TÉCNICO LISBOA

Ad hoc teamwork with unknown task model and teammate behavior

Gonçalo Alfredo dos Santos Rodrigues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

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

Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. Francisco António Chaves Saraiva de Melo
Member of the Committee: Prof. Pedro Manuel Urbano de Almeida Lima

OCTOBER 2018

Acknowledgments

I want to thank both my supervisors, Francisco Melo and Alberto Sardinha, for steering me in the right direction, helping me whenever I had questions and providing adequate resources when I needed. You were always there to answer my questions and provide technical guidance and also, of course, advice when it comes to writing.

I also want to thank my family for supporting me throughout this process, always assuring me that I could finish this thesis and providing everything I needed to do so.

Finally, I want to thank my friends, for sharing their thesis stories and relating to my problems, for supporting me and keeping me sane. Without you, I would have gone crazy trying to write all of this.

Abstract

This thesis addresses the problem of ad hoc teamwork. Ad hoc teamwork consists of creating an agent that can cooperate with a team without pre-coordination. We focus on settings where the task and world model are unknown, and the agent must learn this task, learn its teammates' policies and adapt to them. We propose a new approach that combines model-based reinforcement learning with Monte Carlo tree search to enable an ad hoc agent to learn the underlying task and coordinate with its teammates. We model the underlying environenment and the teammates' behaviors using deep neural networks that are learned as the ad hoc agent explores. By using a model-based approach, we can achieve a good performance with low sample complexity. The solution is demonstrated in the well-established pursuit domain and our results show that it is competitive against state-of-the-art approaches that rely on the perfect model of the task and known behavior of the teammates.

Keywords

Ad hoc teamwork; Multiagent reinforcement learning; Deep learning; Model-based reinforcement learning; Artificial intelligence

Resumo

Esta tese aborda o problema de ad hoc teamwork. Ad hoc teamwork consiste em criar um agente que coopera com uma equipa sem pre-coordenacao. Nos focamo-nos nos cenários em que o modelo da tarefa e do mundo são desconhecidos, e o agente tem que aprender a tarefa, aprender o comportamento dos colegas de equipa e adaptar-se a eles. Nos propomos uma nova solução que combina model-based reinforcement learning com Monte Carlo tree search para que o agente ad hoc aprenda a tarefa e consiga coordenar com o resto da equipa. Nos modelamos o ambiente e o comportamento da equipa usando deep neural networks que são aprendidas a medida que o agente explora. Usando esta abordagem model-based, conseguimos atingir uma boa performance sem usar muitos samples. A solução e demonstrada no conhecido pursuit domain e os nossos resultados mostram que e competitiva com soluções state-of-the-art que assumem perfeito conhecimento da tarefa e do comportamento da equipa.

Palavras Chave

Ad hoc teamwork; Multiagent reinforcement learning; Deep learning; Model-based reinforcement learning; Artificial intelligence

Contents

List of Figures

List of Tables

Acronyms

MDP	Markov Decision Process
MMDP	Multiagent Markov Decision Process
POMDP	Partially Observable Markov Decision Process
RL	Reinforcement Learning
MCTS	Monte Carlo Tree Search
HFO	Half Field Offense
FQI	Fitted Q-iteration
MAB	Multi-armed bandits
SGD	Stochastic Gradient Descent
UCT	Upper Confidence Bound for Trees
DQN	Deep Q-Networks

1

Introduction

Contents

1.1 Motivation

As agents are incorporated in the world, the more we need them to work together for better performance. In the cooperative multiagent setting, two or more agents act in the environment with the goal of maximizing their joint reward. Solutions for this setting either incorporate explicit communication between the agents or the same algorithm is shared among the agents and so they are implicitly coordinated.

However, there is an increasing need for agents that can cooperate with their team without communicating or without pre-coordination. Consider these 2 scenarios:

1. Developers from different countries create agents independently to rescue people from a natural catastrophe, without defining a communication protocol or coordinating their approaches.

2. Agents have been deployed in some environment and years later one of them breaks and it must be replaced. Since many years have passed there is an opportunity to create an agent more powerful than the existing ones but we still wish to keep the old agents.

In both cases we need to create an agent able of cooperating with others that possesses different behavior from them. It must be able to adapt to a wide range of behaviors in an online fashion, possibly behaviors not known *a priori*. Agents can have different capabilities, different models of the world, different roles and different ways of communicating and yet they need to be able to cooperate. We are facing an ad hoc teamwork problem.

