Using deep model-based RL in ad hoc teamwork with partial observability

Eduardo Filipe Sousa Paiva

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

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

June 2023

Acknowledgments

I would like thank my supervisors Prof. Francisco Melo and Prof. Alberto Sardinha for their guidance, without them this work would not have been possible

I also thank João Ribeiro for providing the well-structured code that I used to start off my work and for helping me by answering my initial questions that I had about the challenge of this dissertation.

I would also like to like to thank my parents and my sister for their love and support throughout this journey. I would also like to thank my cats and dog for keeping me company.

Last, but certainly not least, I would like to thank my friends and colleagues that I met in these past years.

To each and every one of you – Thank you.

Abstract

This thesis contributes a novel model-based reinforcement learning approach to ad hoc teamwork. Ad hoc teamwork has been the focus of intense research in the multi-agent systems community and consists of the problem of teaming up an agent with a group of unknown agents in a cooperative task. The agent must use its prior knowledge to quickly figure out how to coordinate with the other agents. Most work in ad hoc teamwork (e.g., the PLASTIC architecture) focuses on settings with full observability, where the agent is able to perfectly observe the state of the environment and the actions of the teammates. We propose a novel ad hoc teamwork approach, which we name PLASTIC-Dyna. Our approach combines the PLASTIC architecture with an extended version of Dyna-Q that includes a recurrent neural network. PLASTIC-Dyna is thus able to scale to complex environments with partial observability. At the same time, PLASTIC-Dyna enables the ad hoc agent to generate simulated experiences that can be used for learning, significantly improving the data efficiency of our ad hoc agent. We test our approach in well-known benchmarks from the ad hoc teamwork literature and show that PLASTIC-Dyna significantly outperforms competing methods, both in terms of sample efficiency and in terms of ad hoc teamwork.

Keywords

Multi-Agent Systems; Multi-Agent Planning; Reinforcement Learning; Ad Hoc Teamwork; World Model.

Resumo

Esta tese contribui com uma nova abordagem model-based reinforcement learning para ad hoc teamwork. Ad hoc teamwork tem sido foco de intenso estudo na comunidade de sistemas multi-agente e consiste no desafio de juntar um agent com um grupo de agentes desconhecidos numa tarefa cooperativa. O agente deve usar seu conhecimento prévio para descobrir rapidamente como se coordenar com os outros agentes. A maioria dos trabalhos em ad hoc teamwork (por exemplo, a arquitetura PLASTIC) concentra-se em configurações com observabilidade total, nas quais o agent é capaz de observear perfeitament o estado do ambiente e as ações dos teammates. Nós propomos uma nova abordagem de ad hoc teamwork, que chamamos de PLASTIC-Dyna. A nossa abordagem combina a arquitetura PLASTIC com uma versão estendida do Dyna-Q que inclui uma rede neuronal recorrente. PLASTIC-Dyna é assim capaz de escalar para ambientes parcialmente observáveis mais complexos. Ao mesmo tempo, PLASTIC-Dyna possibilita que o agente ad hoc gere experiências simuladas que podem ser usadas para aprender, melhorando significativamente a sua eficiência de dados. Nós testamos a nossa abordagem em benchmark conhecidas da literatura em ad hoc teamwork e mostramos que o PLASTIC-Dyna supera significativamente os métodos concorrentes, tanto em termos de eficiência de dados como em termos de ad hoc teamwork.

Palavras Chave

Sistemas Multi-Agente; Planeamento Multi-Agente; Aprendizagem por Reforço; Trabalho em Equipa Ad Hoc; Modelo do Mundo.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

Adam	Adaptive moment estimation
AI	Artificial Intelligence
ANN	Artificial Neural Networks
ATPO	Ad hoc Teamwork under Partial Observability
ATSIS	AdHoc Teamwork by Sub-task Inference and Selection
DL	Deep Learning
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
DRQN	Deep Recurrent Q-Network
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
MARL	Multi-Agent Reinforcement Learning
MAS	Multi-Agent Systems
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
ML	Machine Learning
MMDP	Multi-agent Markov Decision Process
MSE	Mean Squared Error
NLL	Negative Log-Likelihood
PODQN	Partial Observable Deep Q-Network
POMDP	Partial Observable Markov Decision Process
ReLU	Rectified Linear Unit
RL	Reinforcement Learning

RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SE	Standard Error
SimPLe	Simulated Policy Learning

1 Introduction

Contents

For the past few years, the field of Artificial Intelligence (AI) has been growing at a very fast rate [1]. It has also shown remarkable contributions in several other fields, such as robotics, medicine, and outer-world exploration. One thing that all these fields have in common, is that they all present highly complex tasks that are almost impossible for a single autonomous agent to solve [2]. Multi-Agent Systems (MAS) incorporate cooperation between two or more agents and proved to be very useful when dealing with tasks that require decentralized reasoning, distributed actuation, and robustness to component failure [3]

Recently, it has been witnessed a significant volume of research in the problem of ad hoc teamwork, first introduced by Stone et al. [4]. Ad hoc teamwork consists of the problem of teaming an agent (henceforth referred as the "ad hoc agent") with a group of unknown agents (henceforth referred as "the teammates") in a cooperative task. The ad hoc agent must take advantage of prior knowledge (e.g., coming from previous interactions with other agents) to quickly coordinate with the teammates in the target task.

Most work in ad hoc teamwork considers settings in which the ad hoc agent has full observability [5–8]. In other words, at each time step, the ad hoc agent is able to observe the actions of the teammates and the complete state of the environment. However, such an assumption is often unrealistic. In most real-world scenarios, the ad hoc agent will only be able to perceive the actions of the teammates indirectly, through their effects on the environment; on the other hand, most agents can only observe the state of the environment through imperfect sensors, making full state observability an assumption hard to meet in practice.

Ribeiro et al. [9] proposed Ad hoc Teamwork under Partial Observability (ATPO), a first approach to ad hoc teamwork with partial observability that makes use of decision-theoretic planning—namely, the policy of the ad hoc agent is computed using Partial Observable Markov Decision Process (POMDP) solution methods. Despite the good results reported in the aforementioned work, the use of a POMDP-based solution implies that ATPO scales poorly with the dimension of the problem.

It also assumes perfect knowledge of the underlying tasks/teams that the ad hoc agent is expected to encounter. In practice, such knowledge will often be imperfect and the result of some prior learning process, in which partial observability may also prove a challenge.

In contrast with ATPO, the PLASTIC-policy architecture, proposed by Barrett [5], assumes full observability of both the environment state and the teammate actions, but builds on learned parameterized policies. Specifically, the agent's prior knowledge regarding underlying tasks/teams is represented in the form of neural networks trained using Q-learning during the agent's prior interactions with those teams. In that sense, the approach already uses learning to acquire prior knowledge and is able to scale better to larger problems.

In this thesis, we propose an extension of the PLASTIC architecture, alleviating the assumption of full observability of state and action. Our approach, which we name PLASTIC-Dyna, replaces the Q-learning algorithm used in PLASTIC-policy with an extension of Dyna-Q [10] that uses a recurrent neural network to represent the Q-function to be learned. The use of a recurrent network enables the agent to handle partial observability by introducing a dependence on history. The use of Dyna-Q, on the other hand, enables the agent to make better use of the learning data, rendering the overall approach more sample efficient.

The world model learned by Dyna-Q also provides PLASTIC-Dyna with the necessary information to identify the teammate. In the original PLASTIC approach, such identification was performed by relying on the observation of the teammates' actions which, in our partially observable setting, is no longer possible.

We trained and tested PLASTIC-Dyna on the classical pursuit domain. Our results show that PLASTIC-Dyna clearly outperforms PLASTIC policy, both in terms of its ability to handle partial observability and in terms of sample efficiency.

1.1 Research Problem

In this thesis we undertake and expand on the challenge proposed by Stone et al. [4], answering the question, **is it possible to create a sample efficient reinforcement learning approach that is able to correctly identify the target task under partial observability?**

We break down the previous question into three different parts:

1. How to achieve good performances in a partial observable environment?
2. How to attain sample efficiency?
3. How to correctly identify the target task in an ad hoc teamwork setting?

1.2 Contributions

In this thesis we introduce a novel approach, the ad hoc deep model-based reinforcement learning approach denominated PLASTIC-Dyna, that answers the question presented in section 1.1.

Our contribution consists of two big components. The first one, DynaDRQN, answers the first two sub-questions of our research problem. The usage of a deep recurrent layer allows our agent to keep track of the history of previous observations and therefore solve the issue created by partial observability [11]. Whilst the Dyna sub-component, which utilizes a preconceived world model to generate simulated experiences for the agent, combats the sample scarcity [10]. The second component deals with the challenge of ad hoc teamwork [4], it employs a variation of the PLASTIC method introduced by Barret et al. [5], which identifies the target task based on the behaviour of past teammates. In our case, it utilizes the preconceived world model to predict the next observation of the teammates' location for each possible task given the current observation and action taken, which are then compared with the real next observation. This PLASTIC method presents the answer to the third question, by granting our agent the ability to identify the target task in an ad hoc teamwork setting.

1.3 Organization of the Document

This thesis is is organized as follows: Chapter 1 presents this thesis research problem and our contribution to the field of MAS. In chapter 2 we offer a detailed explanation of all the technical concepts employed or discussed in this work. In chapter 3 we write about the literature and state of the art in the field of MAS and Machine Learning (ML). Chapter 4 introduces our novelty contribution, the PLASTIC-Dyna, as well as a comprehensive overview of all its components. In chapter 5 we analyze and discuss the results obtained during our work. By comparing the performance of our contribution with some well-known baselines in the benchmark pursuit environment. Chapter 6 presents a summary of all the themes discussed in this thesis and endorses the value of our contribution. In it we also discuss some work limitations and possible future work

2 Background

Contents

In this chapter we present a detailed analysis of the main concepts implemented in this thesis. Starting with a report on the different types of machine learning. Then a more detailed summary of model-based and model-free reinforcement learning. Followed by the concept and benefits of deep reinforcement learning and concluding with an overview of the ad hoc teamwork PLASTIC architecture.

2.1 Machine Learning Overview

ML is a subset of AI and its defined as ”the field of study that gives computers the ability to learn without being explicitly programmed” according to Domingos [12]. The field of ML can be subdivided into unsupervised learning, supervised learning and Reinforcement Learning (RL), just as it is displayed in fig. 2.1.

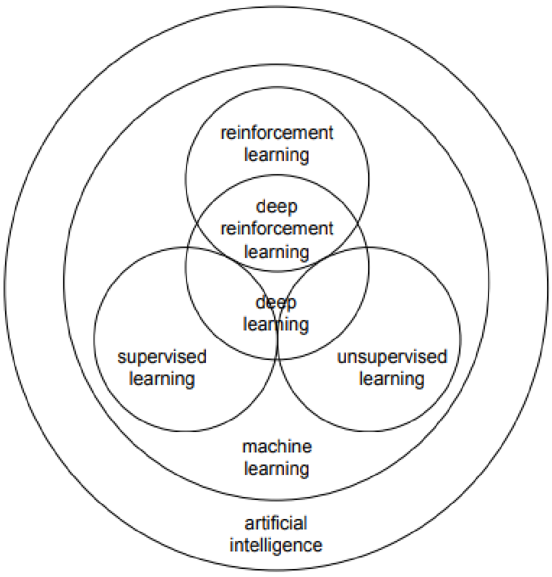


Figure 2.1: Arificial Intelligence sub-fields

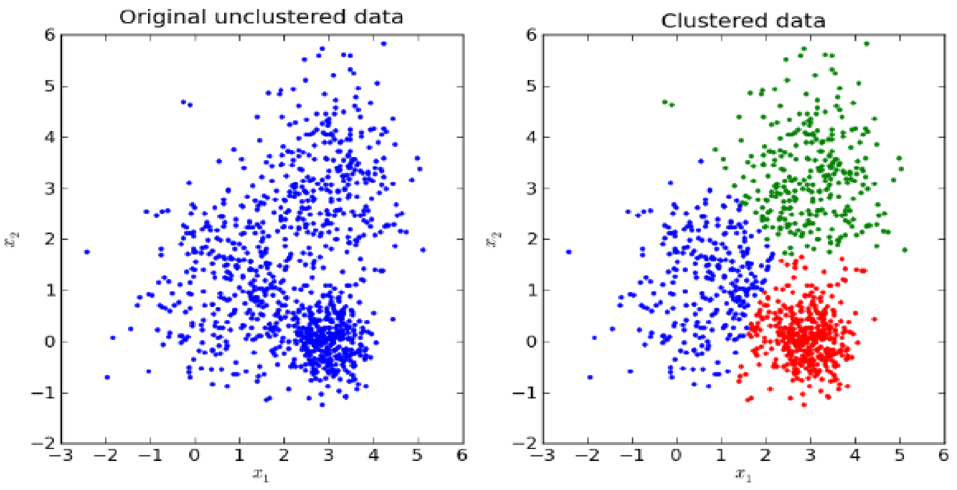


Figure 2.2: Application of the k-means clustering technique in unlabeled data

2.1.1 Unsupervised Learning

Unsupervised learning is a type of machine learning that does not require supervision nor labeled data. These type of algorithms aim to learn patterns and associations between the data, to make predictions and classification of new unseen data [13]. A famous unsupervised technique is clustering, which aims to group the data according to the similarities and patterns between the unlabeled data samples. In fig. 2.2 it is represented the grouping of the data samples after the application of the k-means clustering for k equal to 3, which aims to minimize the distance of the samples within the clusters (intra-cluster's distance) and maximize the distance between clusters (inter-cluster's distance).

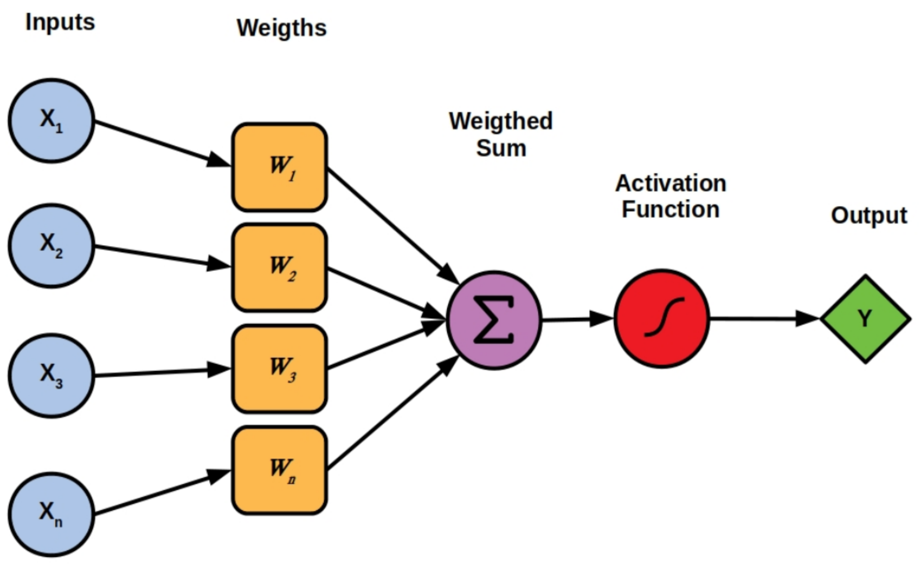


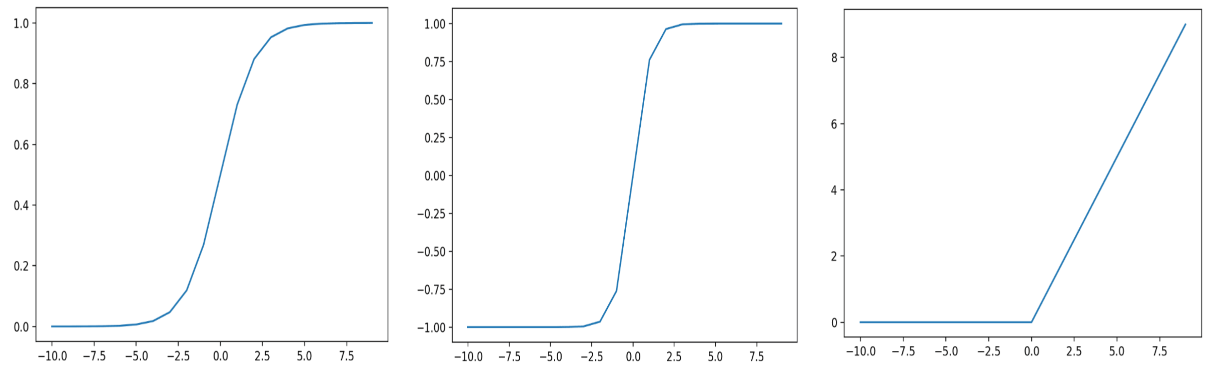
Figure 2.3: Perceptron's Architecture

2.1.2 Supervised Learning

Supervised learning is a type of machine learning that utilizes labeled data to learn a mapping function between the inputs and the targets (labeled data) and later apply it to make accurate predictions or classifications in unseen data. The most well-known type of supervised algorithms are the Artificial Neural Networks (ANN).

Neural networks are a type of algorithm that is inspired by the structure and function of the human brain [14]. The most basic component of ANN is the perceptron (fig. 2.3), introduced by Rosenblatt in 1957 [15]. In the displayed perceptron's figure we have an input layer where each node has a certain weight attributed, which will be used to determine the importance of each input during the forward pass. The weighted sum result then goes through an activation function, which is applied to insert non-linearity to the network, allowing it to model more complex correlations and make more accurate predictions. The type of activation function to apply may vary depending on the situation.

The sigmoid function (fig. 2.4 (a)), which is defined as (2.1), outputs values between 0 and 1 and can be used when the target output is binary. The sigmoid function even if intuitive has a hard to compute gradient and a *vanishing gradient* problem, which happens during the backpropagation step, when the gradient gets too small and has almost no impact on the weights' updates. The hyperbolic tangent function (tanh) (fig. 2.4 (b)) is defined as (2.2) and outputs values between -1 and 1. Even though tanh solves the *vanishing gradient* problem, it has an hard to compute gradient and a saturation region, similar to the sigmoid function. The Rectified Linear Unit (ReLU) (fig. 2.4 (c)), on the other hand, not only has an easier to compute gradient, but also solves the *vanishing gradient* problem [16]. It is defined as



(a) Sigmoid Function **(b)** TanH Function **(c)** ReLU Function

Figure 2.4: Activation Functions

(2.3) and despite solving the previous problems, it suffer from the ”dead region” one, which means that negative values will lead to a gradient of 0 which will never update the respective weights. Finally, there is the softmax activation function, which is defined as (2.4) and is utilized to output probabilities that sum to 1, especially helpful when dealing with multi-class targets.

The neural networks employed to solve modern problems consist of multiple layers of perceptrons and can handle more complex and nonlinear problems.

