLASTIC-Policy still thrives as a *state of the art* algorithm in the field, even though it was surpassed in performance by AATEAM, due it's simplicity, adaptability and scalability.

Nevertheless, we found a gap in the literature, by realizing none of the existing approaches tackle a scenario with complex domains and partial observability. To that, we added the sparsity of the rewards due to its plausibility in real-life scenarios.

We presented a novel *ad hoc* teamwork algorithm POPP, designed to deal with these constraints, by combining PLASTIC-Policy with DRQNs. We performed several experiments to assess the learning process of the DRQN, and the performance of POPP in *ad hoc* circumstances.

We tested our algorithm in the HFO domain and concluded that, although it did not achieve a near-optimal performance, POPP successfully generalized its knowledge from working with past teams to its new teammates, even achieving a higher performance in the presence of unknown than of known teammates. Our approach was also able to quickly identify two out of the three previously known teams. Moreover, it surpassed the score rate of its non-recurrent variant, DQN-PP whenever the observability was partial.

6.1 Limitations

As we have noted before, there are still multiple avenues to explore, if we wish to surpass POPP's performance.

Although one of the main goals of *ad hoc* teamwork is adaptability to diverse tasks and situations, the implementation of a more fine-tuned and custom-tailored architecture like that of AATEAM could be a possible way to improve POPP's performance in the HFO domain.

Regarding the issue of policy with very unpredictable or diverse behavior being preferentially followed by POPP, a possible solution could be having the nearest neighbor models compute the similarities between the seen observation and the next observations for the N closest observations to the previous observation in the model, with $N > 1$, ensuring the selected model witnessed multiple similar transitions to the observed one. Alternatively, an (unsupervised) clustering approach could be explored, where the measure to choose a the best transition would take into account both the similarity between the predicted and real observations and a weight for the predicted observation, that was higher if that observation was in a more dense cluster, and lower for outliers.

To deal with the issue of some actions failing very often, an option could be to use more reliable actions (e.g., Mid Level actions), but that would also require some parameter tuning, since, apart from

Intercept(), all Low and Mid Level actions are parameterized.

Also, due to lack of availability of computational resources, we only trained the DQN and DRQN agents for $40,000$ episodes, whilst other authors ([6], [7]) trained their agents for $100,000$ episodes.

6.2 Future Work

We tested POPP for only 25 episodes per trial to more easily compare our results to those of previous approaches which had chosen that value, yet, we noticed that POPP's behavior distribution over the correct team was still increasing after the 25^{th} episode, so it would be interesting to see how the distributions evolve during a longer time span. This, together with testing POPP in other environments, and with a higher number of teammates would contribute to the reliability of our results, since our work considered HFO with 6 specific teams.

Developing architectures that could tackle the *ad hoc* teamwork task with parameterized actions, would also be an interesting path to explore, since it would avoid the need to convert them to discrete actions by fixing their parameters. This would mimic the real-life scenario, where agents might need to choose specific parameters for each specific situation, especially in tasks where a great level of precision is required.

Bibliography

[1] G. Goos, J. Hartmanis, J. van Leeuwen, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. P. Rangan, and B. Steffen, "Lecture Notes in Computer Science," in *16th International Conference Athens, Greece, September 2006 Proceedings, Part I*. Athens, Greece: Springer, Sep. 2006.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, Nov. 2016, google-Books-ID: omivDQAAQBAJ.

[3] S. Barrett, P. Stone, S. Kraus, and A. Rosenveld, "Learning Teammate Models for Ad Hoc Teamwork," in *AAMAS Adaptive Learning Agents (ALA) Workshop*. Citeseer, 2012, pp. 57–63. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.258.6384&rep=rep1&type=pdf

[4] S. James, G. Konidaris, and B. Rosman, "An Analysis of Monte Carlo Tree Search," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*, San Francisco, USA, Feb. 2017.