In ad hoc teamwork, the goal is to design an agent that can learn its teammates behaviors and characteristics online and adapt to them in order to reach a certain shared objective. This problem was introduced by Stone et. al. [1] and some work has been done that tackles some specific constrained instances of the problem. Some of the more complex domains used as a testbed for ad hoc teamwork approaches include the pursuit domain (where 4 agents chase a prey) and Half Field Offense (HFO) (a simulated soccer game played only in one half of the field). Most theoretical solutions propose algorithms for which the time of execution grows exponentially with the number of agents. Some of the empirical approaches either have strong assumptions – usually known task and team behavior – or they are model free, leading to a high sample complexity. In this thesis we address the following research question: How can we tackle the ad hoc teamwork problem without imposing many restrictions on the other agents and without incurring an excessive sample complexity?

1.2 Hypothesis

Similarly to previous work [2], our ad hoc agent must cooperate with a team of agents in the pursuit domain [3], and must complete the goal in different world sizes.

We also expect that our solution, while having less knowledge regarding the domain and teammates' policies, still achieves comparable performance to state-of-the-art solutions that incorporate such knowledge, notably PLASTIC-Model [4].

Finally, we expect that our solution can converge faster to a good performance when compared to model-free approaches, such as PLASTIC-Policy [5] or Deep Q-networks [6].

1.3 Contributions

The main contribution is a novel ad hoc agent architecture addressing the ad hoc teamwork problem in full, including task identification, teammate identification and planning. The approach taken comprises three main modules, each addressing one of the aforementioned challenges:

- A model-based reinforcement learning to learn a model of the environment and task. The model uses a deep neural network that predicts, given the current state and the actions of all agents in the environment (ad hoc and teammates), the resulting state.
- A fictitious-play-like approach to model the teammates, constructing a predictive model that, given an input state, returns a distribution over the teammates' actions.
- Finally, using the environment and teammate models, use of Monte Carlo tree search to select the action of the ad hoc agent.

The approach is illustrated in the well-established pursuit domain, providing a comparative analysis between our own performance and that of state-of-the-art methods from the literature.

2

Background

Contents

Most related work in the ad hoc teamwork field looks at the ad hoc teamwork problem from one of 3 different perspectives – as Multi-armed bandits (MAB), as a fully cooperative normal form game or as a Markov Decision Process (MDP). In this section we introduce these 3 frameworks and some of their properties. Additionally we introduce Monte Carlo Tree Search (MCTS) and neural networks, as they are used in this work.

2.1 Multi-armed bandits

MAB are usually used to study exploration vs exploitation problems. There are k arms, each with an unknown payoff distribution (usually assumed to be stationary). An agent can choose one of the arms and receive a reward drawn from the corresponding distribution, but he is only be able to see the reward of the arm he picked and none other. It either has a finite number of rounds (finite horizon) or an infinite discounted number of rounds, where future rounds are multiplied by a discount factor. The goal is to choose a sequence of actions that maximize the total reward. If the agent always picks the same action, he might be missing out as he cannot know whether he is picking the optimal one. But if it keeps exploring and changing arms, he is giving up some of the reward as he is not always choosing the optimal arm.

Formally, we measure the performance of different algorithms using regret R. It can be computed by finding the difference between the sum of rewards of the optimal arm in hindsight and the sum of actually collected rewards. Denoting X_{it} the reward that arm i gave or would give if picked at time t, and a_t the arm chosen at time t, the expected regret after T plays is defined as in (2.1). One goal when solving MAB problems is to minimize expected regret.

$$\mathbb{E}[R_T] = \mathbb{E}\left[\max_{i=0,1,\cdots,k} \sum_{t=0}^{T} X_{it} - \sum_{t=0}^{T} X_{a_t t}\right] \tag{2.1}$$

MAB bandits can be further extended to the multi-agent scenario in which all pick their actions at the same time or take turns. If the agent is allowed to see other agent's actions, it can use such information to update its estimative of the average reward of each arm.

2.2 Fully cooperative normal form game

Fully cooperative normal form games can be fully defined by a reward matrix where each joint action is associated with a reward. All agents choose an action at the same time, and they all get the same reward that is defined in the matrix. We only consider fixed reward (not stochastic) and a repeated

scenario where agents play the same game for a finite or infinite number of rounds. One of the problems that this framework can be used for is cooperation. Consider the scenario represented by the matrix below:

	b_0	b_1
a_0	1	0
a_1	0	1

Two agents play a game where they get 1 point if they pick the same action, either (a_0, b_0) or (a_1, b_1), but get 0 points if they play differently from each other, (a_0, b_1) or (a_1, b_0). In this case the optimal action of one agent depends on the other agent's actions and when no communication is available this can be a pretty hard problem. One possible way to solve this is to try to predict their behavior, for example using the history of actions. If the agent can predict their behavior then the problem is completely solved. This framework will be vastly used as the ad hoc teamwork problem includes this coordination without communication problem.