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.1}$$

$$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.2}$$

$$f(x) = max(0,x) \tag{2.3}$$

$$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{2.4}$$

2.1.3 Reinforcement Learning

RL is a type of machine learning that focuses on trial-and-error. An agent interacts with environment through discrete time steps and learns to make sequential decisions in it to maximize a cumulative reward. At each time step, with full visibility the agent observes the current state of the environment, takes an action, and receives feedback in the form of a reward and a new state (fig. 2.5). The agent’s goal is to learn the optimal $policy$, which represents the action to take in a certain state that maximizes

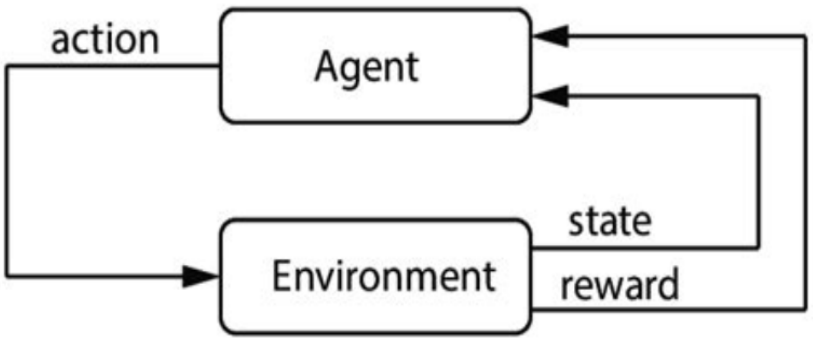


Figure 2.5: Flowchart of an agent interacting with a fully visible environment

the cumulative reward over time.

The interaction of an agent with the environment can be described as a Markov Decision Process (MDP) [17], which can be represented as a tuple $M = (X, A, P, r, \gamma)$. X and A represent the set of spaces and actions respectively, P is the transition probabilities given by (2.5), r is the reward function defined as (2.6) and γ is the discount factor, that varies between 0 and 1 and determines the importance given to future rewards.

$$P(y|x,a) \triangleq \mathbb{P}[X_{t+1} = y | X_t = x, A_t = a] \tag{2.5}$$

$$r(x,a) \triangleq \mathbb{E}[R_{t+1} | X_t = x, A_t = a] \tag{2.6}$$

In RL the agent chooses the actions to take according to its policy (π). To evaluate an agent's policy we can use the value function $v^{\pi}(x)$ (2.7), which measures the expected discounted accumulated reward assuming the agent is in state x and follows its policy π.

$$v^{\pi}(x) = \mathbb{E}_{\pi}\left[\sum_{k=1}^{\infty} \gamma^k R_{t+k+1} | X_t = x\right] \tag{2.7}$$

It is also important to derive the respective action-value function ($q^{\pi}(x,a)$) (2.8), which measures the expected discounted accumulated reward assuming the agent is in state x and executes action a and follows its policy π.

$$q^{\pi}(x,a) = \mathbb{E}_{\pi}\left[\sum_{k=1}^{\infty} \gamma^k R_{t+k+1} | X_t = x, A_t = a\right] \tag{2.8}$$

The optimal policy (π^*) can be derived from the optimal action-value function with (2.9)

$$\pi^*(x) = \underset{a \in A}{\arg\max}(q^*(x,a)), \quad \forall \quad x \in X \tag{2.9}$$

2.1.3.A Model-Free Reinforcement Learning

Model-free reinforcement learning is a type of RL where an agent learns to make decisions and optimize its behavior directly from interactions with the environment, without having access to a model. [18].

The most notable model-free RL algorithm is the Q-learning [19]. Q-learning is an off-policy value-based learning algorithm that estimates the action-value function (Q-function) for state-action pairs and uses it to determine the policy that will maximize the accumulated discounted reward. This approach is considered value-based because it aims to estimate the optimal action-value function and then derive the optimal policy from it (2.9). It is also considered an off-policy algorithm, because it uses a different policy to update the Q-values different from the one that it follows to interact with the environment, as it can be seen by the usage of $\max_a$ in (2.10).

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)] \tag{2.10}$$

Through the interaction with the environment the agent gathers samples (S, A, R, S') that it uses to update the $action - value$ function (2.10). α represents the learning rate and γ represents the discount factor.

2.1.3.B Model-Based Reinforcement Learning

Model-based reinforcement learning is a type of RL where an agent learns a model of the environment and uses it to make decisions and optimize its behavior. Unlike model-free approaches, model-based RL agents , employ the learned model to simulate possible outcomes and plan its actions accordingly, without requiring to interact directly with the environment.

Dyna-Q is an example of a model-based reinforcement learning algorithm that utilizes a model to perform planning and accelerate the learning process. By using both real and simulated experiences, Dyna-Q can learn and improve its policy more efficiently compared to pure model-free approaches, such as the Q-learning. The planning step allows the agent to explore potential state-action pairs and update their Q-values, even without directly experiencing them. The algorithm of the Dyna-Q introduced by Sutton [10] is displayed in algorithm 2.1.

2.2 Deep Q-Network

The combination of Deep Learning (DL) and RL led to what is now called Deep Reinforcement Learning (DRL) (fig. 2.1). One very well-known example is the deep Q-network (DQN), which combines Q-learning with deep neural networks [20]. Q-learning is a model-free RL algorithm that aims to learn the optimal policy for a given environment, by storing the cumulative discounted rewards (Q-values) for

Algorithm 2.1: Dyna-Q Algorithm

```
Initialize $Q(x, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A$
Do forever:
  (a) $S \leftarrow$ current (non-terminal) state
  (b) $A \leftarrow \epsilon - greedy(S, Q)$
  (c) Execute action $A$; Observe resultant reward, $R$, and state, $S'$
  (d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
  (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
  (f) Repeat $n$ times:
    $S \leftarrow$ random previously observed state
    $A \leftarrow$ random action previously taken in $S$
    $R, S' \leftarrow Model(S, A)$
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
```

every state-action pair. The original Q-learning stores the Q-values in a Q-table, yet as we have previously mentioned, this approach may not work for more complex environments that have a high number of possible states or actions. With the inclusion of deep neural networks, and function approximators, it becomes possible to represent the Q-values with a Q-function. The Q-function can be described as $Q(s, a; \theta)$, where s represents the state, a the action and θ represents the weights of the neural network.

Nevertheless, while these approximators can be powerful and flexible, due to the nature of the Q-learning algorithm, they do not guarantee convergence [21]. To solve this problem, several techniques have been developed to improve the stability and convergence of Q-learning with non-linear approximators. One approach is to use experience replay, which stores past experiences in a buffer and samples batches of experiences at random from it, in order to remove correlations in the data and to smooth changes in the data distribution [20]. The other one is the use of a target network, which can be employed to compute the target Q-values, while the agent follows the main network's policy. This helps to reduce the correlation between successive updates and improve the stability of the learning process. Without a target network, the main network would be ”chasing” an ever-moving target. The target network weights are only updated in a defined interval.

The Deep Q-Network (DQN) takes the state of the environment as input and outputs a Q-value for each possible action. The weights of the neural network are updated through back-propagation of the gradients through the use of an optimizer, with Stochastic Gradient Descent (SGD) [22] being the most well-known one. The update of weights is given by

$$\theta \leftarrow \theta - \alpha \nabla_\theta \text{Loss}(y, Q(s, a; \theta)), \tag{2.11}$$

where α represents the learning rate, $Q(s, a, \theta)$ is the output of the network for state s and action a and the y is the output of the target network, which is calculated as

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-). \tag{2.12}$$

In (2.12), γ represents the discount factor used in Q-learning, which defines the importance given to future rewards, and r denotes the reward obtained from the execution of a certain action; s' is the state following s, and θ^- are the weights of the target network. The loss in (2.11) is given by

$$\mathrm{Loss}(y, Q(s, a, \theta)) = (y - Q(s, a; \theta))^2.$$

2.3 Deep Recurrent Q-Network

Recurrent Neural Network (RNN) are neural networks that are often used in the processing of sequential data. The way that these work is that the output of the hidden layer of each input will be used along with the next input of the sequence to calculate the respective output values. This allows the network to maintain a memory of the previous inputs and to use it for its predictions.

The most common type of RNN architecture is the Long Short-Term Memory (LSTM) network [23]. It uses a set of gates to control the flow of information through the network, including an input gate, an output gate, and a forget gate that allow the network to selectively keep or discard information from the previous state. The ability to control the flow of information through the network helps to stabilize the gradients, hence also preventing vanishing gradients.

A more detailed overview of each one of the LSTM's gates is presented in fig. 2.6 (Blue activation function is a tanh function. Red activation function is a sigmoid function. ”X” represents point-wise multiplication and cross represents point-wise addition). It receives as inputs the current sequence input (x_i), the previous hidden state (history) (h_{t-i}) and the previous cell state (memory) (c_{t-i}). The first gate is the forget gate, which takes the output of a sigmoid function that receives the current sequence input and the previous hidden state to make the cell state forget certain information. The sigmoid layer is used since it outputs values between 0 and 1 which are later multiplied with the cell state, with 0 representing information to forget. The other gate is the input gate, which takes the current sequence input and the previous hidden state used to update the cell state. The information goes through a tanh activation function and a sigmoid function, which is utilized to determine the information that should be utilized to update the cell state. Finally, the output gate consists of a sigmoid layer that determines the information of the cell state that should be used ,in conjuction with the sequence input and the previous hidden state, to output the current hidden state (h_t).

One other more recent well-known RNN architecture is the Gated Recurrent Unit (GRU) [24]. GRU has two gating mechanisms, the reset gate and the update gate, as it can be seen in fig. 2.7. This figure

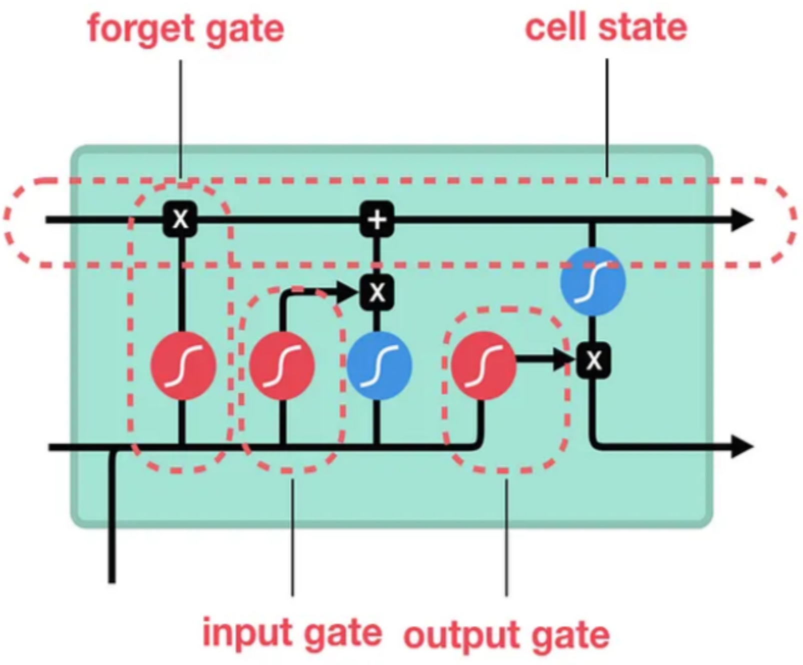


Figure 2.6: Architecture of a LSTM

follows the same nomenclature as fig. 2.6. GRU, differently from the LSTM, does not keep a memory of past observation (cell state). It instead applies the two gates in combination with the current sequence input (x_t) to determine which information from the past hidden state (h_{t-1}) should be output (h_t). The reset gate determines the amount of information from the previous hidden state that is passed onto the current hidden state, by applying a sigmoid activation function to the merge of sequence input and previous hidden state. This information then goes through a tanh function that outputs what is called as the candidate hidden state. The only other gate is the update gate, which controls how much information from the previous hidden state should be combined with the candidate hidden state. The last point-wise addition calculates how much of the previous hidden state and candidate state is kept for the output hidden state. Compared to LSTM, GRU has fewer parameters and is computationally less expensive, making it faster to train and less prone to overfitting [25].

Deep Recurrent Q-Network (DRQN) is the combination of DQN with the sequence processing capabilities of RNN, and has been proposed as an extension of DQN to handle partial observability [11].

2.4 PLASTIC architecture

The PLASTIC algorithm uses information gathered from prior interactions with a set of teammates, in order to identify the current set of teammates and task (algorithm 2.2).

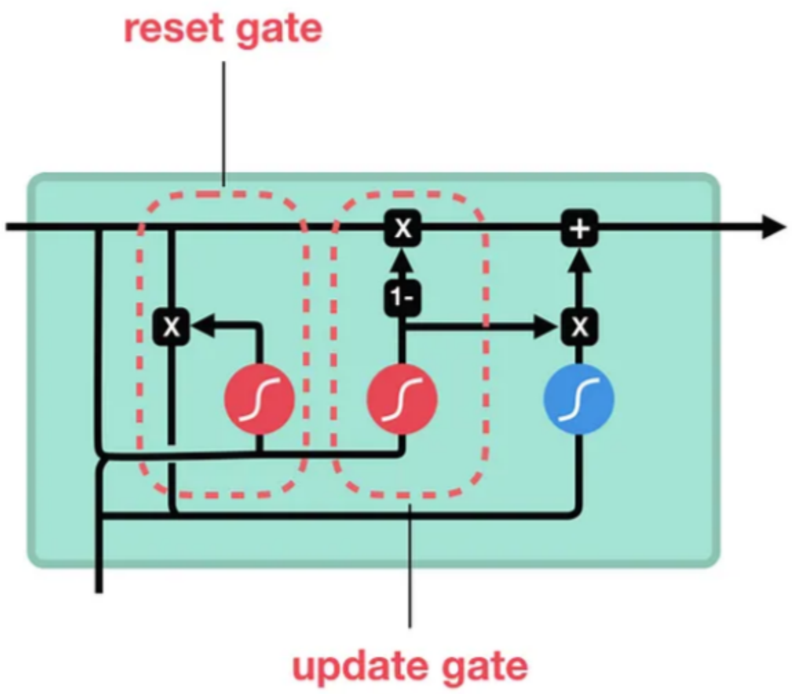


Figure 2.7: Architecture of a GRU

Algorithm 2.2: PLASTIC Algorithm

```
(1) Function PLASTIC:
        inputs:
            PriorTeammates
            HandCodedKnowledge
            BehaviourPrior

Initialize knowledge using information from prior teammates

(2)     PriorKnowledge = HandCodedKnowledge:
(3)     for $t \in$ PriorTeammates do
(4)         PriorKnowledge = PriorKnowledge $\cup$ {LearnAboutPriorTeammate($t$)}
(5)     BehaviourDistr = BehaviourPrior(PriorKnowledge)

Act in domain

(6)     Initialize $s$
(7)     while $s$ is not terminal do
(8)         $a$ = SelectAction(BehaviourDistr, $s$)
(9)         take action $a$ and observe $r, s'$
(10)        BehaviourDistr = UpdateBeliefs(BehaviourDistr, $s$, $a$)
```

The PLASTIC-Policy (algorithm 2.3) is a version of the PLASTIC algorithm that uses the policies learnt by the agent for each task and set of teammates as prior knowledge. It is also displayed as a figure in fig. 2.8. It starts by loading the prior knowledge and initializing the set of beliefs for each possible task. The prior knowledge (function **LearnAboutPriorTeammate**), incorporates the policies learnt beforehand for each possible task, which determines the best course of actions for the agent to work with teammates t. It also learns the model m of those teammates, to use later on.

After the setup, the agent starts interacting with the environment (act in domain), making use of the **SelectAction** function. It chooses which policy to follow according to the task (teammates' behaviour) with the highest belief value. The ad hoc agent utilizes this policy to select an action based on the state that it is in. After each interaction the agent updates the set of believes by applying the function **UpdateBeliefs**. This function makes use of the state and teammates' actions to update the current task's believes. For each policy/model in the set of possible tasks, the loss can be calculated as

$$loss = 1 - P(a|m, s), \tag{2.13}$$

where $P(a|m, s)$, represents the probability of the teammates in a certain task (policy/model m) and state s taking action a. The loss is then used to update the belief for each task as

$$\text{Belief}(m) \leftarrow \text{Belief}(m)(1 - \eta\ loss), \tag{2.14}$$

where η determines how meaningful each update will be, with values close to 0 representing softer updates. $\text{Belief}(m)$ represents the agent's belief of being in a task where the teammates' behaviour is given by m.

Algorithm 2.3: PLASTIC-Policy Functions

```
(1) Function LearnAboutPriorTeammate:
        inputs:
          $t$
        outputs:
          $\pi$
          $m$
        params:
          $Q$-learning parameters: $a$, $\gamma$ and the function approximation
(2)    Data = Ø
(3)    repeat:
(4)        Collect $s, a, r, s'$ for $t$
(5)        Data = Data ∪ $\{(s, a, r, s')\}$
(6)    Learn a policy $\pi$ for Data using $Q$-learning
(7)    Learn a nearest neighbors model $m$ of $t$ using Data
(8)    return $(\pi, m)$

(9) Function UpdateBeliefs:
        inputs:
          BehaviousDistr
          $s$
          $a$
        outputs:
          BehaviousDistr
        params:
          $\eta$
(10)    for $(\pi, m) \in$ BehaviourDistr do:
(11)        loss = 1 - $P(a|m, s)$
(12)        BehaviourDistr($m$) *= (1 - $\eta$loss)
(13)    Normalize BehaviourDistr
(14)    return BehaviourDistr

(15) Function SelectAction:
        inputs:
          BehaviousDistr
          $s$
        outputs:
          $a$
(16)    $(\pi, m)$ = argmax(BehaviourDistr)
(17)    $a$ = $\pi(s)$
(18)    return a
```