[5] S. Barrett, *Making Friends on the Fly: Advances in Ad Hoc Teamwork*, ser. Studies in Computational Intelligence. Cham: Springer International Publishing, 2015, vol. 603. [Online]. Available: http://link.springer.com/10.1007/978-3-319-18069-4

[6] S. Chen, E. Andrejczuk, Z. Cao, and J. Zhang, "AATEAM: Achieving the Ad Hoc Teamwork by Employing the Attention Mechanism," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, pp. 7095–7102, Apr. 2020, number: 05. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/6196

[7] S. Barrett and P. Stone, "Cooperating with Unknown Teammates in Complex Domains: A Robot Soccer Case Study of Ad Hoc Teamwork," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, Feb. 2015. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/9428

[8] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://www.nature.com/articles/nature14236

[10] M. Hausknecht and P. Stone, “Deep Recurrent Q-Learning for Partially Observable MDPs,” in *2015 AAAI Fall Symposium Series*, Sep. 2015. [Online]. Available: https://www.aaai.org/ocs/index.php/FSS/FSS15/paper/view/11673

[11] P. Stone, G. Kaminka, S. Kraus, and J. Rosenschein, “Ad Hoc Autonomous Agent Teams: Collaboration without Pre-Coordination,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, pp. 1504–1509, Jul. 2010. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/7529

[12] S. Barrett, P. Stone, and S. Kraus, “Empirical Evaluation of Ad Hoc Teamwork in the Pursuit Domain,” in *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Taipei, Taiwan, May 2011, pp. 567–574.

[13] J. G. Ribeiro, M. Faria, A. Sardinha, and F. S. Melo, “Helping People on the Fly: Ad Hoc Teamwork for Human-Robot Teams,” in *Progress in Artificial Intelligence*, G. Marreiros, F. S. Melo, N. Lau, H. Lopes Cardoso, and L. P. Reis, Eds. Cham: Springer International Publishing, 2021, vol. 12981, pp. 635–647, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-86230-5_50

[14] C. Martinho, “Ad Hoc Teamwork under Partial Observability,” Master’s thesis, Universidade de Lisboa, Nov. 2021.

[15] M. Hausknecht, P. Mupparaju, S. Subramanian, S. Kalyanakrishnan, and P. Stone, “Half field offense: An environment for multiagent learning and ad hoc teamwork,” in *AAMAS Adaptive Learning Agents (ALA) Workshop*, vol. 3. sn, 2016.

Half-Field Offense Details

A.1 High Level State Feature Set

Table A.1: High Level State Feature Set for Half-Field Offense; T represents the number of teammates ($T = |I^{\omega}| - 1$) and O the number of opponents ($O = |I^{\delta}|$)

Feature	Count	Range	Description
X Position	1	[-1, 1]	Agent's x position on the field, with -1 being the leftmost position, and 1 the rightmost one.
Y Position	1	[-1, 1]	Agent's y position on the field, with -1 being the topmost position, and 1 the bottommost one.
Orientation	1	[-1, 1]	An angular value that represents the agent's orientation on the field.
Ball X	1	[-1, 1]	The ball's x position on the field.
Ball Y	1	[-1, 1]	The ball's y position on the field.
Able to Kick	1	{-1, 1}	Whether or not the agent is able to kick the ball.
Goal Center Proximity	1	[-1, 1]	Distance between the agent and the center of the goal.
Goal Center Angle	1	[-1, 1]	Distance between the agent and the center of the goal.
Goal Opening Angle	1	[-1, 1]	Amplitude of the largest open angle the agent has to shoot the ball towards the goal, taking into account other entities' positions.
Proximity to Opponent	1	[-1, 1]	Distance between the agent and the closest opponent.
Teammate i's Goal Opening Angle	T	[-1, 1]	Amplitude of the largest open angle teammate i has to shoot the ball towards the goal, taking into account other entities' positions.
Proximity of Teammate i To Opponent	T	[-1, 1]	Distance between teammate i and the closest opponent.
Teammate i's Pass Opening Angle	T	[-1, 1]	Amplitude of the largest open angle the agent has to pass the ball to teammate i, taking into account other entities' positions.
Teammate i's X Position	T	[-1, 1]	Teammate i's x position in the field.
Teammate i's Y Position	T	[-1, 1]	Teammate i's y position in the field.
Teammate i's Uniform Number	T	{1, ..., 11}	Number in teammate i's soccer uniform. Every teammate has a different uniform number.
Opponent i's X Position	O	[-1, 1]	Opponent i's x position in the field.
Opponent i's Y Position	O	[-1, 1]	Opponent i's y position in the field.
Opponent i's Uniform Number	O	{1, ..., 11}	Number in opponent i's soccer uniform. Every opponent has a different uniform number.
Last Action's Success Possible	1	{-1, 1}	Whether or not it is possible that the last action the agent executed was successful.