A simple strategy that has been studied frequently is the bounded-memory best-response strategy. An agent following such strategy remembers the last mem (where mem is an integer ≥ 1) joint actions and, for each other player, computes the distribution of their actions. It then assumes that each player chooses randomly from their action distribution. The strategy is to choose the action that leads to the best expected value. When memory is unbounded, this algorithm is also known as fictitious play.

MAB and normal form games serve different purposes – MAB are usually concerned with minimizing regret while normal form games are usually looking for nash equilibria.

2.3 Markov decision processes (MDP)

MDPs are defined as a 5-tuple $(\mathcal{X}, \mathcal{A}, \{\mathbf{P}_a\}, r, \gamma)$ where $\mathcal{X}$ is a set of states, $\mathcal{A}$ is a set of actions available to the agent, $\mathbf{P}_a(s'|s)$ is probability of next state being s' when coming from state s and using action a, $r(s, a)$ is the reward received for selecting action a in state s and γ is the discount factor. The main difference behind MDPs and the other 2 frameworks is that MDPs have multiple states in which the agent transitions over time. In an MDP, the environment has a state and the rewards depend on this state and therefore the rewards are no longer stationary. At each timestep, the agent transitions to a new state, depending on the action it chose given by the function P and gets a reward that depends on the current state and the action chosen given by the function R. This framework can model a huge variety of problems (i.e. robot soccer) and is much more general than the previous ones. S, A and γ are always assumed to be known but P and R can be unknown. In the case that they are unknown we are facing a Reinforcement Learning (RL) problem.

If everything is known, the optimal action to choose at each timestep can be computed by iterating

the following equation until the value converges (value iteration).

$$Q^{t+1}(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) \max_{a' \in A} Q^t(s',a') \tag{2.2}$$

It can be proven that the Q function converges to a unique solution and the greedy policy with respect to this Q function (for each state s, choose the action a that maximizes $Q(s,a)$) is the optimal policy. Each step of value-iteration has complexity of $O(|A||S|^2)$.

If however we do not know P or R, there are different approaches that can be used - model-free or model-based. Model-free approaches attempt to get the optimal policy without computing P or R, while model-based approaches try to infer P and R from the observations and then compute the optimal policy from that. One example of model-free approach is Q-learning, that updates Q-values as $(state, action)$ pairs are visited. For each transition from s_t to s_{t+1} using action a_t receiving reward r_t, the update is done as in (2.3), where α represents a learning rate.

$$Q^{t+1}(s_t,a_t) = (1-\alpha)Q^t(s_t,a_t) + \alpha(r_t + \gamma \max_{a \in A} Q^t(s_{t+1},a)) \tag{2.3}$$

It can be proven that Q-learning converges to the optimal policy if every $(state, action)$ pair is visited infinitely many times.

Many times the number of states can be quite big and these solutions are no longer practical as they depend at least quadratically in the number of states. Imagine a grid world with a $n \times n$ grid and k agents positioned in it. The state is fully represented by the positions of all agents in the grid and must have a x and y coordinate for each agent. So it corresponds to having $2k$ variables each with a value from 1 to n. To represent all possible values, we'd need n^{2k} states which might be impractical for a dozen of agents. Another problem with the above techniques arises when we are dealing with continuous states/actions as they need to keep a value of each state-action pair. One approach would be to discretize states/actions, but this can lead to huge number of states to get a good enough precision.

Another model-free algorithm, called Fitted Q-iteration, attempts to solve these problems. In this algorithm, one collects a dataset of transitions offline and then uses a supervised learning technique to model the Q-value function for each action given the state, where the error function can be, for example, $(Q^t(s_t,a_t) - (r_t + \gamma \max_{a \in A} Q^t(s_{t+1},a)))^2$. The input for the supervised learning algorithm is a set of features that fully describes the state. This way, the similarity between states is leveraged as states with similar features should have similar values which can be naturally represented in supervised learning methods. For the example given before, only $2k$ features – x, y coordinates for each agent – would be needed to represent the state.