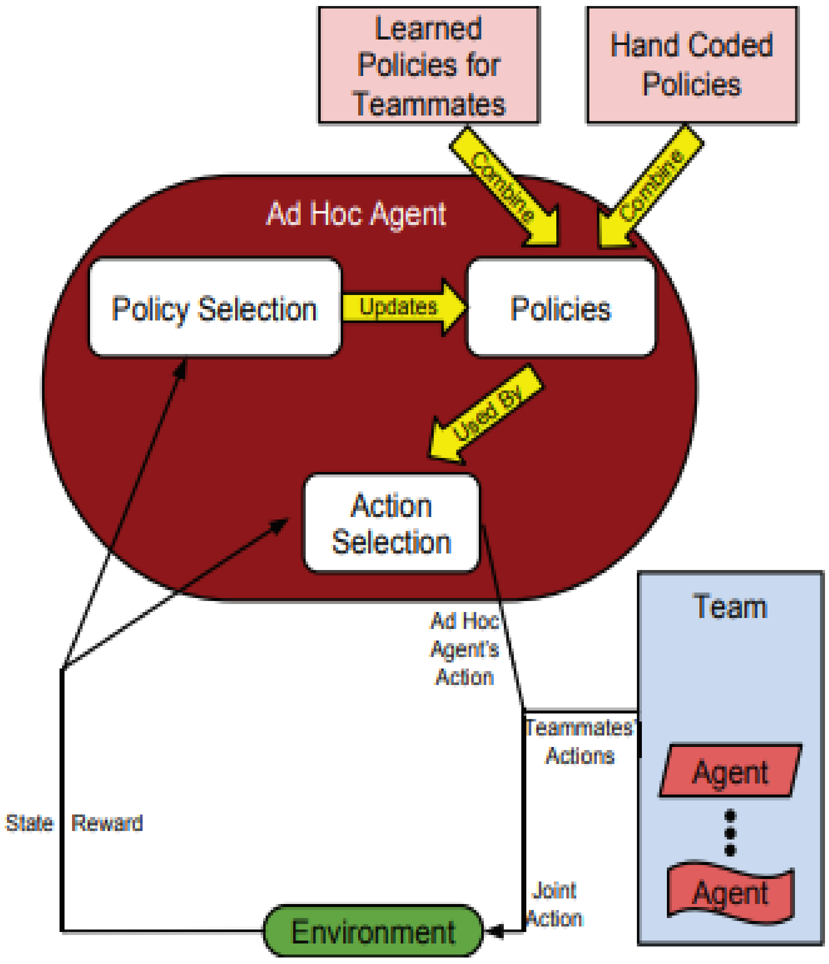


Figure 2.8: PLASTIC-Policy flowchart

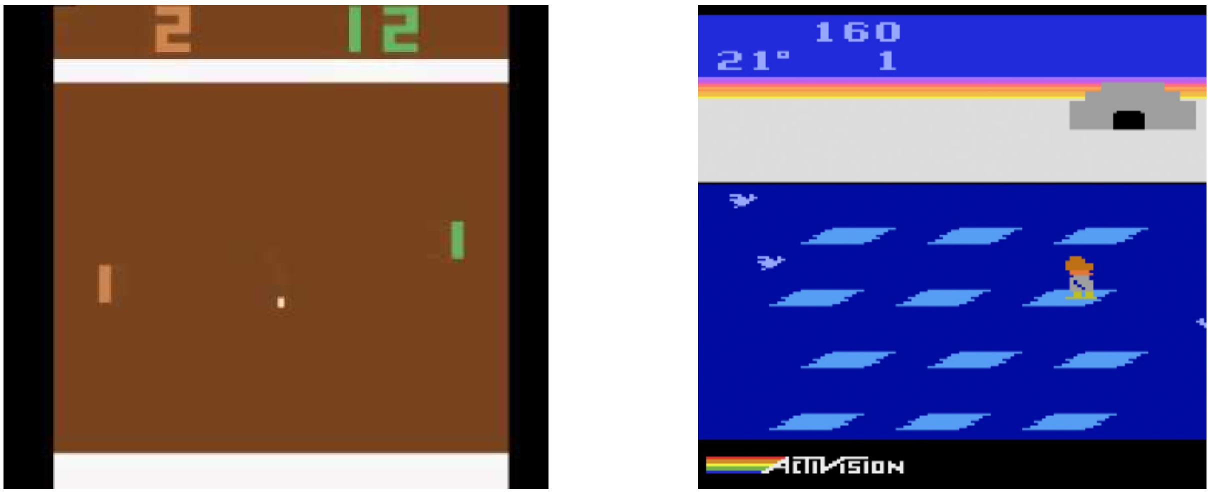
3 Related Work

Contents

This chapter presents an overview of the literature and state of the art that influenced the work of this thesis. Specifically, we start by going over recent works that address reinforcement learning with partial observability, before moving to reinforcement learning in multi-agent systems, and then discussing how to combine planning with reinforcement learning. We conclude by discussing the state of the art in ad hoc teamwork.

3.1 Reinforcement Learning in Partial Observable Environments

Since its introduction, Q-learning has become one of the foundational algorithms in reinforcement learning and has been widely used in various applications and extensions [26]. However, high-dimensional state spaces like those in continuous domains or images are often difficult for Q-learning to handle. [27].



(a) Pong

(b) Frostbite

Figure 3.1: Atari 2600 games

This model-free approach stores the Q-values of each state-action pair in a table, which does not scale well when utilized in more complex domains, since it requires a lot of memory space to store all the possible state-action combinations. The total memory space required would be equal to $|S| * |A|$, where S represents the total number of possible states and $|A|$ the total number of possible actions. Another limitation of this tabular format is that it requires several iterations through every possible state-action pair in order to learn a good representation of the environment, which is not only computationally inefficient but also nearly impossible for more complex domains, such as the Atari 2600 games (fig. 3.1). To solve this type of games, Mnih et al. introduced the idea of DQN (section 2.2) [20]. The DQN algorithm combined deep neural networks, with Q-learning to enable agents to learn directly from raw pixel inputs and achieve human-level performance on several Atari 2600 games.

Despite the groundbreaking results, the work of Mnih et al. assume that all the Atari 2600 games possess the *Markov Property* [28] when given as input four sequenced frames of the environment. However, more complex games cannot be described as a MDP (section 2.1.3), instead due to incomplete information and possible noise they become a POMDP, just as Hausknecht et al. described in their paper [11]. In this paper the authors introduced the concept of DRQN (section 2.3), which combines the DQN method with a recurrent layer, such as the LSTM (section 2.3) [23]. The architecture of the neural network used in the mentioned work can be seen in (fig. 3.2), it follows the architecture employed by Mnih et al with the insertion of a LSTM layer that allows the agent to maintain temporal memory and process sequences of game frames, enabling better decision-making in partially observable settings. This novel approach showed outstanding results, especially for games where the agent is required to plan several steps ahead, such as the frostbite game (fig. 3.1) (b). It also displayed very good results

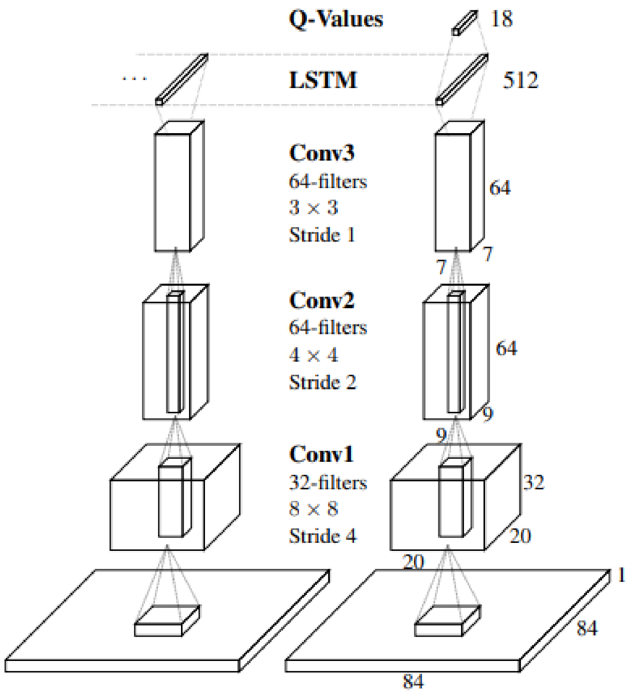


Figure 3.2: DRQN architecture used by Hausknecht et al. to learn to play the Atari 2600 games

when dealing with noise in the environment. When some frames of the game were removed, the DQN showed worse results for games that required knowledge about previous observations, such as Pong (fig. 3.1) (b). The DQN was not able to predict the movement of the ball nor its velocity without having access to the four set of frames, while the DRQN with the ability to keep track of correlations between the sequence of frames did.

3.2 Multi-Agent Systems

The methods mentioned in section 3.1 demonstrated excellent results for single agent tasks, such as the Atari games. However, in the real world some complex tasks are too difficult for a single agent to perform. By enabling multiple agents to cooperatively perform a task, it is possible to reduce its complexity and at the same time make it more computationally efficient. This new type of environment is called a multi-agent environment. ”A multi-agent environment is one in which there is more than one agent, where they interact with one another, and further, where there are constraints on that environment such that agents may not at any given time know everything about the world that other agents know (including the internal states of the other agents themselves)” [29]. MAS proved to be very useful when dealing with tasks that require decentralize reasoning, where each agent makes independent and autonomous decisions based on their local information and individual knowledge. Distributed actuation, where each

Table 3.1: Pong game rewards

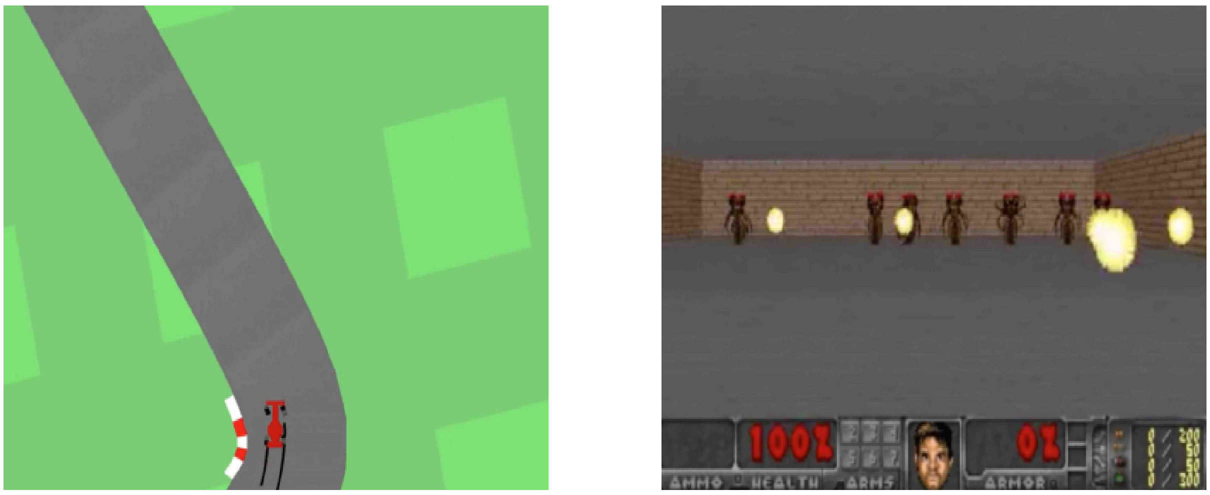
Parameter	Left player scores	Right player scores
Left player reward	ρ	-1
Right player reward	-1	ρ

agent is responsible for executing specific actions or solving specific tasks, and robustness to component failure [3].

The paper presented by Panait et al. cemented the concept of MAS. The authors explained concepts such as team learning and concurrent learning. The former uses single-agent machine learning techniques, while at the same time taking into account the joint behaviour of all agents, just like a team. Team learning alleviates the process of co-adaptation between different type of learners. Concurrent learning, sees each agent with its ”own unique learning process” [29] and trying to maximize its own performance. This method can be very useful when the problem can be sub-divided into smaller tasks, therefore requiring distributed actuation.

The exponentially growing interest in the MAS field caught the attention of other fields, such as the ML one. In 2008 Busoniu et al. presented a paper that discussed the benefits and limitations of the merge of those two fields [30]. The combination of MAS and RL originated the Multi-Agent Reinforcement Learning (MARL). This new ML approach benefited from the previously mentioned MAS' advantages, such as parallel computation that can help speed up the learning process. It also gains robustness, since if one agent fails to complete its task, other agents can cover for it. This approach also allows for possible communication between agents, either through direct contact or through example. Nevertheless, the combination of these two fields also creates some challenges, starting with the fact that managing a single agent is already a difficult task, hence managing multiple agents that require coordination between them turns a into a much more complex task. This challenge adds to the ones produced by the RL methods adopted, such as the *curse of dimensionality* and *exploration-exploitation trade-off* [18].

MARL has been applied in various works throughout the years and we would like to highlight the work of Tampuu et al. [31], which combines DRL with MAS to learn the well-known Atari 2600 game Pong (fig. 3.1 (a)). The authors explore the idea of creating a multi-agent environment where two decentralized agents using a DQN, learn to play the game of Pong from scratch. Each agent is assigned a player paddle and receives a reward according to table 3.1. The ρ may vary from -1 to 1, with -1 indicating a fully collaborative game between the two agents and 1 a fully competitive one. The authors use this reward table to explore the transition between a collaborative and a competitive game between the two DQN agents. In their work they concluded that an agent trained in a multiplayer environment performed better than an agent trained in a single-player setting (trained against the computer algorithm of Pong).



(a) Car Racing **(b)** VizDoom

Figure 3.3: Games used in Ha et al. paper

3.3 Planning and World Models

Sample efficiency is one the most important aspects for an agent, especially considering that sometimes in the real world there may be a scarcity of data or time limitations that restrain the agent's learning process [32]. These challenges can be solved through model-based learning (section 2.1.3.B) and planning. However these methods proved to be heavily dependent on the world model of the environment, hence in 1990 Sutton et al. proposed an approach that integrates a preconceived model of the environment with the traditional model-free algorithm (Q-learning) (section 2.1.3.A) [10]. This method, which the author denominated Dyna-Q, could simulate experiences through the world model, while at the same time gathering real world experiences from the environment to update the agent's policy (section 2.1.3.B). The results displayed in this paper proved that this new approach enables more effective exploration and faster convergence than a simple model-free algorithm. Sutton also suggested using the world model's output as probabilistic distribution, which would allow the world model to be used not only for deterministic environment, but for stochastic ones as well.

The concept of world models was much later expanded by Ha et al. in 2018, who introduced the concept of generating world models based on the idea of the human mind, which is ”not just about just predicting the future in general, but predicting future sensory data” [33]. In this paper, the author creates a compact and versatile representation of the environment that can be used for an agent to learn the optimal policy to play the car racing game (fig. 3.3 (a)). The world model can also be used as dream/hallucination environment for the agent training process, before acting on the real game environment, such as the VizDoom (fig. 3.3 (b)).

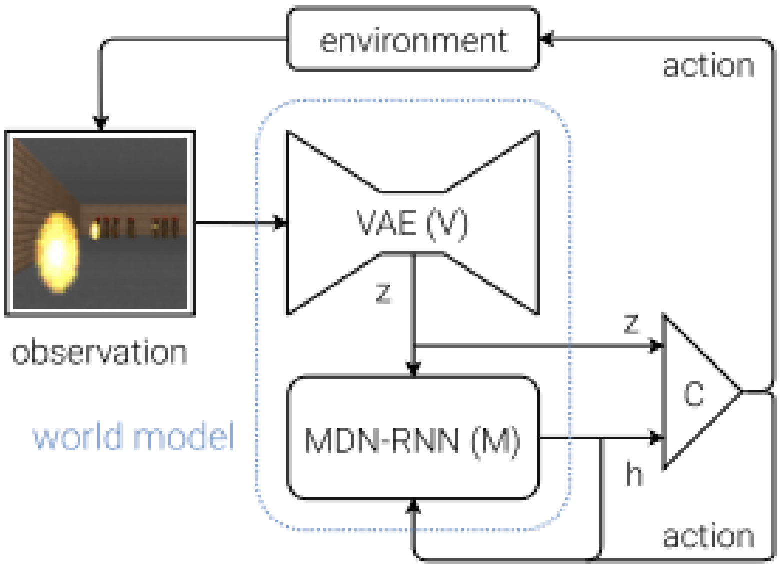


Figure 3.4: Flow diagram of Agent Model used in Ha et al. paper

The architecture used in the Ha et al. work is displayed in fig. 3.4. It uses three main components, the variational auto encoder (V), the mixture-density network recurrent neural network (M) and the controller (C). The V component is utilized to encode each raw sensory input into a low dimensional latent vector. This compressed representation of each observed input frame is then fed into the M component which wields a LSTM recurrent layer (section 2.3). The M model is used to predict the next observations and, since many real world environments are stochastic, the output is a probabilistic distribution, just like Sutton proposed in their paper [10]. For the probabilistic distribution the authors use a Gaussian one, as a result of working in a continuous environment. Lastly, the C component is responsible for determining the course of actions to take in order to maximize the expected cumulative reward based on the observation and next observation received from the world model.

The work of Ha et al. sparked a lot of interest in model-based learning using a world model of the environment. Some of the latest works in the state of the art are the Simulated Policy Learning (SimPLe) [34], the MuZero [35] and the DreamerV2 [36] approaches.

3.4 Ad Hoc Teamwork

The concept of ad hoc teamwork was first introduced by Stone et al. in 2010 [4]. The authors defined the ad hoc setting as a scenario where ”Multiple agents with different knowledge and capabilities find themselves in a situation such that their goals and utilities are perfectly aligned, yet they have had no prior opportunity to coordinate”. They give the example of a scenario where a cyclist falls off the

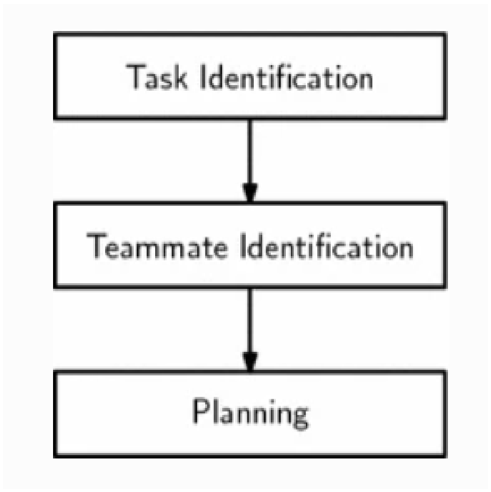


Figure 3.5: Ad hoc teamwork reformulated by Melo and Sardinha

bicycle and gets injured, leading to the nearby people rushing to help with the common goal of saving the person's life, despite not knowing each other beforehand. This create an ad hoc teamwork setting, where every person needs to coordinate on the spot to determine which task each one should execute.

Earlier works assumed that the ad hoc agent knew both the task and the team's policy. Stone et al. used these assumptions to create a multi-armed bandit ad hoc setting, where the ad hoc agent (teacher) would have to guide the non-ad hoc agent (learner) to achieve the common goal [37]. The multi-armed bandit problem is a classic dilemma in reinforcement learning and decision-making. It involves a set of arms, and the agent must decide which of the arms to pull in order to maximize the cumulative reward over time [38]. In later works, Agmon and Stone expand the work to more than one teammate [39].

The previously mentioned works assume that the ad hoc agent has full knowledge about the game (task) and the behaviour of the teammates (policy), which, in certain ways, breaches the idea of ad hoc teamwork, since the ad hoc agent is supposed to coordinate on the spot with unknown teammates. Hence, in recent works, Chakraborty and Stone [6] and Barrett and Stone [5] stopped assuming that the ad hoc agent has access to the teammate's policy. In the latter the authors introduced a version of the PLASTIC algorithm (section 2.4), the PLASTIC-Model. This novel approach utilizes the Monte Carlo Tree Search (MCTS) model-based planner to calculate the optimal ad hoc agent's action based on the current state and policy of the teammate (determined by the PLASTIC algorithm).

Despite the PLASTIC-Model's advancements on the field of ad hoc teamwork, it was still limited by the assumption that the ad hoc agent has access to the problem's target task. Therefore, later, Melo and Sardinha reformulated the idea of ad hoc teamwork into three parts. First, the ad hoc agent has to identify the target task (task identification). Followed by the identification of the teammates' behaviour (teammate identification). Lastly, determine the best action according to the task's goals and teammates' behaviour (Planning), (fig. 3.5)

Later on, Barrett et al. expanded on the previous work [5] and introduced the new PLASTIC version, the PLASTIC-Policy. This approach was in line with the reformulation of the ad hoc teamwork proposed by Melo and Sardinha [7], where the ad hoc agent does not have access to the task's goals. It used the same PLASTIC method explained in section 2.4 with the inclusion of a set of pre-trained policies for each possible task. The ad hoc agent's policy to follow is chosen according to the teammates' predicted behaviour (target task), which is determined based on their current actions. This method still has its limitations, starting by the fact the PLASTIC-Policy ad hoc agent cannot adapt to a set of agents that it never encountered before. The second big limitation is that it assumes that the actions of the teammates are fully visible to the ad hoc agent, which does not happen in partial observable environments, since in the real world most agents can only observe the state of the environment through imperfect sensors.