MDPs can be extended to multi agents in a fully cooperative scenario simply by considering the action space as all the joint actions possible. If all agents run the exact same value iteration algorithm

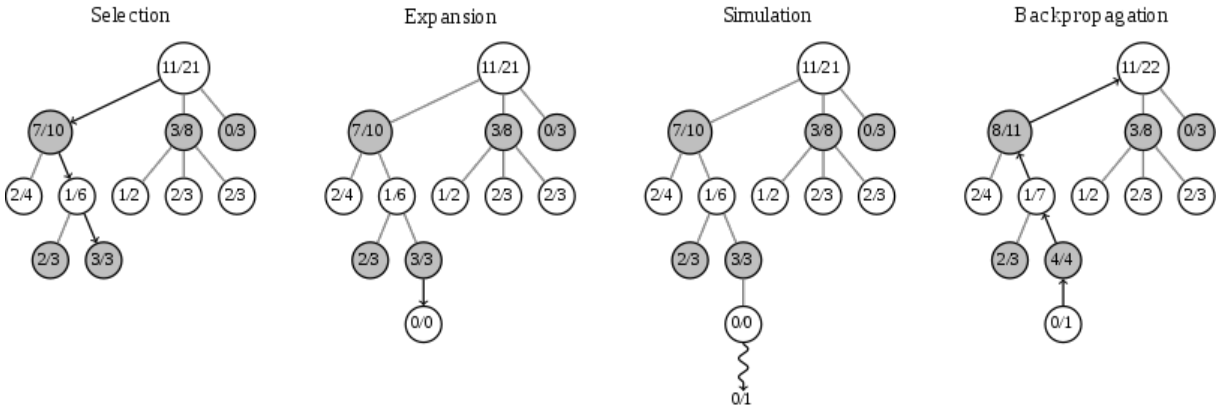

Figure 2.1: Monte carlo tree search algorithm in four steps

breaking ties the same way, then the value iteration approach still works as each agent simply picks its action from the optimal joint policy. Of course that if we cannot control all agent's actions (decentralized approach) then the optimal policy is no longer the one given by value iteration. This is the case for the ad hoc teamwork problem – we are not able to control all agents.

Multiagent Markov Decision Processes (MMDPs) are more general than multi armed bandits and fully cooperative normal form games described previously. In fact, a fully cooperative normal form game can be seen as a special case of a MMDP with a single state where the reward function represents the reward matrix.

2.4 Monte carlo tree search (MCTS)

To solve MDPs, you can use Value Iteration as explained in Section 2.3, but this can be very slow when there is a big amount of state-action pairs. Rather than solve it optimally, MCTS is able to solve MDPs approximately, reducing the time it takes to finish.

MCTS is an algorithm composed of four stages: selection, expansion, simulation and backpropagation. It builds a game tree as time goes on, where nodes represent either states or actions. It starts with a root node, representing the initial state. A representation of these steps can be seen in Figure 2.1.

The first step is selection. In this step, we go through the tree, starting at the root node, selecting a child at each iteration until we hit a leaf node. The children are not necessarily selected at random, but rather using a tree policy.

The second step is expansion. In this step, we generate a child of the leaf node that was selected in the previous step. Children of a state node are simply all actions available to the agent at that state. Children of an action node are all possible states resulting from the previous state and action chosen.

The third step is simulation. Starting from the child generated in the previous step, we simulate a

game starting from that state until the game ends or a max depth is reached. In this simulation, the agent usually chooses according to a default policy, usually just random actions. Nodes resulting from this simulation are not kept in the tree after the simulation ends. The score achieved is saved in the child generated as an estimate of its value.

The final step is backpropagation. In this step we backpropagate the reward that the agent got in the simulated game, so that each node, starting form the root to the child generated in the second step, has an estimate of the reward that is achieved when starting from its state.

After a given number of iterations, a tree has been generated and there is an estimate of each action's value. To pick the best action, the agent simply chooses the action linked to the root node that has the best reward estimate.

2.4.1 Upper Confidence Bound for Trees (UCT)

UCT is an implementation of MCTS with a specific tree policy and default policy. When choosing which action to explore, it balances exploration of unexplored nodes and exploitation of nodes with good rewards. It does this by casting the problem of selecting a node as a MAB problem and using upper confidence bounds to solve it.

In UCT, at a state node, the action i is selected if it maximizes the following value:

$$\bar{r}_i + c\sqrt{\frac{\log N}{n_i}} \tag{2.4}$$

where $\bar{r}_i$ is the estimate of the value of the action i, N is the number of visits to the parent node state, n_i is the number of times that action has been picked and c is the exploration parameter which depends on the problem and is usually chosen empirically.

The default policy, used for the simulations, is to just pick an action from the available actions uniformly at random.

2.5 Neural networks

Supervised learning is a subset of machine learning that studies algorithms for learning some mapping from input to output given examples. Formally, supervised learning algorithms are given examples, each constituted by input features x and a target output y. Their goal is to learn a mapping from features to target so that when they are given a set of features of a new example, they predict its target.

Neural networks are one framwork for supervised learning, that has been heavily studied in recent years and is state-of-art for many different domains. A neural network can be seen as a direct acyclic graph composed of an input layer, an output layer and H hidden layers. Each layer is composed of many

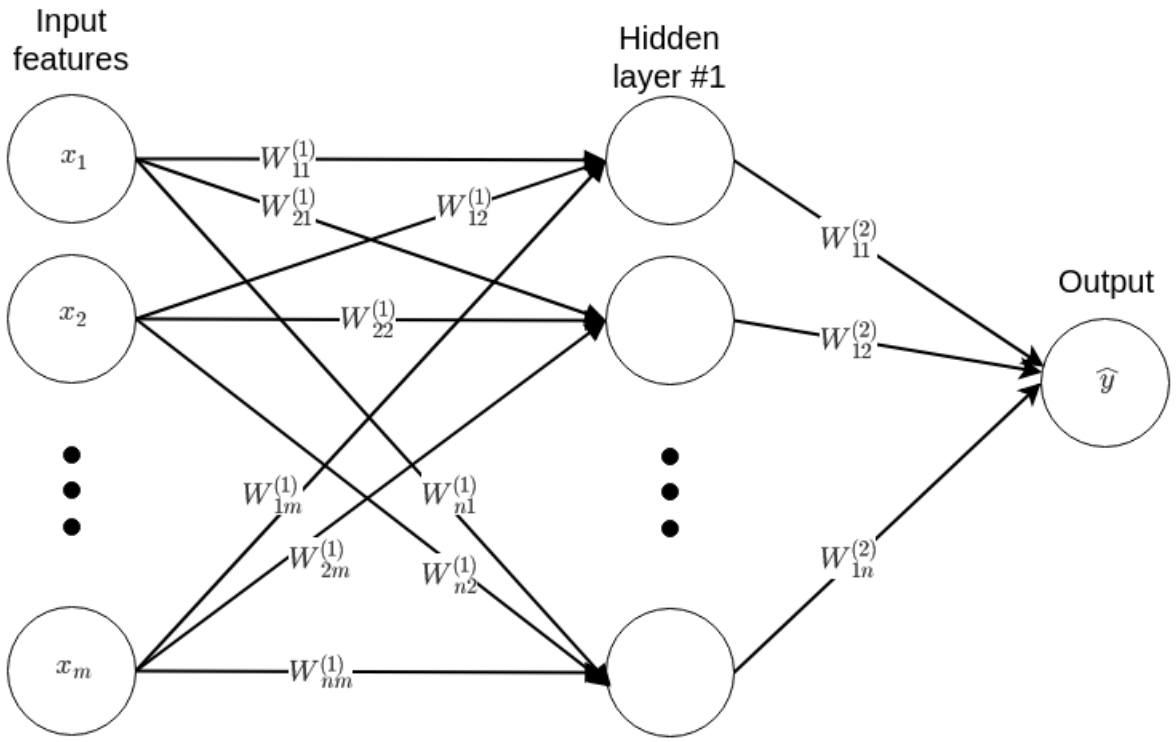

Figure 2.2: Example of a neural network with one hidden layer

nodes and each node is connected to every node of the next layer by a weighted edge. These weights are organized in matrices. The weight that connects the node i from layer $h-1$ to node j in layer h is denoted by $W^{(h)}_{ji}$ An example can be seen in Figure 2.2. Given a set of features $\boldsymbol{x}$ as a column vector, the target prediction is computed by

$$\hat{y} = f(W^{(H)} \cdots f(W^{(2)} f(W^{(1)} \boldsymbol{x}))) \tag{2.5}$$

where f is an activation function. The activation function must be non-linear, or else $\hat{y}$ would just be a linear combination of $\boldsymbol{x}$, which would severely limit the types of function the network could learn. Usually f is chosen to be the sigmoid[1] or the relu[2] functions, as they seem to work in practice. One key characteristic of $\hat{y}$ is that it is differentiable with respect to $\boldsymbol{x}$.

In order to learn, neural networks optimize their weights by minimizing the error between its predictions $\hat{y}$ and the known targets y. This error can be captured by many different functions but usually one of two is chosen depending on the task: mean squared error for regression or cross-entropy for classification. Mean squared error is defined as $\text{MSE}(\hat{y}, y) = ||\hat{y} - y||^2$ while cross-entropy is defined as $\text{CE}(\hat{y}, y) = -y \log(\hat{y})$. As $\hat{y}$ gets closer to the true y, the value of these loss functions gets smaller –

[1] $\sigma(x) = \frac{1}{1+e^{-x}}$
[2] $\text{relu}(x) = \max(0, x)$

and therefore if we minimize the loss functions we should get $\hat{y}$ close to the true y leading to accurate predictions of the target.