The most recent work of Ribeiro et al. tackles the challenge of ATPO [9]. The novel contribution proposed by the authors makes use of POMDP solution methods to compute the ad hoc agent's policy. The authors keep a set of beliefs for the agent's possible state, just as in a normal POMDP, with the addition to a set of beliefs for the possible task. Each task is defined as a Multi-agent Markov Decision Process (MMDP), which is defined as (3.1). The "2" is the number of agents (one ad hoc agent and one agent as a team), X_k, r_k and γ_k are the set of states, reward function and discount factor for task k respectively. A^a and A^{-a} is the set of actions for the ad hoc agent and for the non-ad hoc team respectively. Lastly $P_{k,a}$ are the transition probabilities for task k.

$$m_k = (2, X_k, \{A^a, A_k^{-a}\}, \{P_{k,a}, a \in A\}, r_k, \gamma_k) \tag{3.1}$$

Even though, Ribeiro et al.'s work achieved very good results, it is conditioned by the poor scaling of the POMDP-based solution for more complex environments. It also assumes that the ad hoc agent has perfect knowledge about its possible set of teammates.

To surpass the limitation of the ad hoc agent being unable to adapt to a set of never seen teammates, Chen et al. introduced AdHoc Teamwork by Sub-task Inference and Selection (ATSIS). This approach infers the teammates' subtasks based on their actions and observations from the partial observable environment. Similar to Ribeiro et al.'s work [9], the authors utilize a POMDP-based solution to determine the agent sub-task. Then they apply MCTS planning to determine the best actions to perform.

4

PLASTIC-Dyna

Contents

In this chapter we present an overview of our novel model-based RL contribution to ad hoc teamwork, the PLASTIC-Dyna, as well as a detailed analysis of all of its components

4.1 PLASTIC-Dyna Overview

PLASTIC-Dyna is an ad hoc teamwork model-based RL approach that is able to work under partial observability and sample scarcity. Its architecture is composed by two big main components (fig. 4.1).

The first component is denominated DynaDRQN, which consists of the recurrence component (DRQN w/ GRU) and the planning component that resorts to a preconceived world model to generate simulated experiences. The recurrence component is responsible for ability of the agent to keep track of the history

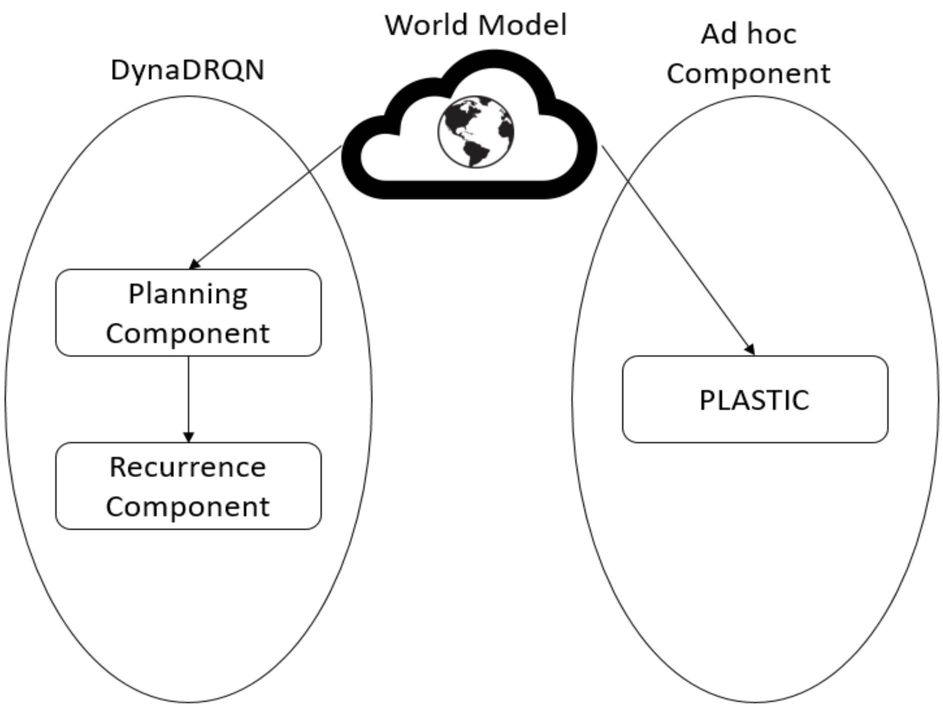


Figure 4.1: PLASTIC-Dyna Overview

of past information and, therefore, being **able to achieve good performances under partial observability**. The Planning component is in charge of storing the simulated experiences generated by the world model in the experience replay memory, which is used in the learning process of the agent. This component allows the agent to **attain sample efficiency**.

The other main component is the ad hoc one, which employs a version of the PLASTIC architecture proposed by Barrett et al. [5]. This PLASTIC version utilizes the preconceived world model to predict the next possible observations of the teammates and, hence, determine the target task based on the observed behaviour of the teammates. This ad hoc component allows the agent **to correctly the identify the target task without access to the teammates' actions**.

4.2 Recurrence component

In this section we present a detailed analysis of the recurrence component, which we denominated as DynaDRQN. This component is at its core a DQN with a RNN, that uses a preconceived World Model to add new simulated experiences into the experience replay buffer. This RNN can either be a GRU or a LSTM (section 2.3), we opted for a GRU layer since it is known to be computationally inexpensive [25].

The DRQN component is illustrated in fig. 4.2. This network receives as inputs a sequence of observations sampled randomly from the experience replay memory/buffer, which then goes through a

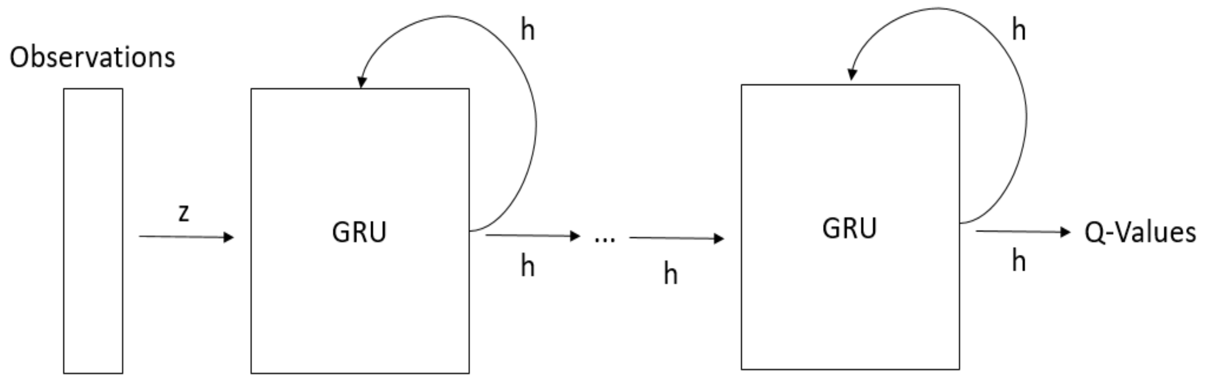


Figure 4.2: DRQN Architecture

number of GRU recurrent hidden layers and outputs the Q-values for every possible action. This recurrent component is what allows our agent to keep track of the history of previous observations. Which can be seen in the mentioned figure, where the hidden value that the GRU layer outputs is also used in conjunction with next input in the sequence of observations to calculate the subsequent output. Thus, **granting our agent the ability to work under partial observability**.

During the training process of our agent, the loss function used to update the weights of the DRQN is the Mean Squared Error (MSE), which is defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{4.1}$$

Where n represents the total number of samples, $\hat{y}$ represents the predicted value and y the real (target) value. For the targets we use the Q-values given by the target network.

4.3 World Model

The world model used for the planning component was trained with partial observable data gathered by a pre-trained DQN with full visibility of the environment. The dataset gathered by the DQN obtained better results when combined with the DynaDRQN and granted a smoother optimization process, than with randomly gathered data. This model is able to predict the next observation of the environment, reward acquired and episode conclusion (terminal) based on the input observation and action taken by the agent.

The world model is trained using a neural network with an hidden recurrent layer LSTM. A detailed decomposition of this network can be seen in fig. 4.3. It receives as inputs the observations gathered from the environment and the one-hot encoded action taken by the agent based on those observations.

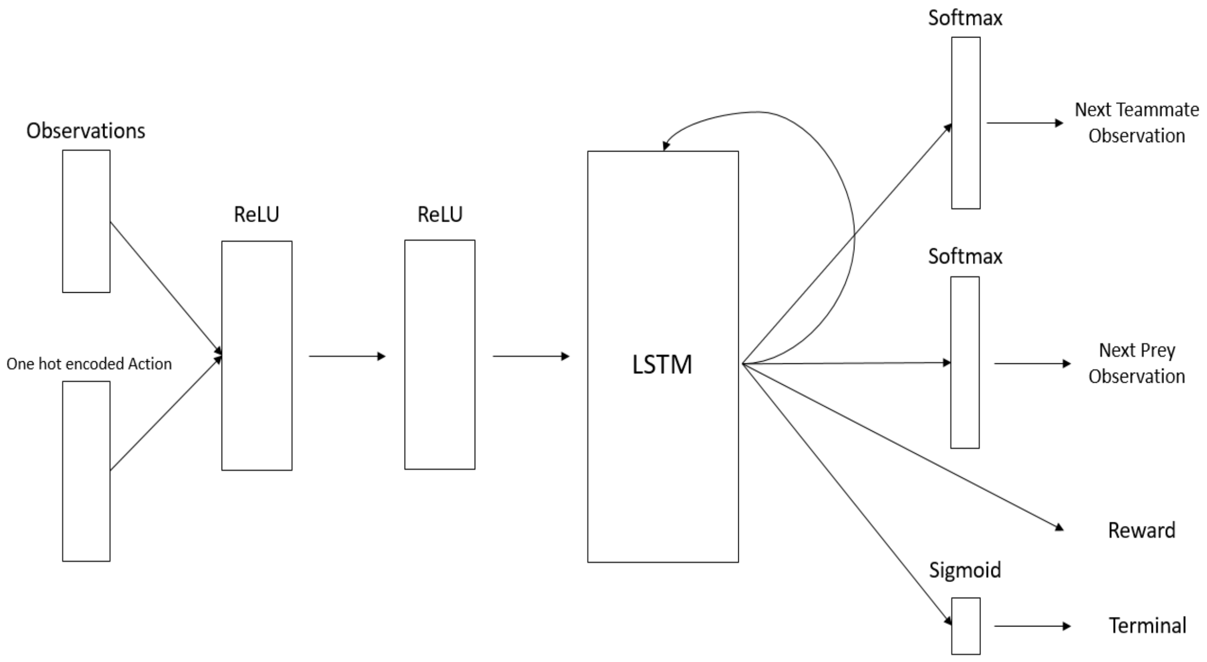


Figure 4.3: World model

It employes an LSTM layer, which is used to maintain store information from the input sequence of observations and actions, similar to the GRU layer in section 4.2.

Finally the output displayed in the figure, represents the next set of observations, in our case (the pursuit domain) we have the teammate and prey's next observation gathered by the agent of the teammate's and prey's position. The next observations' outputs are given by a softmax layer. the decoding process of the observations treats the softmax values as a set of probabilities for the possible next observations. This is due to the partial observability of the environment, where the prey moves randomly and the teammate acts according to information not totally available to the agent. This is another feature **that allows our agent to work in a partial observable environment**. The other two output values are the reward, which is a continuous value that represents the environment's reward based on the action taken by the agent, and the done/terminal value, which goes through a sigmoid layer with the output 1 representing the terminal state.

For the optimization process of the model we use a loss that is calculated for each batch containing sequences of steps. The Negative Log-Likelihood (NLL) was the loss function used to calculate the loss of the teammate and prey's next observations, given by the following equation:

$$NLL = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij} * log(\hat{y}_{ij}) \tag{4.2}$$

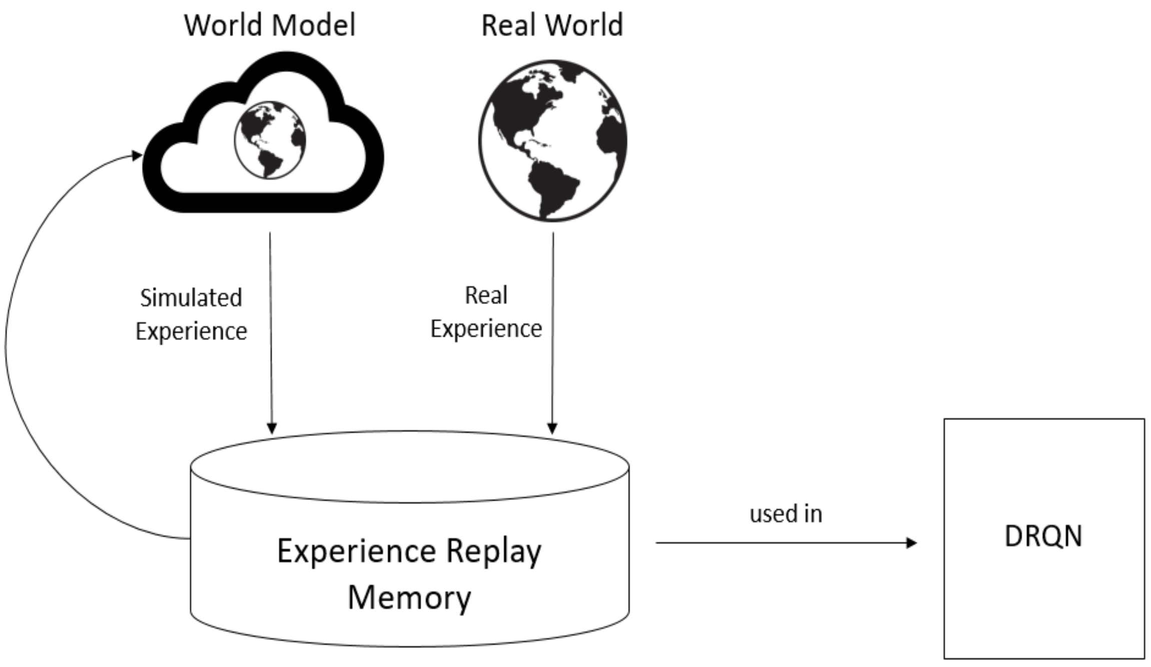


Figure 4.4: Planning Component

Where n represents the number of samples and k is the number of classes for the target (y) and output ($\hat{y}$). The target array will have the target class with the value of 1 and all the others set to 0.

Lastly, the done/terminal loss was calculated using binary cross entropy/log loss, which is the binary version of the NLL (4.2) defined as:

$$logloss = -\frac{1}{n}\sum_{i=1}^{n} y_i * log(\hat{y}_i) + (1 - y_i) * log(1 - \hat{y}_i) \tag{4.3}$$

Where n represents the number of samples. Since the target (y) and output ($\hat{y}$) are binary the subtracting 1 to them gives us the values of the other class.

The total loss value is the sum of all the mentioned losses divided by the number of output nodes, which works as a simple scaling. It is also worth mentioning, that our dataset was very unbalanced, containing mostly non-terminal observations. However, we did not find the need to add different weights to our outputs, since the reward value difference used for MSE loss already fulfills that requirement.

4.4 Planning Process

The planning component of our agent uses information from the experience replay memory, in order to generate new simulated experiences. For each sequence of real steps stored, the agent plans a new

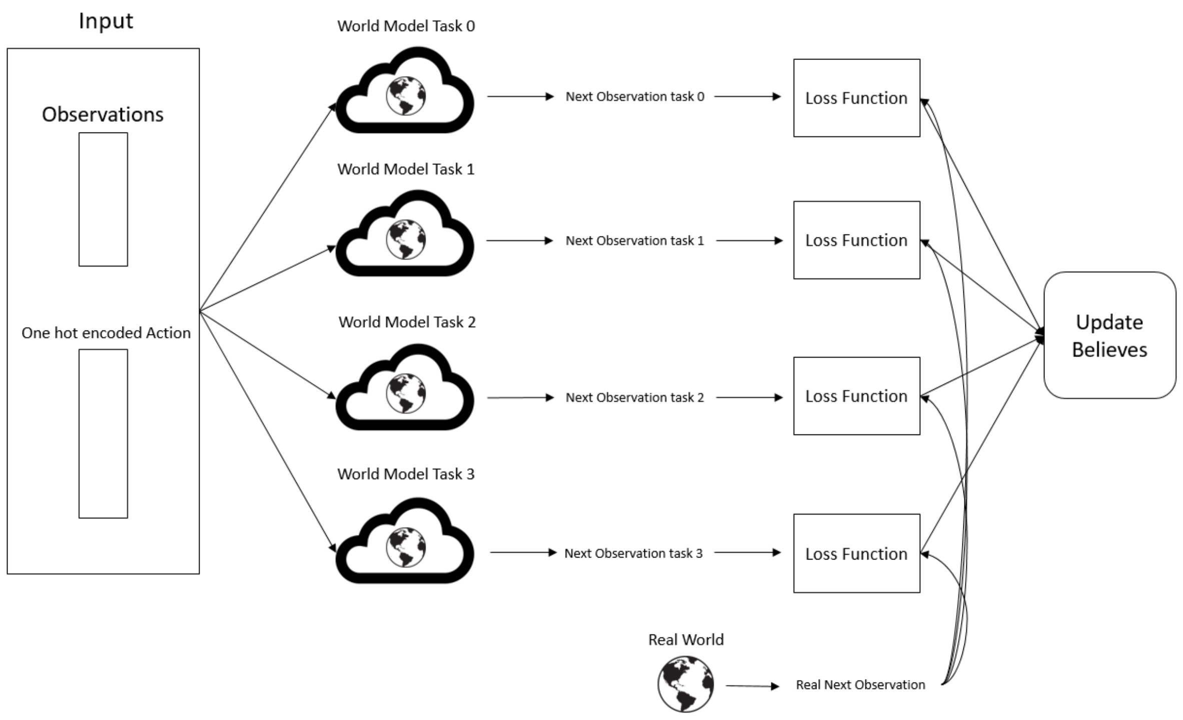


Figure 4.5: Ad Hoc Component

sequence, which is the equivalent of one planning step per real world step. It uses the last non-terminal step information from a randomly picked sequence form the replay memory to generate a new simulated sequence, which is also stored in the experience replay memory, as it can be seen in fig. 4.4. These experiences are then used to optimize the deep recurrent component, the DRQN. It should be pointed out that the agent has knowledge about the possible rewards of the environment. This is necessary for the decoding of the continuous values obtained by the world model's reward output and to make sure that the simulated experiences coincide with what would be a real experience. This planning component only becomes active after a certain number of random steps taken by the agent on the environment. This allows the agent to fill the replay memory with enough information to apply in the planning step.

Whereas a regular DRQN has to balance exploration and exploitation, by using methods such as the $\epsilon - greedy$ [18]. Our contribution just needs to use the planning component to generate simulated experiences, while simultaneously following the current policy. This component, based on the Dyna-Q (section 2.1.3.B), is what **allows our agent to attain sample efficiency**.