Optimization in neural networks is usually done with Stochastic Gradient Descent (SGD) as the loss function is differentiable, making it suitable for derivative based optimization. There are many variations and adaptations of SGD such as those with momentum which yield better results in practice. One commonly used is ADAM [7], which adapts the learning rate for each parameter that needs to be learned in each iteration.

3

Related work

Contents

The general ad hoc teamwork problem was first introduced by Stone et al. [1] as the problem of cooperating with a team of agents without pre-coordination. This definition includes problems such as figuring out teammate's behaviors, environment rewards and planning. Regarding teammate's behaviors, they can either be known, unknown or one of a fixed set of known behaviors.

Most work done so far formalizes the problem as a MAB [8, 9, 10], as a fully cooperative normal-form game [11, 12, 13, 14] or as an MDP [4, 15], with multiple agents where one is on our control (ad hoc) and the others are not (legacy). Some of the work is focused on the teammate identification problem, while others are focused solely on the planning part of the problem. In the continuation, we review existing literature on ad hoc teamwork, organizing it according to which formalization is used to describe the problem.

3.1 Multi-armed bandits

In this setting, there is a finite set of actions that the ad hoc agent can pick, called arms, which are followed by the action(s) of the other agent(s). This is a very limited scenario as the environment does not change and the reward each one gets is simply the sum of each agent's individual reward. The reward distribution can be stationary or changing in time, but all work presented in the following sections assume a stationary distribution.

Stone et al. [8] were the first to formalize the ad hoc scenario in this manner, in which there are only 2 agents, one of them ad hoc, and the other one always chooses the action with the best empirical average reward, with all actions being observable to both sides. The challenge here lies in the fact that the other agent does not know the expected rewards of each action, while the ad hoc one does, leaving opportunity for our agent to communicate by showing which actions are best allowing the legacy agent to see their superior reward. The ad hoc agent then has to be able to balance exploiting the reward of superior actions only available to him and teaching the other agent the rewards of the other arms. They prove that the ad hoc agent should always select either the optimal arm which is not available to the legacy agent or choose the best arm that is available to the legacy agent, in order to teach it. They propose a dynamic programming algorithm running in polynomial time for the optimal decision sequence in finite horizons settings. While this work assumed a finite number of rounds, Barrett and Stone [9] later extend the theoretical results to discounted infinite horizons.

Barrett et al. [10] considered a slightly different scenario, where there is one ad hoc agent and $N-1$ legacy ones and they are allowed limited communication. They can communicate their last selected arm and payoff, the observed average reward for a specific arm and they can suggest others to select a specific arm.

Regarding teammate behavior they consider different scenarios and analyze their results: known

teammate behavior, unknown but one of a finite set of possible behaviors and one of a finite set of possible behaviors but with unknown continuous parameters (in practice this constitutes an infinite set). The problem can be modeled as an MDP and solved in polynomial time or a Partially Observable Markov Decision Process (POMDP) in the unknown continuous parameters case and an approximation can be done in polynomial time. Solving the POMDP is very computationally costly and infeasible in practice, for which they use a Monte Carlo approximation.

These first approaches to the ad hoc teamwork problem solve a very simplified version of the general problem and while they are important foundations for the succeeding solutions they are not practical enough to deploy in real scenarios on their own. It is easy to see that all of these algorithms are exponential in the number of agents.

3.2 Iterated normal form game

In this case there is still a finite set of actions for each agent but they all select their action simultaneously, receiving a joint reward. The environment does not change and the rewards are fixed and shared but are now deterministic.

Brafman and Tennenholtz [16] looked into the ad hoc teamwork problem in a normal form game. They describe a setting with two agents, composed by a student and a teacher. It is not possible to directly control the student, but using the teacher actions we can influence it. They assume the student behavior is known and can either be based on Q-learning or follow some hand-coded policy. They show that the teacher can choose actions to punish or reward the student for its behavior, so the student may learn a better policy. The main difference between this early paper and those described in this thesis is that Brafman and Tennenholtz do not assume a fully cooperative game, and instead focus on non-cooperative games such as prisoner's dilemma.

Stone et al. [17] introduced this formalization for the ad hoc teamwork problem when tackling the following problem: a 2-agent team, one ad hoc and one legacy which follows a ϵ-greedy bounded-memory best-response policy, try to maximize their joint reward. Only the ad hoc agent knows the reward matrix. It might be tempting to make the ad hoc agent simply choose the action which maximizes the joint reward and the legacy agent will eventually choose that one too. However this is not the optimal behavior (even when memory = 1) because if the ad hoc agent moves gradually through the matrix it can get higher rewards until it gets to the optimal joint action. Computing the optimal path of actions to choose can be done with a dynamic programming algorithm shown in [17] when memory=1. If the legacy agent has a longer memory than that, the algorithm proposed is exponential in the memory size.

Later, Agmon and Stone [12] studied this setting for more than 1 legacy agent. With many legacy agents, all of them best-response, the optimal joint action might be unreachable. Sometimes it is impos-

sible to remain in a single joint action, so the goal becomes to find the optimal reachable steady cycle of actions and optimal path to reach that cycle. To solve this, they describe the problem as a graph where each state is a possible joint action and there is an edge for each possible ad hoc agent's action that connects the joint action at time $t-1$ and the joint action at time t with the cost of the joint action at time t. For $memory$ = 1, these edges are easy calculated as we can always fully determine the action of each agent at time t if we know the joint action at time $t-1$. Having this graph, finding the optimal steady cycle corresponds to finding the minimum cycle mean, and we can find the best path leading there by using Dijkstra's algorithm for each node in the cycle. Both these algorithms are polynomial in the nodes and edges, which are polynomial in the action space but exponential in the number of agents. For $memory > 1$, each node must contain information about the last $memory$ joint actions, and therefore there the number of nodes grows exponentially with the memory size. This algorithm also naturally extends for cases where there is more than one ad hoc agent.

In a step towards the more general problem, Chakraborty and Stone [14] relaxed the assumption that the legacy agent had memory-bounded best-response behavior and focused on 2-agent teams where the legacy agent has a Markovian behavior. A set of features is Markovian if their value at time t depends only on their collective value and joint action at time $t-1$. A teammate is said to be Markovian if his action at each step depends only on the current value of a set of Markovian features. The work considers only discrete features. This is a much more general assumption than best-response agents, as the features can contain (summarized) information about the whole joint action history. For example, the memory-bounded best-response agent is Markovian where there are as many features as memory size and feature i is the joint action at time $t-i$. They further assume that these features and how to compute them is known and can be ordered by relevance. To solve the problem, K MDPs are constructed where the kth MDP considers the k most relevant features only. A state of the MDP is a possible value for the features, while the actions and rewards are the ones described in the game matrix. Since we do not know the policy of the legacy agent in general, we cannot know the transition function of the MDP, and therefore RL is used. To choose the best MDP, they compute the difference between the policies assigned by each and choose the first (least number of features) that is close enough to the following ones.

So far all work has been focused on either planning the optimal behavior if we know the teammate's behaviors and the task (environment rewards) or finding out teammate's behaviors and planning accordingly, still aware of the environment rewards. Melo and Sardinha [11] focus instead on finding out the task and, to some extent, the teammates' behaviors while planning accordingly. Not knowing the task leads to a much more challenging setting, as we now have to find out the environment rewards, the teammates' policies and choose the optimal action without having control of our teammates. They assume the teammates can be thought of a single meta-agent which behaves as a memory-bounded

best-response agent. Note however that in general there are multiple best-responses, and therefore we still need to identify their behavior to predict which of these actions the teammate chooses. They first cast the problem as an online learning problem, where the loss function is 0 if the agent successfully guesses the meta-agent action and 1 otherwise and use an exponentially weighted forecaster [1] to solve it. Another approach is taken where the authors cast the problem as a POMDP where states include information about the history of actions and the real unknown task. A theoretical analysis is performed which bounds the maximal regret of both approaches.

The solutions above are already quite interesting for simple scenarios. Regarding scalability as team size grows, the four solutions have different properties. Both [17] and [14] only deal with teams of 2 agents and therefore we cannot analyze their scalability with respect to team size. For [12], it builds an MDP that has a number of states that grows exponentially with the number of agents, making the algorithm itself also exponential with respect to the team size. Melo and Sardinha [11]'s solution also scales exponentially when solved with the POMDP, but is tractable when using online learning, however it does not perform as good. Analyzing the task size is the same as analyzing the number of actions for this formalization, and all proposed algorithms scale polynomially in this direction.

3.3 Markov Decision Processes

So far we have seen static environments, where the rewards for each joint action is either fixed and constant or has a stationary distribution. For real world applications, we need more expressiveness as there is only a small subset of problems that can be effectively modeled with either a MAB or a fully cooperative iterated game. In this section we will study work that attempts to solve the ad hoc teamwork problem modeling it as an MDP. Modeling it as an MDP means we know a finite set of states S, and there is a reward function for each one of these states $R(s, a)$ and also a transition function $P(s'|s, a)$ between states. The action space, A, includes all joint actions available. Note that if we do not know other agents' behaviors, then we cannot predict the reward value and the transition probabilities as these depend on the joint action and not only on our agent's action, and this is the main challenge introduced by the ad hoc teamwork problem.