4.5 Ad hoc teamwork

The ad hoc component contains the novelty of our contribution. As we previously mentioned, Ad hoc teamwork under partial observability still has a lot to be explored. Yet, we were able to create an agent capable of identifying the target task solely by having access to observations gathered from a partial observable environment.

Our approach uses a variation of the PLASTIC algorithm presented in section 2.4, in conjunction with a world model section 4.3 for each task, where each unknown set of teammates acts differently. The world model used for this component has some slight changes when compared to the one used for the non-ad hoc component. First of all, the loss function only uses the observation of the teammate's location to calculate the respective loss. This is due to the fact that only the teammates' behaviour will be used to identify the target task. Hence the second change, we chose a DRQN agent to generate the dataset that we utilized to train the world model since we required a more diverse set of teammates' observations to identify the task. However, due to the unbalance coming from the high number of observations outside of the agent's field of vision, to guarantee that our world model predicts the visible observations correctly, we chose to increase the weight of the observations gathered from the agent's visible cells.

The PLASTIC algorithm used in our approach is exhibited in fig. 4.5. It employs the world model to generate the predicted next observation of the teammates' position, given the current observation and action taken, for each possible task and compares each prediction with the real next observation gathered by the agent provided from the interaction with the real world. This information is then used to calculate the loss for each possible task, according to eq. (2.13) in section 2.4. Which is then used to update the set of task's beliefs by applying the eq. (2.14) in the same section. At each step in the environment, the ad hoc agent chooses the policy to follow based on the task with highest belief value. The plastic agent chooses the policy of the non-ad hoc component of our contribution, the DynaDRQN, which was pre-trained for each possible task. The combination of the PLASTIC algorithm and the DynaDRQN conceives the PLASTIC-Dyna, which **gains the ability to correctly predict the target task in a partial observable environment** with this ad hoc component.

5 Results

Contents

In this chapter we are going to present and analyse the results obtained in our work. We employ a set of baselines in the pursuit test environment, in order to compare the performance of our own contribution, the PLASTIC-Dyna (section 2.4), by making use of a well-defined set of metrics.

Most of the results are displayed through either a bar plot or a line plot. Both of these feature the Standard Error (SE), defined as $\frac{\sigma}{\sqrt{n}}$, calculated using a confidence of 0.95%. The bar plots are used to showcase the accumulated reward of an agent after trained. While the line plots are used to display the training learning curves and also the task beliefs' progression for the ad hoc component. Every agent is trained three independent times and the accumulated results are obtained from one hundred test runs for each of the three different trains. Each agent is trained for all the possible tasks, which leads to a total of twelve independently trained agents.

5.1 Evaluation

To evaluate the quality of our contribution, we defined a set of metrics to be used in a well-known MAS benchmark environment, the pursuit domain. We introduced partial observability by limiting the field of view of our agent to one cell of distance and introduced some noise on the observations gathered from the environment. To measure the **ability to deal with partial observability** of our agent, we compared the non ad hoc version's accumulated reward on the environment with other notable non ad hoc baseline methods, such as a DQN with full observability, a DQN with partial observability and a DRQN (using a GRU recurrent layer) also with partial observability. We used a bar plot with the average accumulated reward for all of these agents to evaluate the results.

In order to test the **sample efficiency** of our approach, we compared its training plot line with the previously mentioned baselines. To evaluate these results, we examined the number of steps that it took for the average accumulated reward to start converging, with few number of steps indicating a more sample efficient approach.

Lastly, we measured our ad hoc agent's **capability to identify the target task under partial observability**. We plot the progression of the agent's beliefs for each task throughout the agent's interaction with the environment. While also comparing the accumulated reward of our contribution with baselines that have access to the actions of the teammates.

5.1.1 Test Environment

The environment used for this work was the well-known MAS benchmark, the pursuit domain, Figure 5.1. More specifically, the two predators versus one moving prey domain. The agent's teammate has full observability and follows a greedy policy. We assume that our agent does not have the same field of vision as its teammate, in fact our agent can only see as far as one cell of distance. The observation obtained from the environment has a 15% probability of containing noise, which means that the teammate or the prey might not be visible while in the agent's field of vision.

The movement within this 5x5 grid domain can only be done within the four cardinal directions and the agent only advances one cell at a time. Therefore, the distance between the teammate and prey is calculated using the Manhattan distance, as:

$$distance = \sum_{i=0}^{n-1} |s[i] - t[i]| \tag{5.1}$$

Where *s* and *t* represent the source and target coordinates, respectively, and n represents the dimension of the domain. In our test domain, we have a two dimensional environment. The distance in our environment is also toroidal, which is very common in game-like domains, such as pac-man and the

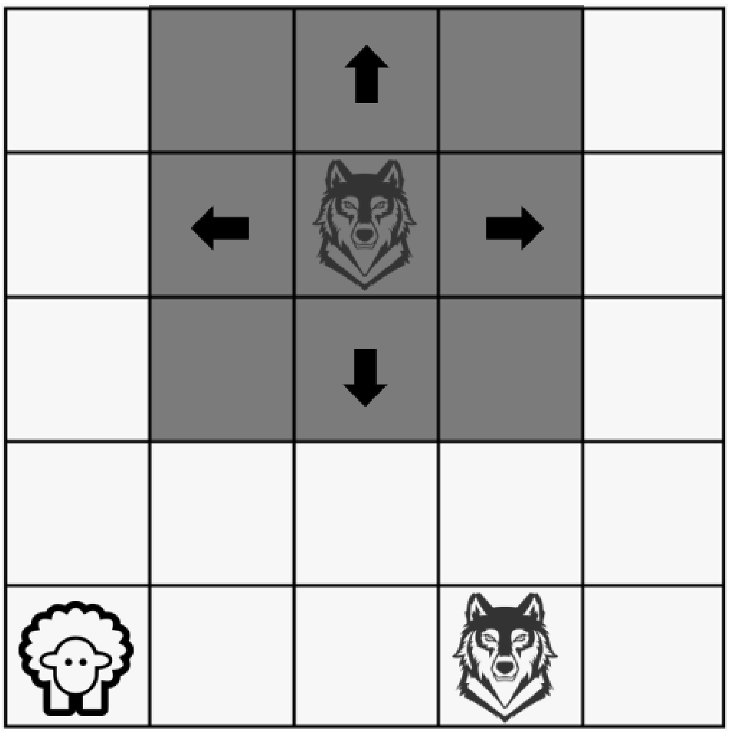


Figure 5.1: Pursuit Environment

snake game. The toroidal distance assumes that the agents are not bound by the edges of the grid, hence the agent can go through a ”wall” and appear on the opposite side of the grid.

Our domain considers four possible ways for the predators to capture the prey, north/south, west/east, southwest/northeast and northwest/southeast. Each is associated with a task, numbered from 0 to 3 respectively. As mentioned previously the ad hoc agent does not know the task required to capture the prey.

The rewards given by the environment can be -1, when the action taken did not lead to the capture of the prey, 100, when the prey is captured or 0 if the state is terminal. The terminal variable will be used to determine if the environment needs to be reset or not. The pursuit environment is episodic, which means it resets after the catch of the prey. This reset can also happen if the predators fail to capture the prey within the established horizon, which in our case is 500 steps.

5.1.2 Baselines

In this section, we will briefly describe the algorithms that we used to compare the performance of our contribution.

- **Random Agent**: This agent selects actions randomly, therefore, it is the lower bound for our performance comparisons

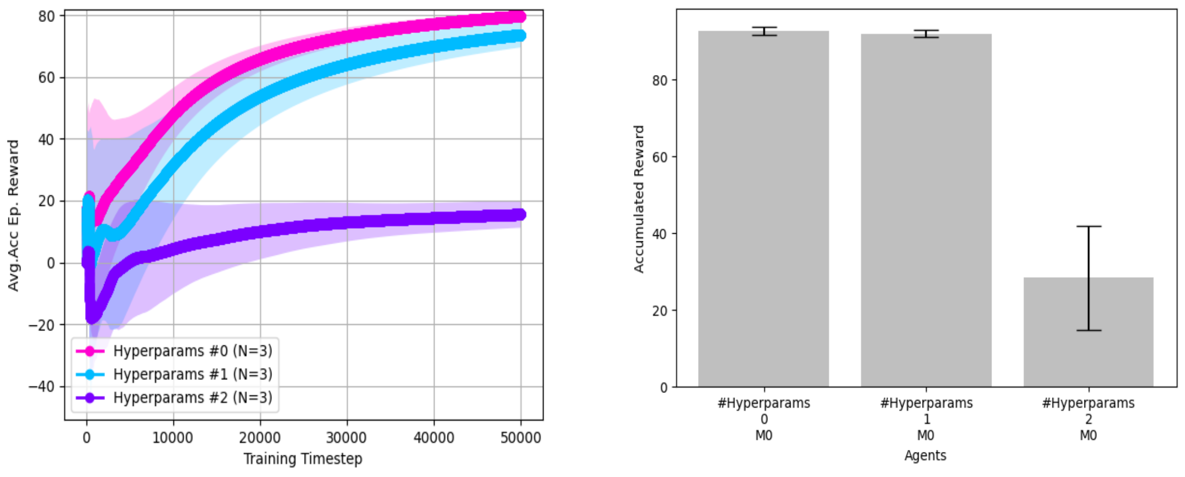
- **DQN Agent**: This agent uses the policy learnt from a DQN to selects its actions. It knows the target task and has full visibility over the environment. This agent can be seen has the upper bound for our performance comparisons
- **PODQN Agent**: This agent extends the DQN agent. However, it has partial observability
- **DRQN Agent**: This agent uses a DQN with a GRU recurrent layer to learn the optimal policy. It also has partial observability
- **PLASTIC-DQN Agent**: This agent is an extension of the DQN agent without access to the target task. It can be seen as the ad hoc upper bound for our performance comparisons
- **PLASTIC-PODQN Agent**: This agent extends the Partial Observable Deep Q-Network (PODQN) agent, whereas it does not have access to the target task
- **PLASTIC-DRQN Agent**: This agent is an extension of the DRQN agent that does not have access to the target task

It is worth noting that all the PLASTIC agents used as baseline have full access to the actions and policies of their teammates.

5.1.3 Metrics

To measure the quality of our approach we used three different metrics. These were used alongside line and bar plots containing the pre-defined baseline agents' results, to determine the quality of our contribution, the PLASTIC-Dyna. The three metrics used were the following:

1. **Ability to deal with partial observability**: We evaluate the ability to act in a partial observable environment by limiting an agent's field of vision and inserting noise into the observations captured from the environment. High accumulated rewards under these conditions express a strong ability to deal with partial observability
2. **Sample efficiency**: Sample efficiency is correlated to the capability to obtain good results when restrained by the number of interactions with the environment, either by an exploration limitation or a reduced number of training steps. We use the accumulated reward and the training line plot convergence to determine sample efficiency
3. **Capability to identify the target task**: The ability to correctly identify the target task (from a set of possible tasks) showcases the capacity of an agent to work in an ad hoc teamwork setting. We utilize the beliefs' update progression for each task to evaluate the ability of an ad hoc agent to identify the target task. This ability is present when there is a steep convergence of the target task belief and high accumulated rewards in the ad hoc setting.



(a) Accumulated Reward for the Learning Process

(b) Accumulated Reward in 100 Test Runs

Figure 5.2: Learning Rate Hyperparams (#0 = 0.01, #1 = 0.001, #2 = 0.1)

5.2 Baselines Deployment

The first step of our work was to set up the baselines and observe how they perform in the benchmark pursuit environment.

5.2.1 Hyper-parameter Tuning

5.2.1.A Learning Rate

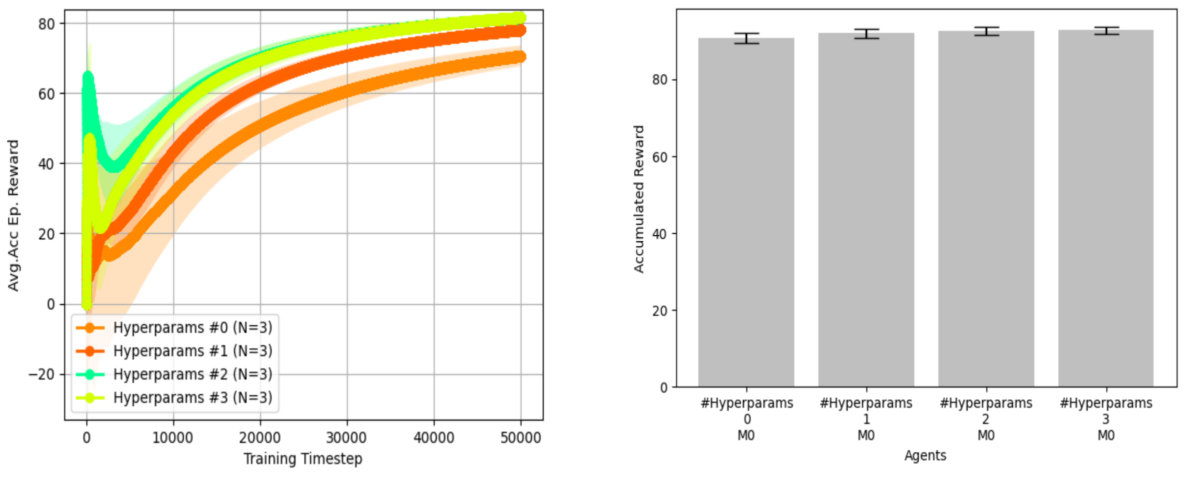
The learning rate is a hyperparameter that determines the intensity of each update during the training process.

In **Figure** 5.2 (b) it is possible to observe that a very high learning does not allow the agent to learn the optimal solution. On the other hand, a very small learning rate takes longer to converge to the optimal solution, as it can be seen in **Figure** 5.2 (a). Therefore, we opted to go for the middle-ground learning rate value of 0.01.

5.2.1.B Batch Size

The batch size hyperparameter determines the number of samples used for each iteration of the optimization step of the network.

From the results obtained during the tuning of the batch size, we noticed that the influence of this hyperparameter is very similar to the learning rate. In fig. 5.3 (a), it is possible to observe that a smaller



(a) Accumulated Reward for the Learning Process **(b)** Accumulated Reward in 100 Test Runs

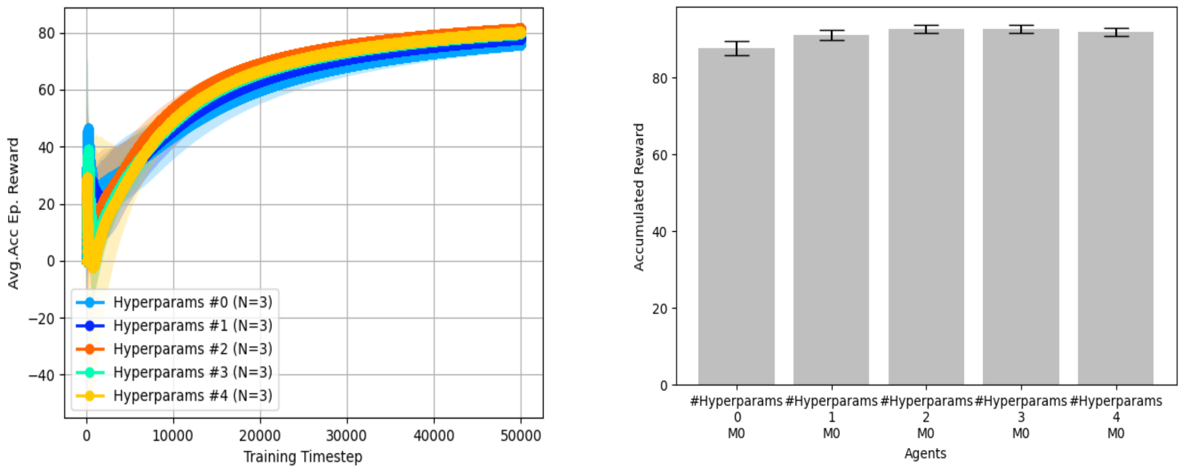
Figure 5.3: Batch Size Hyperparams (#0 = 8, #1 = 16, #2 = 32, #3 = 64)

batch size leads to a slower convergence, while an agent consisting of a larger one converges faster to the optimal accumulated reward. A larger batch size grants the network more meaningful and correct updates during the optimization process, whereas a smaller batch size is susceptible to noise, which can lead to a less stable convergence. The overall results of each agent were very similar (fig. 5.3 (b)), however based on the convergence rates, we opted for a batch size of 32 for the DQN agent.

5.2.1.C Hidden Layer Size and Number of Hidden Layers

The number of hidden layers and the size of each hidden layer impact the agent's ability to learn complex representations and generalize well to new data. A larger number of neurons or larger hidden layer sizes allow the network to capture more convoluted patterns and relationships in the data but can also increase the risk of overfitting.

The results obtained for the hidden layer hyperparameter tuning demonstrate that in low complex environment such as the pursuit domain, this hyperparameter does not have a huge influence in the test runs' accumulated reward, fig. 5.4 (b). The DQN agent consisting of a network with one 32 nodes' hidden layer was the only one that underperformed compared to the rest. It is also able possible to notice that a lower number of hidden layers leads to a slower convergence during the training process, as it can be seen in fig. 5.4 (a) for the two agent utilizing only 1 hidden layer. All the deep networks with more than two hidden layers showcase very good results. Hence, we opted for a network with 2 hidden layers and 64 nodes each, since it not only presented good results but also a decently fast accumulated reward convergence, without "sacrificing" computational power.



(a) Accumulated Reward for the Learning Process **(b)** Accumulated Reward in 100 Test Runs

Figure 5.4: Hidden Layer Size Hyperparams (#0 = 32 , #1 = 128, #2 = 128, #3 = 64, #4 = 64) and Number of Hidden Layers Hyperparams (#0 = 1, #1 = 1, #2 = 2, #3 = 2, #4 = 3)

5.2.1.D Target Network Update Frequency

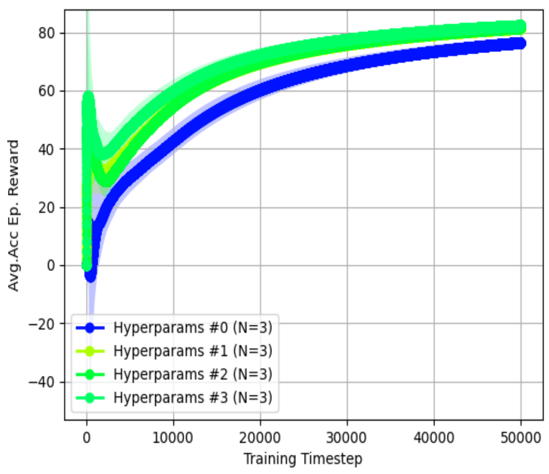
The target network update frequency refers to how often the parameters of the target network are updated to match the parameters of the main network.

When tuning this hyperparameter, we noticed that it does not have a big impact on the agent's overall performance of the agent fig. 5.5 (b). However, a very low target update frequency leads to a much slower convergence of the training accumulated reward, fig. 5.5 (a). For the higher values (8, 16, 32) both the results and the learning plot lines were almost identical, hence we chose a target update frequency of 8 for our DQN agent.

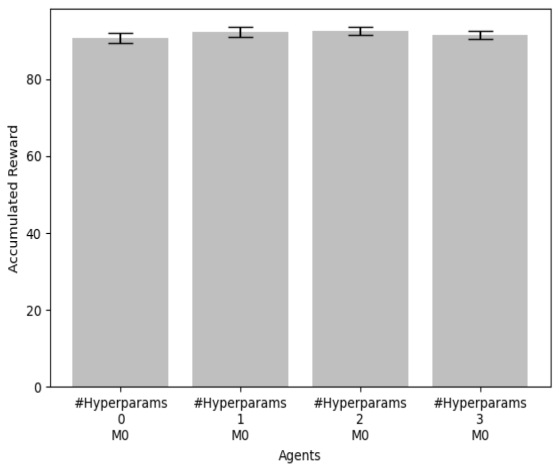
5.2.1.E Sequence Data Length

The amount of successive time steps that the DRQN utilizes as input to make predictions is referred to as the sequence data length.

The tuning of this hyperparameter is displayed in fig. 5.6. The average accumulated reward in fig. 5.6 (b) show that smaller sequences achieve the worst results in a partial observable environment and that by increasing the sequence length of the inputs the overall performance of the agent also increases. It is also possible to observe that the difference between the 16 and 32 sequence lengths is very small, which makes us assume that increasing the length even further will only increase the computational power required and not the accumulated reward of the agent. Therefore, we picked the 16 sequence data length value, which allows our agent to learn a good policy in a partial observable environment

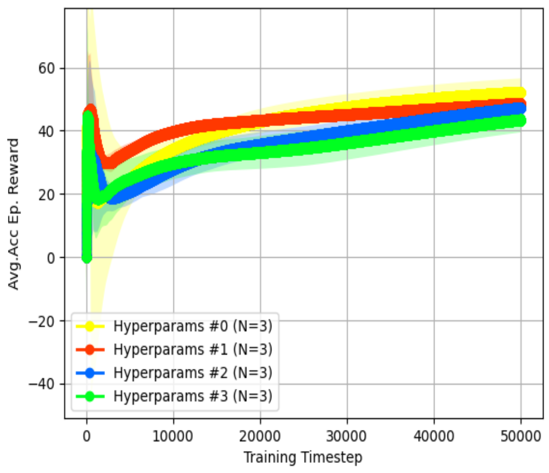


(a) Accumulated Reward for the Learning Process

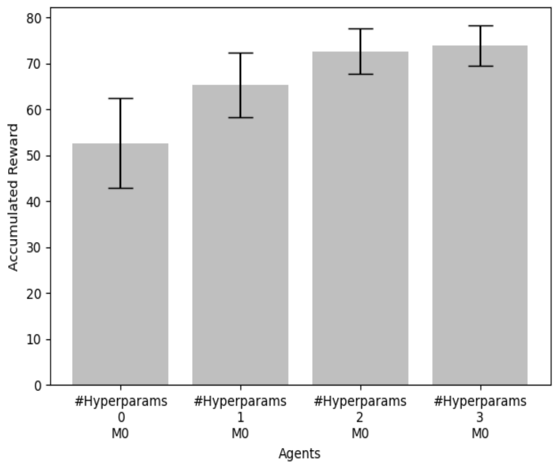


(b) Accumulated Reward in 100 Test Runs

Figure 5.5: Target Update Frequency Hyperparams (#0 = 2, #1 = 8, #2 = 16, #3 = 32)



(a) Accumulated Reward for the Learning Process



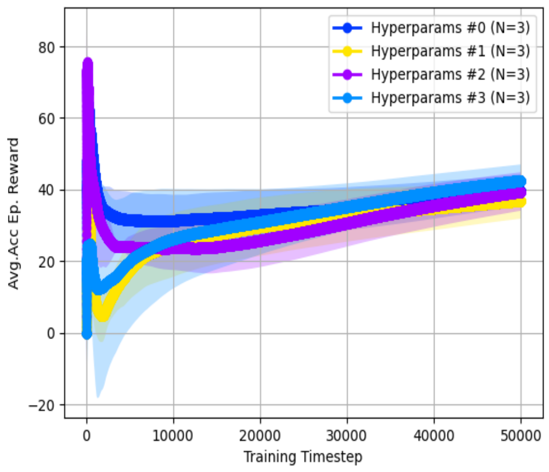
(b) Accumulated Reward in 100 Test Runs

Figure 5.6: Sequence Data Length Hyperparams (#0 = 4, #1 = 8, #2 = 16, #3 = 32)

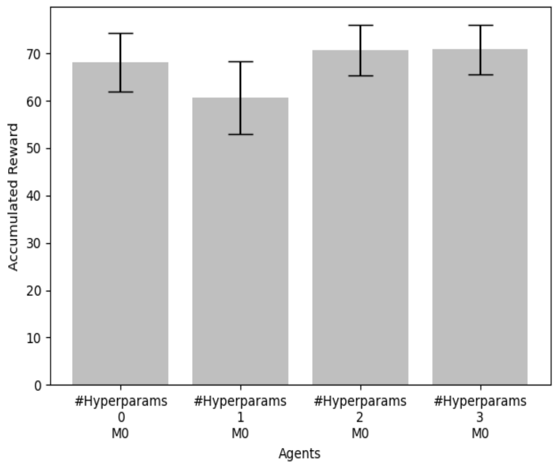
without requiring much more computational power. From fig. 5.6 (a) we can observe that the smaller sequence length agents converge faster for a sub-optimal solution.

5.2.1.F Exploration Rate

The exploration rate is a very important hyperparameter in RL, it determines the balance between exploration and exploitation during the learning process. In this work we adopt a $\epsilon - greedy$ approach



(a) Accumulated Reward for the Learning Process



(b) Accumulated Reward in 100 Test Runs

Figure 5.7: Initial Exploration Rate Hyperparams (#0 = 0.95, #1 = 0.05, #2 = 0.55, #3 = 0.55) Last Exploration Step Hyperparams (#0 = 50000, #1 = 1000, #2 = 25000, #3 = 5000)

combined with a linear annealing to gradually reduce the exploration and increase exploitation. The initial ϵ is given by the initial exploration rate hyperparameter and the final ϵ is always 0.05 in the following tests. The linear annealing function starts after 1000 timesteps, before that the agent is only focused on exploring, and stops at the last exploration step hyperparameter.

The results displayed in fig. 5.7 demonstrate that very long exploration can lead to a very slow convergence of the average accumulated reward, as it can be seen for hyperparams #0 and #2. On the other hand, a short exploration does not allow the agent to fully understand the environment and leads it to learn a sub-optimal policy, as shown by the #1 hyperparameter's results. The #3 hyperparameter results display the best trade-off between exploration and exploitation found for the DRQN agent.

5.2.2 Baseline Results

In fig. 5.8 we display the average accumulated reward obtained by all the non-Ad Hoc baselines for all the possible tasks, using the hyperparameters presented in table 5.1. The results obtained are in line with what was expected. The DQN with full visibility obtains the highest reward values, therefore serves as the upper bound for our agent. Next, we got the DRQN followed by the PODQN, both of these agents have partial observability, however, since the DRQN is able to keep track of information about the previous steps, the history, it can easily identify patterns and correlation in the sequence of observations and achieve a better performance. Hence, we conclude that a recurrent layer grants the agent the **ability to deal with the partial observability constrainment**.

Table 5.1: Hyparameters of all the non-ad hoc agents

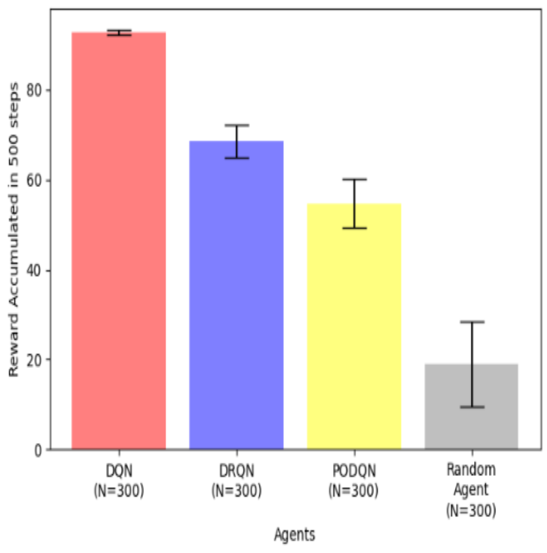
Parameter	*DQN/PODQN*	*DRQN*	*DynaDRQN*
Batch Size	32	32	32
Hidden Size	64	64	64
Learning Rate	0.01	0.01	0.01
Number of layers	2	2	2
Target update freq.	4	8	8
Sequence Length	-	16	16
Initial Updating Step	100/1000	1000	100
Last Exploration Step	10000	5000	100
Initial Explor. Rate	0.95	0.55	0.05
Final Explor. Rate	0.05	0.05	0.05
Planning Steps	-	-	1
Final Planning Step	-	-	5000

In fig. 5.9 we present the line plots for the learning phase of each agent. We limited the training process to 50k timesteps, due to the lack of computational power. However, since one of the goals of this work was to deal with sample scarcity, it made sense to set this limitation. The results, as we can see, are once again exactly what we would expect. The agent with full visibility achieves better results and very quickly, as it can be seen by the steep learning curve. The DRQN and PODQN, on the other hand, require many more iterations through the environment.

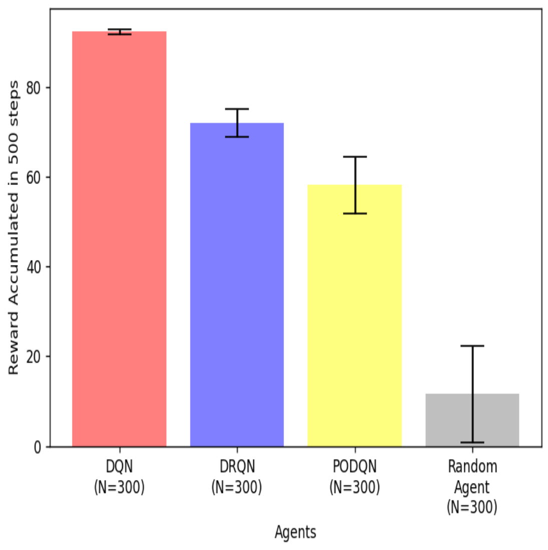
5.3 World Model

The world model is trained using a Neural Network with an hidden recurrent layer with 64 hidden nodes. The data gathered by the DQN agent is saved in 8000 files, each one with a different run on the environment. The files are divided into train and test data, and for each epoch, the network uses 500 train data files and 200 test data files.

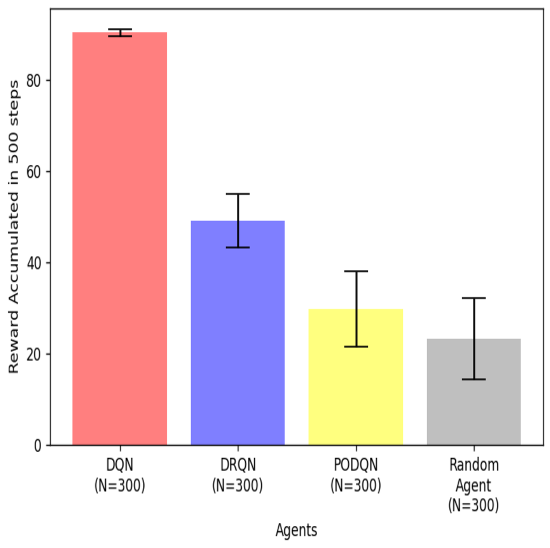
The weights of the network are updated after determining the loss for each batch of 16 sequences with 8 timesteps each. The RNN receives as input 4 toroidal distance values, the first related to the teammate and the second to the prey. These represent the gathered observation by the agent on a certain state (position). It also receives as input the action taken on the current observation as one hot encoding with 5 classes (up, down, left , right, stay). The output is followed by a softmax layer for the teammate and prey's next observation. These turn into an output with 9 classes, each representing the probability of being at a certain cell of our agent's field of vision. These are numbered from top to bottom and left to right. The number 4 represents the location of the agent, but also every cell outside of the agent's field of vision, since we assume that the agent cannot see under its own location. The other two output values are the reward, which is a continuous value and the "done" value, which goes through a



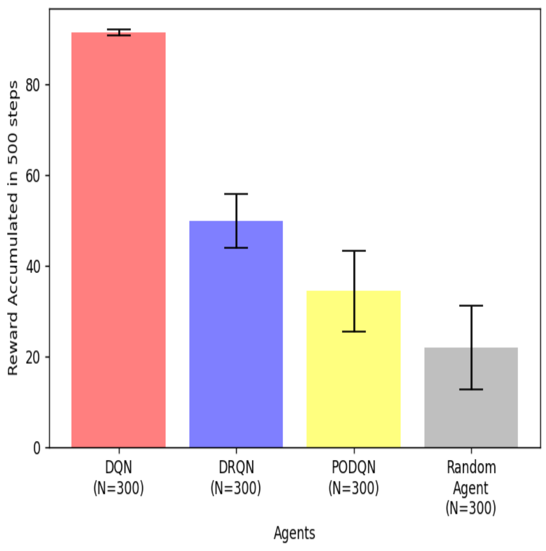
(a) Task 0



(b) Task 1



(c) Task 2



(d) Task 3

Figure 5.8: Non-ad hoc baselines' accumulated reward

sigmoid layer, with 1 representing the terminal state.

During the learning process of the network, we employ an Adaptive moment estimation (Adam) [40] optimizer with a learning rate of 0.001. The Adam optimizer is one of the latest advancements in the neural network's optimization process. It uses adaptive learning rates for each parameter in the neural network, providing a balance between convergence speed and stability. Adam also incorporates momentum to control the update of each parameter. It makes use of the history of previous gradients to

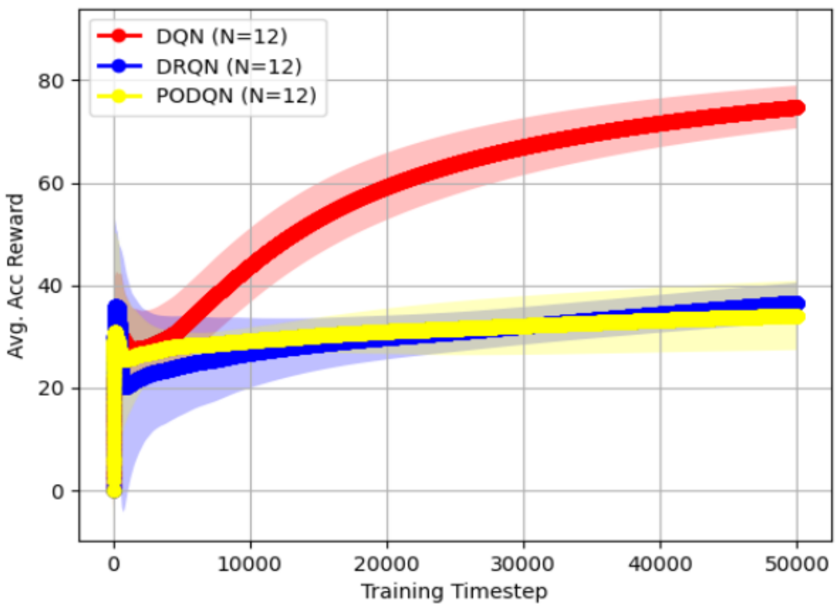
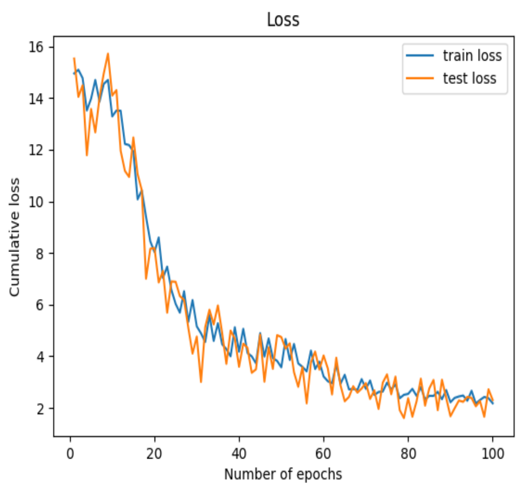


Figure 5.9: Accumulated Reward for the Learning Process of all the non-ad hoc baseline agents

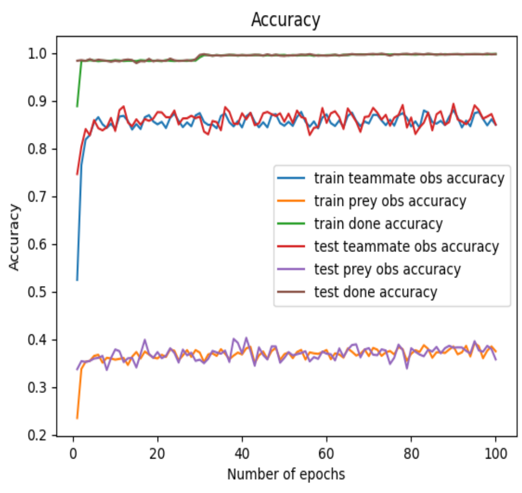
determine weather to ”slow down” or ”speed up” the updates, granting the ability to converge faster and at the same time not getting stuck in a local minimum or overshooting the global minimum.

In fig. 5.10 (a) we display the average accumulated loss for each epoch and in fig. 5.10 (b) the accuracy for teammate and prey’s observation and the done/terminal state value. The first figure shows that the cumulative loss decreases through each epoch and converges at the end, which means that the world model in fact learns to map the inputs to the target outputs. On the other hand, the second figure helps to showcase the nature of the problem a bit better. The accuracy of the prey’s observations is around 40%, which is mainly due to the randomness of its actions, but also due to the partial observability of the environment, where the prey can appear in any position of the field of vision of the agent from the outside. The same goes for the teammate’s observations, because even though the agent has a more predictable behaviour (greedy), it still has access to information unknown to the agent (full visibility of the prey’s position). Hence, the accuracy of around 85% for the teammate’s observation. Lastly, the done/terminal state value has an accuracy of around 100%, however, it is worth noting that the dataset consists of more non-terminal states than terminal ones.

We employ the produced world model in the planning component of our model-based contribution, since a world model capable of reproducing simulated experiences as close to the real world is a crucial step to attain **sample efficiency**.



(a) Average Accumulated Loss over 100 epochs



(b) Train and Test accuracy for the observations and done/terminal output values

Figure 5.10: World Model Training Plots

5.4 DynaDRQN Results

In this section, we introduce the first component of our contribution, the deep model-based RL agent, DynaDRQN. By combining the strength of the DRQN with our preconceived world model, described in section 4.3, we are able to attain **sample efficiency** while maintaining **good results in a partial observable environment**.

The DynaDRQN agent does not get to explore the environment as much as the DRQN agent. Instead, it generates simulated experiences from the world model, producing the equivalent to one simulated experience per real experience obtained from the environment.

In fig. 5.11, we compare the results obtained by the DynaDRQN with non-ad hoc baselines when trained over 50k steps, using the hyperparameters displayed in table 5.1. Our contribution showcases an higher accumulated reward than all the other partial observable agents. This mainly due to the fact that an agent consisting of a DRQN requires more than 50k steps to find the optimal policy. Whereas, our model-based approach benefits not only from the world model to generate more experiences to the replay memory, but it also does not require as many exploration steps as the the other non-model-based agents. Nevertheless, we expect the results of the DRQN and DynaDRQN to be very similar when not limited by the number of training steps.

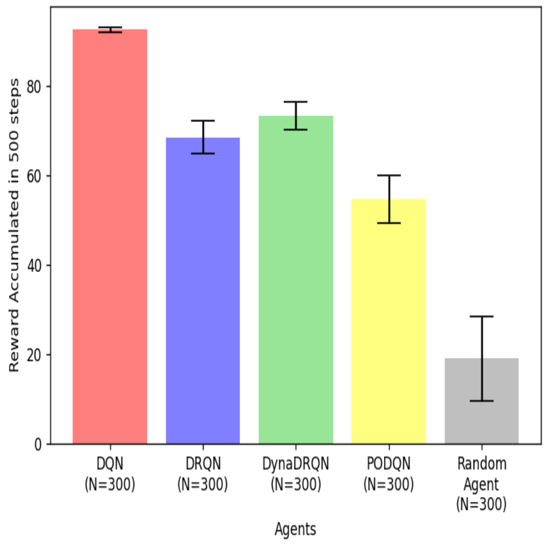
With the average accumulated reward displayed in fig. 5.12, we get a better view of the differences between each agent. The random agent acts as the lower bound for our contribution. It is followed by the PODQN agent with just one observation as the input for the neural network, hence not able to keep track of the history of past observations. Therefore unable to determine the optimal policy for the problem in a partial observable environment. The next agent with the highest accumulated reward is the DRQN agent, which is able to keep track of past information through its GRU recurrent layer. However, it suffers from the sample scarcity of the environment. On the other hand, the DynaDRQN proves to be capable to achieve better results under the two imposed limitations (partial observability and sample sacrcity). The results' difference between our approach and the DQN agent, confirms that not having full visibility of the environment has strong detrimental effect on the agents' performances.

To further prove the sample efficiency of our method, we analyze the training line plots over the pre-established number of steps limitation (fig. 5.13). In fact, it is possible to observe that the DynaDRQN tends to converge faster than the other partial observable agents. This is in line with was expected since the agent does not require many exploration of the environment. We have also noticed in section 5.2.1.F that a DRQN agent with the same exploration as the DynaDRQN, but without the planning component is not able to achieve a performance as good as our method. Therefore, we can confirm that our contribution is more **sample efficient** than all the partial observable baseline methods due to the application of a planning component with a world model.

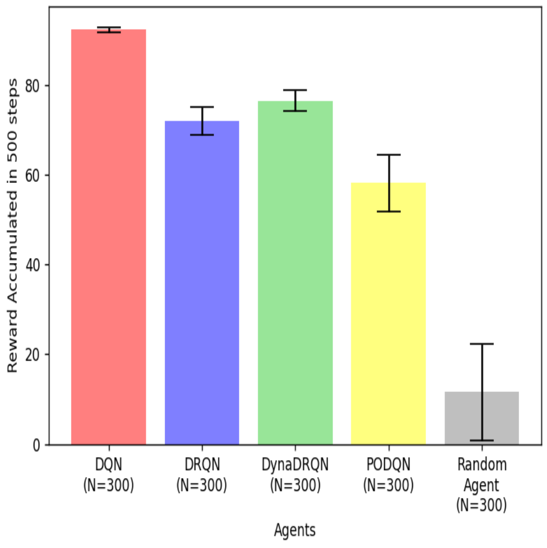
5.5 Ad hoc Results

In this section, we tackle the ad hoc component of our contribution. We utilize the ad hoc baselines as comparison for the results of our approach. These baselines have access to the actions of the teammate as well as their policies, which is used to output the possible action of the teammate given its observed position. The predicted action is then compared with the real action of the teammate, in order to identify the target task. It is important to note that the teammates' policies that the agents have access to in the PLASTIC version (algorithm 2.2) is the one utilized by the teammates in a full observable environment. Hence, it was expected that that baselines employing this method would not be able to perform well in a partial observable environment, where the agents gather observations instead of having access to the state they are in.

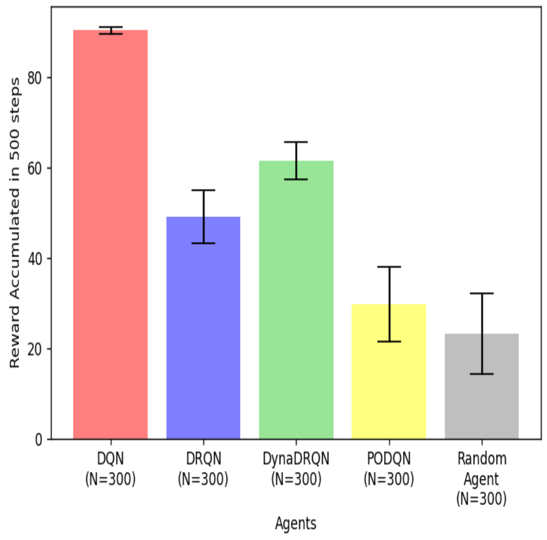
On the other hand, our contribution is able to work with unknown agents without having access to their actions in a partial observable environment. Equally to the baselines, the PLASTIC-Dyna also does not have access to the teammates' behaviour or task. It needs to learn through interaction with the current set teammates to determine which possible task their behaviours correspond to, since each set of teammates acts differently depending on the task expected to be executed. Unlike the other ad hoc baselines, our approach does not have to the actions nor the teammates' policies. Instead, it has a preconceived world model for each possible task, that it utilizes to predict the behaviour of the teammates. This PLASTIC method is similar to the PLASTIC-Policy presented in section 2.4, but without



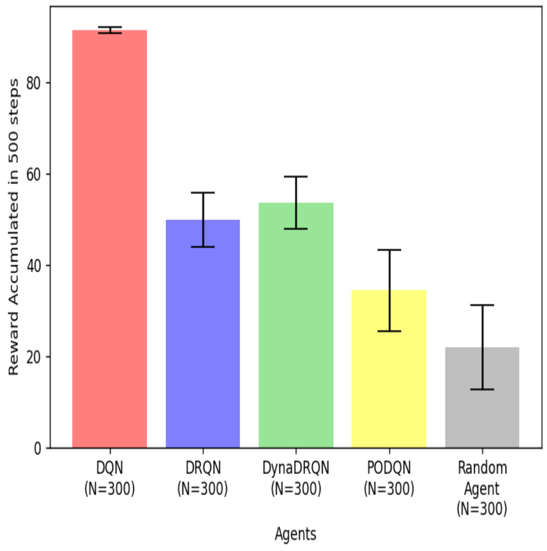
(a) Task 0



(b) Task 1



(c) Task 2



(d) Task 3

Figure 5.11: Non-ad hoc baselines and DynaDRQN agent's accumulated reward

access to the teammates' actions. Rather, it uses the world model to compute the possible observation of teammates' position gathered from environment, based on the current observation and action taken, to later compare it to the real next observation and update the set of beliefs for each possible task.

In fig. 5.14 (a), we display the results obtained by the ad hoc upper bound baseline for task 1, which utilizes the PLASTIC version with access to the teammate's actions, in comparison with the non-ad hoc counter-part. From these we can conclude that an ad hoc agent using this PLASTIC approach with full

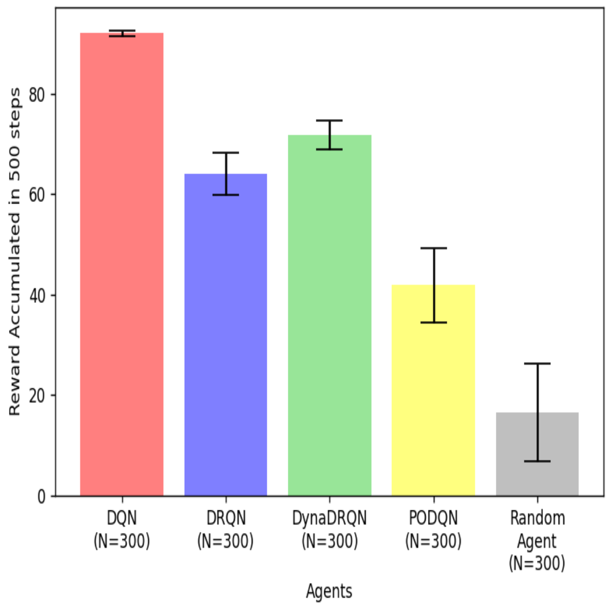


Figure 5.12: Non-ad hoc baselines and DynaDRQN agent's accumulated reward for all tasks combined

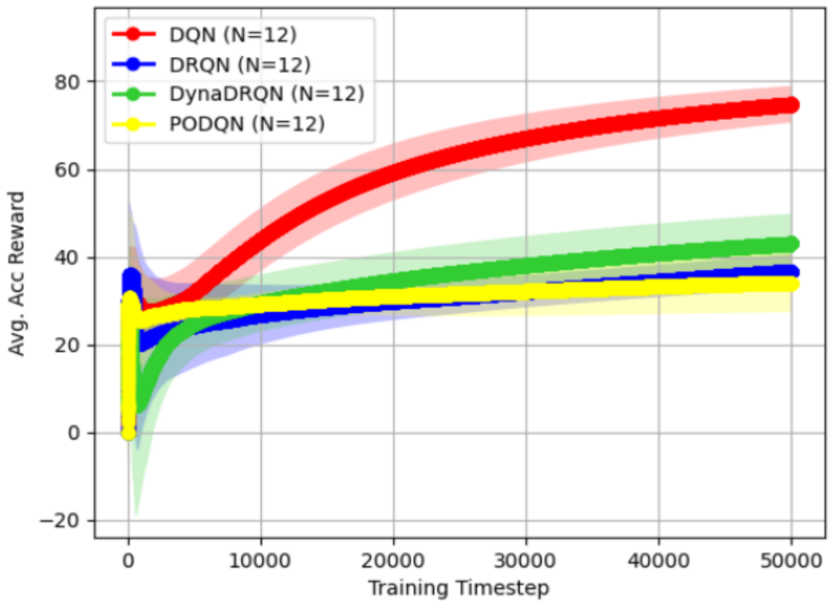
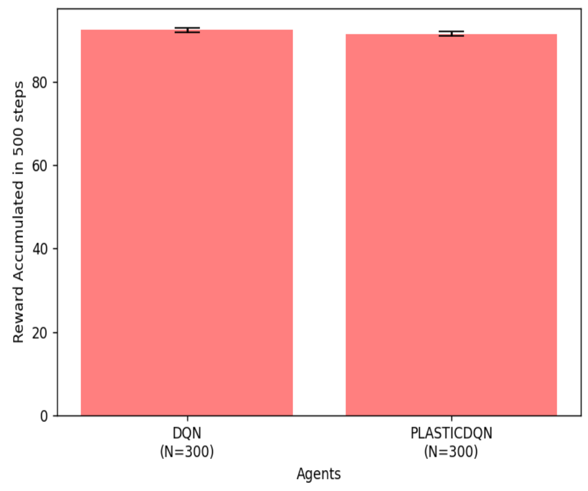
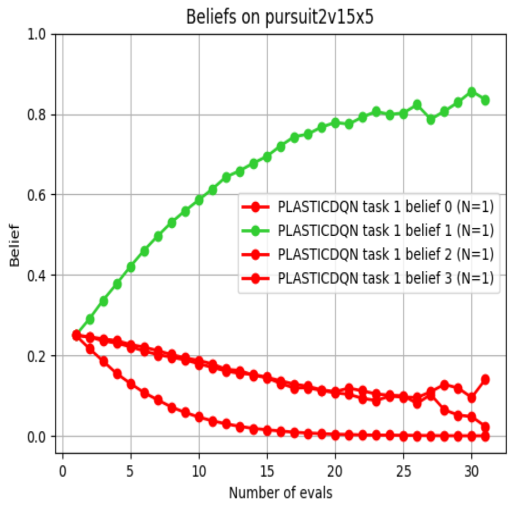


Figure 5.13: Accumulated Reward for the Learning Process of all the non-ad hoc baseline agents and the DynaDRQN agent

visibility can achieve results just as good as if it was working in a non-ad hoc setting. The beliefs' update progression in fig. 5.14 (b) also showcase that the PLASTIC-DQN agent baseline is able to identify the target task under a short number of steps in the environment. In the mentioned figure the green plot line represents the belief progression of the target task. The belief update is calculated using equation (2.14) from section 2.4, with an η of 0.6. When choosing the η value, we had to find a certain equilibrium. A



(a) Accumulated Reward for the DQN and PLASTIC-DQN baselines



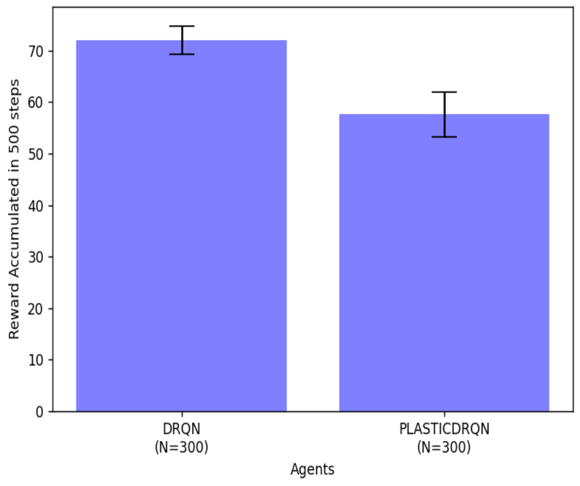
(b) Beliefs' update progression for each task

Figure 5.14: PLASTIC-DQN results

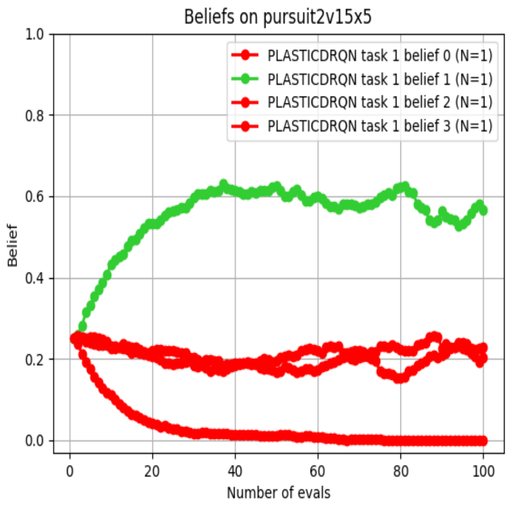
low value guarantees that a wrong update would not ruin the belief update progression, while leading to a slower update that could reduce the accumulated reward on the environment. On the other hand, a higher value may lead the agent to identify the target task faster, but it becomes more susceptible to wrong updates.

However, in fig. 5.15 (a), it is possible to observe that in partial observable environments the ad hoc agent's results are much worse. We show a comparison between the results obtained from a DRQN and from a DRQN in an ad hoc teamwork environment. The results obtained by the ad hoc agent, PLASTIC-DRQN, are in fact not as good as the non-ad hoc ones, since it takes some time for the ad hoc agent to correctly identify the task. This means that it will eventually require more steps to catch the prey. The results' decline is due to the fact that under partial observability the set of policies for the action of the teammates that the baselines have access is not able to predict their behaviour correctly, because it based on the full visibility of said teammates. Nevertheless, these are still very good results for partial observable agent that ends up identifying the target task under a short number of steps in the environment, as it can be seen in fig. 5.15 (b).

Despite attaining good results, the PLASTIC version that has access to the teammate's actions can be considered to breach the idea of ad hoc teamwork under partial observability. This is where our approach presents a novel contribution to the ad hoc teamwork challenge. Through the usage of a world model it is **capable of identifying the target task** by solely accessing the observation of the teammates' location gathered from the environment. By combining the first component of our contribution with the



(a) Accumulated Reward for the DRQN and PLASTIC-DRQN baselines

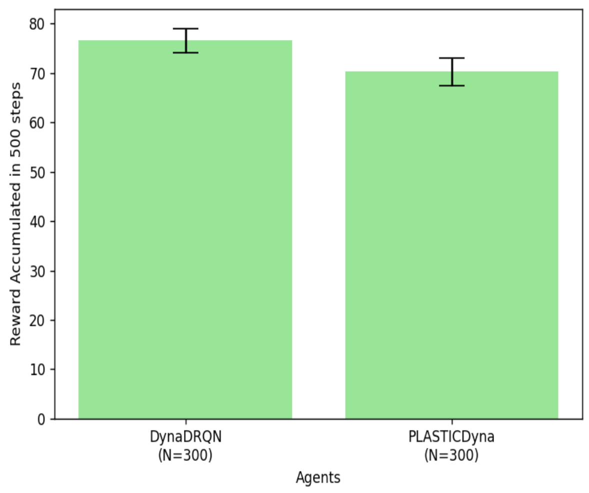


(b) Beliefs' update progression for each task

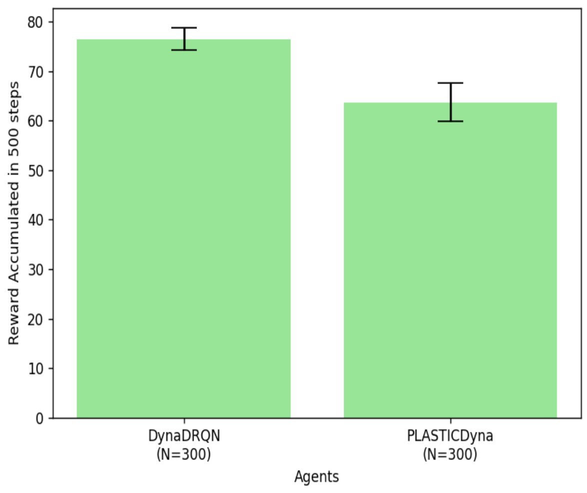
Figure 5.15: PLASTIC-DRQN results

novel ad hoc component, we obtain a **sample efficient** agent **capable of achieving good results under partial observability**, as depicted in fig. 5.16 (a), and, at the same time able to identify the target task under a short number of steps (fig. 5.17). In fig. 5.16 (a) we showcase the results of the PLASTIC-Dyna using the PLASTIC version employed by the baselines, which has access to the teammates' actions and their respective policies. When compared with our own model-based version, we notice that our contribution is able to achieve a better performance in a partial observable environment. Even though the former has access to the teammates' actions, the policies that it has access to assume that the environment is entirely visible. Therefore the results obtained using this method are bound to perform worse than an agent containing a well-trained world model for each possible task under partial observability.

Finally, we compare the PLASTIC-Dyna with all the ad hoc baselines. The results presented in fig. 5.18, reveal that our novel approach is able to correctly identify the target task, better than the partial observable baselines that have access to teammate's action, while maintaining the higher accumulated reward from the DynaDRQN component. The PLASTIC-DQN with full visibility is the only agent that is able to maintain the same results' quality as its non-ad hoc counter-part. When it comes to our contribution, we notice that for some tasks, even though the accumulated rewards decline, it preserves the overall reward difference between the other partial observable baselines. Although in some cases it obtains a bigger difference gap between those baselines, such as in tasks 1 and 3 (fig. 5.18 (b) and fig. 5.18 (d) respectively)



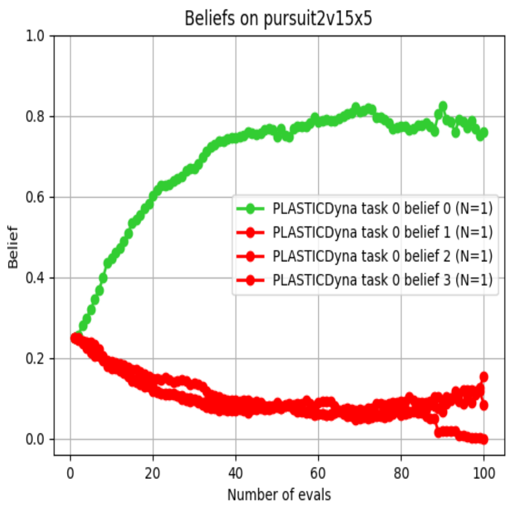
(a) Without access to the teammates' actions



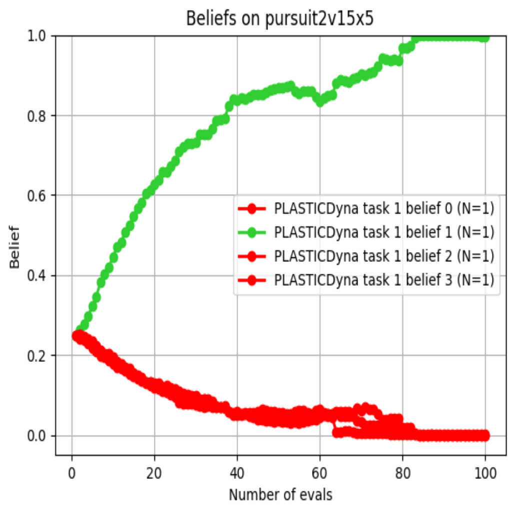
(b) With access to the teammates' actions

Figure 5.16: Accumulated Reward for the DynaDRQN and PLASTIC-Dyna agents

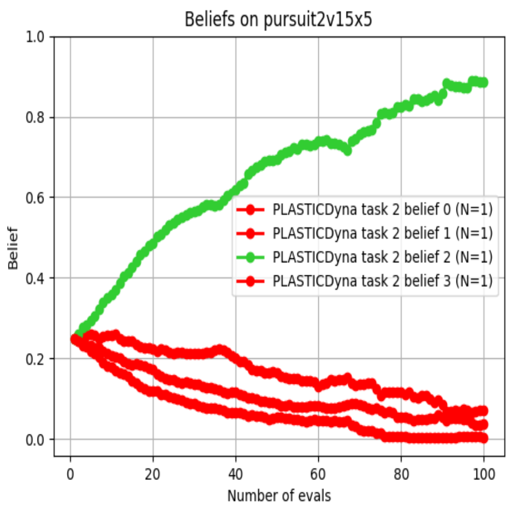
In fig. 5.19 we display the combined average accumulated reward obtained by all the ad hoc agents for all the tasks. It is possible to observe that aside from the full observable PLASTIC-DQN, our approach was the only one able to maintain a result nearly as similar as the one obtained in the non-ad hoc setting. The figure also highlights the increased difference between the PLASTIC-Dyna performance when compared to all the other partial observable baselines. Hence, we conclude that our novel approach, the PLASTIC-Dyna, is capable of **identifying the target task under partial observability**, without access to the teammates' actions. This features make our contribution suitable to be deployed in partial observable and sample scarce environments, where the behaviour of the teammates and task's goals are unknown to our agent.



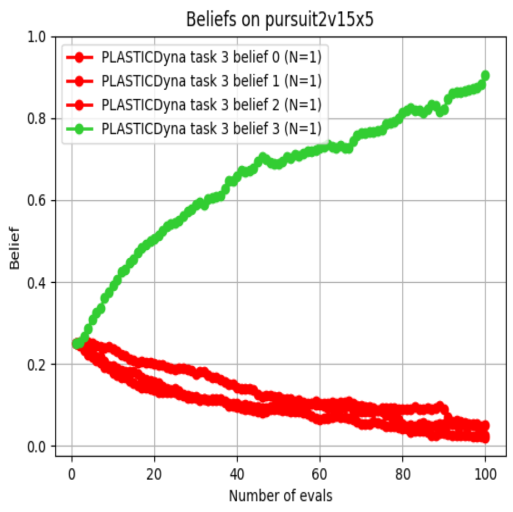
(a) Task 0



(b) Task 1

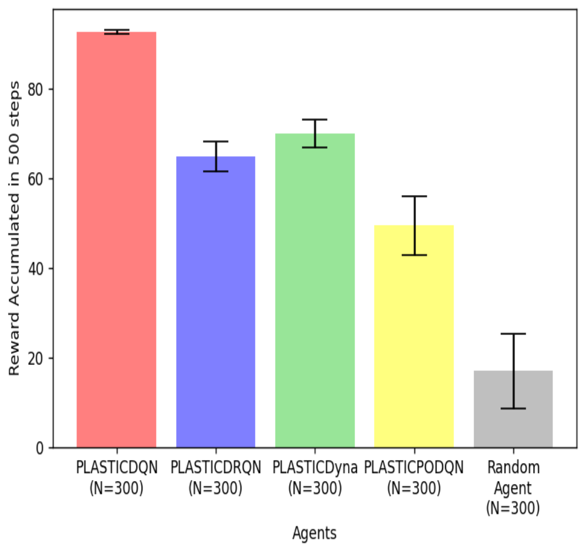


(c) Task 2

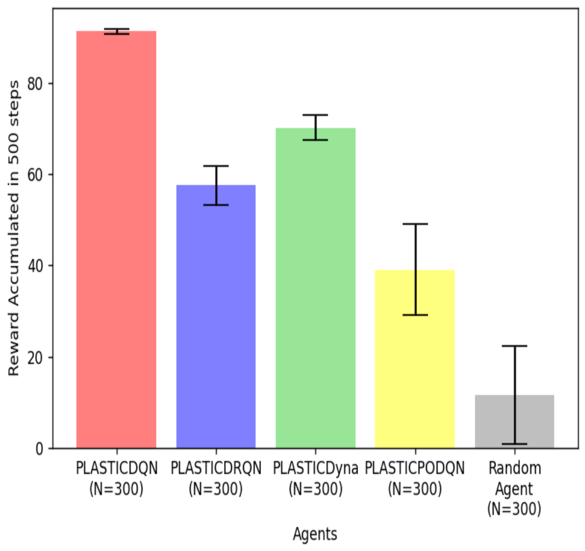


(d) Task 3

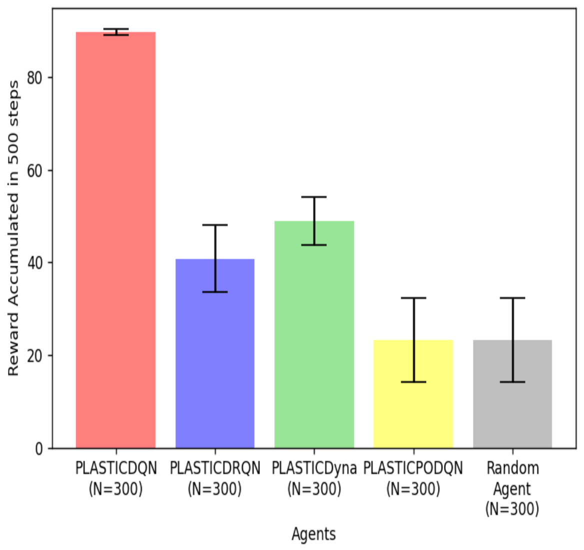
Figure 5.17: Beliefs' update progression for each task



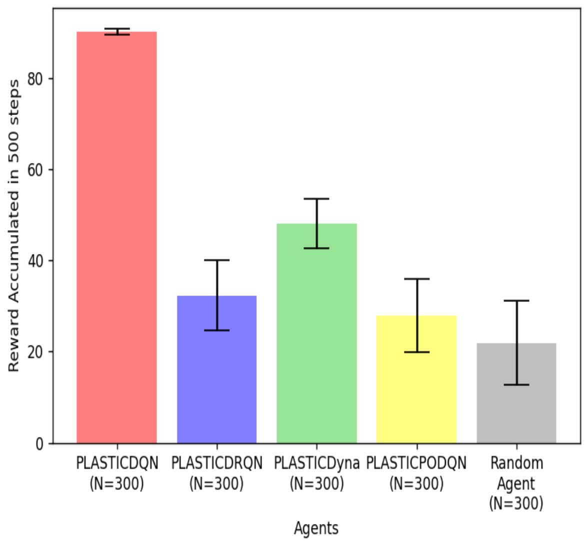
(a) Task 0



(b) Task 1



(c) Task 2



(d) Task 3

Figure 5.18: Ad hoc baselines and PLASTIC-Dyna agent's accumulated reward for each possible task

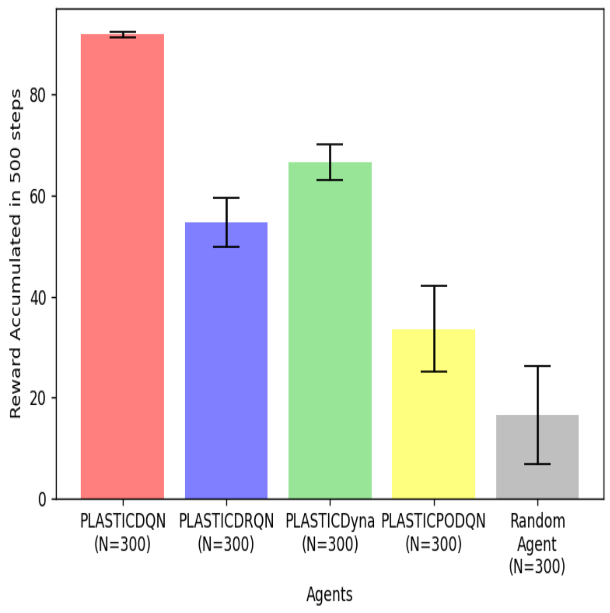


Figure 5.19: Ad hoc baselines and PLASTIC-Dyna agent's accumulated reward for all tasks combined

6 Conclusion

Contents

6.1 Conclusions

In this work, we confronted the challenge of ad hoc teamwork in a partial observable environment. We developed a deep model-based RL approach that was able to achieve a strong and sample efficient performance in a partial observable ad hoc setting (where the teammates' behaviours are unknown to the ad hoc agent), being able to correctly identify the target task in a short number of steps.

Through our experiments, we noticed that to attain good results under partial observability, it is necessary to keep track of the history of previous information, which can be done by employing a recurrent layer. The type of RNN (LSTM or GRU) and sequence data length depend on the complexity of the problem. Due to the lack of computational power, we utilized a low complex environment, such as the pursuit domain in 5 x 5 grid, and also limited the number of interactions with the environment to 50k steps. This limitation was applied to evaluate the sample efficiency of our contribution.

To reach sample efficiency, we created a world model of the environment using supervised learning. This model is able to determine the next observation of the agent gathered from the environment, the reward and the terminal state, based on the input observation and action taken. We combined this world model with a DRQN and created the first component of our contribution, the DynaDRQN. That proved to be more sample efficient than the model-free approach since it was able to generate simulated experiences from the world model while also gathering experiences from the real world.

The second component of our contribution and what we consider to be the most notable one in terms of novelty on the field of ad hoc multi-agent systems under partial observability, is the ad hoc teamwork component. We merged the well-known PLASTIC algorithm with a variation of our initial world model to create an ad hoc architecture capable of identifying the target task based on the observed position of the agent's teammates. This component was integrated with the DynaDRQN to form the PLASTIC-Dyna, an ad hoc deep model-based reinforcement learning agent with the ability to identify the target task better than other ad hoc methods under partial observability.

With this thesis we answer the question, ”Is it possible to create a data efficient reinforcement learning approach that is able to correctly identify the target task under partial observability?”, with our very own contribution, the PLASTIC-Dyna.

6.2 System Limitations and Future Work

Despite the good results attained by our novel contribution to the field of ad hoc teamwork under partial observability, there are still some improvements to be done and options to explore in future works.

The first possible option to expand on this work would be to increase the complexity of the environment. As it was mentioned, the pursuit domain is in itself a fairly simple domain, however it is possible to increase its complexity. This can be done either through grid size expansion or through a refinement of the observability of the surrounding environment. The expansion of the grid can be used to highlight even more the differences between the PLASTIC-Dyna agent and the baselines, at the the cost of requiring more timesteps and computational power to achieve decent results. Another alternative to increase the complexity of the environment is to vary the partial observability, such as increasing or decreasing the noise caused on the observations gathered by the agent or having a different field of vision for the agent.

In future work, it can also be interesting to try different variations of teammates, by changing their behaviour, number or visibility field. In this thesis, we showcased results for an agent acting alongside a greedy agent. Hence in other works it could be a good idea to experiment with a random teammate or a teammate that acts according to the agent's position. Many other works in the literature have also worked with more than one teammate, which increases the complexity of the problem, especially if every teammate has a different behaviour. Another interesting approach to the multi-agent pursuit domain would be to assign a partial observable field to the teammates. By having both agents utilize a DynaDRQN to learn to capture the prey, they would need to learn at the same time and from each other.

The world model is a crucial part of the planning component, therefore improving it can lead to even better performances and data efficiency. One way to enhance the world model would be to add a VAE, which would allow to create a generative model that is able to learn a more simplified version of the environment. The simplified representation can be useful to learn a more complex environment, while the variational feature can be utilized to generate new unseen samples of the world. Another way to develop the planning component would be to create a controller that could be used to predict the best course of actions to maximize the reward based on the observations and next observations gathered from the world model. This controller combined with our contribution, the DynaDRQN, would possibly lead to better data efficiency and faster convergence during the learning process.

One possible extension of the world model would be to use it as a simulated/dream environment where the agent could learn before being deployed in the real world. Nevertheless, this requires a strong generative world model capable of recreating a ”dream”, as close as possible to the real world environment. To leverage this process, some works in the literature utilize a cycle that sees the agent learning in the ”dream” environment and gathering real-world samples to use afterwards to improve the world model for the agent to keep on learning.

The ad hoc component could also benefit from some future work. Even though the PLASTIC method employed in our novel approach, the PLASTIC-Dyna, is able to correctly identify the target task under partial observability without having access to the behaviour of the teammates nor their actions, it is still constrained by the inability to adapt to teammates/situations never seen before. Hence, it would be relevant in future works to explore this side of ad hoc teamwork since in the real world, we might sometimes be subjected to solve problems that we have never dealt with before, despite being equipped with the knowledge to do it, from previous similar situations.

Bibliography

[1] J. Andreu-Perez, F. Deligianni, D. Ravi, and G. Yang, “Artificial intelligence and robotics,” *arXiv preprint arXiv:1803.10813*, 2018.

[2] M. N. Huhns and L. M. Stephens, *Multiagent systems and societies of agents*. Cambridge, MA: MIT press, 1999, vol. 1, pp. 79–114.

[3] V. Julian and V. Botti, “Multi-agent systems,” *Applied Sciences*, vol. 9, 2019.

[4] P. Stone, G. Kaminka, S. Kraus, and J. Rosenschein, “Ad hoc autonomous agent teams: Collaboration without pre-coordination,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, 2010, pp. 1504–1509.

[5] S. Barrett and P. Stone, “Cooperating with unknown teammates in complex domains: A robot soccer case study of ad hoc teamwork,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.

[6] D. Chakraborty and P. Stone, “Cooperating with a Markovian ad hoc teammate,” in *Proc 12th Int. Conf. Autonomous Agents and Multiagent Systems*, 2013, pp. 1085–1092.

[7] F. S. Melo and A. Sardinha, “Ad hoc teamwork by learning teammates’ task,” *Autonomous Agents and Multi-Agent Systems*, vol. 30, pp. 175–219, 2016.

[8] S. Barrett, N. Agmon, N. Hazon, S. Kraus, and P. Stone, “Communicating with unknown teammates.” in *ECAI*, 2014, pp. 45–50.

[9] J. G. Ribeiro, C. Martinho, A. Sardinha, and F. S. Melo, “Assisting unknown teammates in unknown tasks: Ad hoc teamwork under partial observability,” *arXiv preprint arXiv:2201.03538*, 2022.

[10] R. S. Sutton, “Integrated architectures for learning, planning, and reacting based on approximating dynamic programming,” in *Machine learning proceedings 1990*. Elsevier, 1990, pp. 216–224.

[11] M. Hausknecht and P. Stone, “Deep recurrent q-learning for partially observable mdps,” in *2015 aaai fall symposium series*, 2015.

[12] P. Domingos, “A few useful things to know about machine learning,” *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[13] Q. V. Le, “Building high-level features using large scale unsupervised learning,” in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 8595–8598.

[14] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[15] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[16] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

[17] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[19] C. J. C. H. Watkins, “Learning from delayed rewards,” 1989.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[21] T. Lu, D. Schuurmans, and C. Boutilier, “Non-delusional q-learning and value-iteration,” *Advances in neural information processing systems*, vol. 31, 2018.

[22] S. Bengio, Y. Bengio, J. Cloutier, and J. Gescei, “On the optimization of a synaptic learning rule,” in *Optimality in Biological and Artificial Networks?* Routledge, 2013, pp. 281–303.

[23] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.

[25] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv preprint arXiv:1412.3555*, 2014.

[26] E. S. Low, P. Ong, and K. C. Cheah, “Solving the optimal path planning of a mobile robot using improved q-learning,” *Robotics and Autonomous Systems*, vol. 115, pp. 143–161, 2019.

[27] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[28] J. Shin and J. H. Lee, “Mdp formulation and solution algorithms for inventory management with multiple suppliers and supply and demand uncertainty,” in *Computer Aided Chemical Engineering*. Elsevier, 2015, vol. 37, pp. 1907–1912.

[29] L. Panait and S. Luke, “Cooperative multi-agent learning: The state of the art,” *Autonomous agents and multi-agent systems*, vol. 11, pp. 387–434, 2005.

[30] L. Busoniu, R. Babuska, and B. De Schutter, “A comprehensive survey of multiagent reinforcement learning,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, 2008.

[31] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, “Multiagent cooperation and competition with deep reinforcement learning,” *PloS one*, vol. 12, no. 4, p. e0172395, 2017.

[32] Y. Yu, “Towards sample efficient reinforcement learning.” in *IJCAI*, 2018, pp. 5739–5743.

[33] D. Ha and J. Schmidhuber, “World models,” *arXiv preprint arXiv:1803.10122*, 2018.

[34] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine *et al.*, “Model-based reinforcement learning for atari,” *arXiv preprint arXiv:1903.00374*, 2019.

[35] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, “Mastering atari, go, chess and shogi by planning with a learned model,” *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[36] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, “Mastering atari with discrete world models,” *arXiv preprint arXiv:2010.02193*, 2020.

[37] P. Stone and S. Kraus, “To teach or not to teach?: decision making under uncertainty in ad hoc teams.” in *AAMAS*. Citeseer, 2010, pp. 117–124.

[38] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[39] N. Agmon and P. Stone, “Leading ad hoc agents in joint action settings with multiple teammates.” in *AAMAS*, 2012, pp. 341–348.

[40] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.