Barrett et al. [2] created two algorithms for ad hoc teamwork formalizing the problem as a MMDP – PLASTIC-Model [19] and PLASTIC-Policy [15]. PLASTIC-Model was demonstrated on a specific domain, the same for our work - the pursuit domain. In this problem, we have 4 predators living in a grid world that need to catch a prey and they do that if there is an agent in each side of the prey blocking all movement. At each timestep agents can move in any of the 4 directions - up, down, left, right - and the prey moves randomly also in one of the directions. There cannot be 2 agents in the same cell, and so

[1] Exponentially weighted forecaster is a probabilistic algorithm used to solve sequential prediction problems [18]

movement will be blocked to one of them if they try to move to the same cell. The goal is to catch the prey as fast as possible.

This domain is suitable to study teamwork problems because we need all 4 agents to cooperate making the teamwork necessary to achieve good results. Assuming all these dynamics are known and translated as an MDP, if we knew every agent's behavior, the ad hoc agent could solve for its optimal policy using Value Iteration. If we do not know other agents' behavior, the problem becomes more complicated. Let us say there is a small set of possible behaviors and we are uncertain which one our teammates are using. The solution proposed to find out the most likely behavior is to use Bayes theorem to update his belief:

$$P(model|actions) = \frac{P(actions|model) \times P(model)}{P(actions)} \tag{3.1}$$

$P(actions|model)$ can be computed if we know the possible behaviors and $P(model)$ is a prior belief which is initially set to uniform. $P(actions)$ can be treated as a normalizing constant. If we know the probability distribution of teammate's behaviors, we can again use Value Iteration to compute the optimal policy. However, this requires computing it each time the distribution changes which is very expensive. A possible approach is to use MCTS to approximate the exact solution and sample teammates according to the behavior distribution.

What if teammates behave in a way that is not in the fixed set of known behaviors? Barrett et al. [19] also deal with this by casting it as a supervised learning problem. They assume that the mapping $state \rightarrow action$ (also known as policy) does not change over time and treat it as a supervised learning problem where the agent initially has no samples whatsoever. For this, a decision tree (C4.5 algorithm) is used. As time moves on, the ad hoc agent observes his teammates' actions and retrains the model. By assuming an homogeneous team, there are 3 new samples every timestep to train the decision tree. The input features used by the decision tree algorithm include the x,y coordinates of the prey and teammates, the previous 2 actions and occupation information of the neighboring cells. If we were to use this for another domain, we'd need to pick interesting features suitable for that domain. After training with the new team, their behavior is added to the fixed set of known behaviors, in case the ad hoc agent encounters them again in the future.

Later, Barrett and Stone [15] extended this algorithm to a much more complex domain, HFO introduced by Kalyanakrishnan et al. [20]. They call it PLASTIC-Policy. In this problem, a team of 4 offense agents play soccer against a team of 5 defense agents. The offensive team is trying to score a goal without letting the defense intercept the ball. If it succeeds, the offensive team wins, if the ball gets out of the offensive half of the field or it is intercepted by the defense, the defense wins. There is also a timeout of 50 seconds and the defense team wins if the game lasts that long.

In such a complex domain it might be hard to learn the environment dynamics, and so the authors opt

for a model-free approach where the policy for the ad hoc agent is directly learned without using models for the environment and teammates. They create a behavior model for the team in order to identify them in future episodes. It is a nearest neighbor model which maps states s to next states s'. To pick a policy, the ad hoc agent trains a Fitted Q-iteration (FQI) algorithm for each team which lets it handle continuous states and directly compute the Q-value of each state–action pair. By playing some exploratory episodes for different teams, the ad hoc agent can save the FQI model and the nearest neighbor model it found to play with each team in order to use them in future events.

When the ad hoc agent joins a new team, it must choose between which past team has the most similar behavior to the current one. They employ a similar algorithm to their previous work – maintain a belief over all learned teams and play a policy according to the most likely one. Instead of using Bayes theorem to update beliefs, it uses the polynomial weights from regret minimization as Bayes update rule would drop probabilities to 0 for a single wrong prediction.

If the agent joins a team it has not seen before, it is unable to adapt to them and will simply choose the most similar team it played with in the exploratory phase.

For these 2 algorithms it is harder to analyze their scalability, mainly because they use approximate methods such as MCTS. Their execution time is strongly tied to the approximation error meaning as the complexity increases, more computation time is needed to maintain a good score.

3.4 Other formalizations

Some of the work in this area do not fit in the 3 categories above – either they are specific to a single domain or they apply to all domains or they use a novel formalization of the problem.

Bowling and McCracken [21] study ad hoc teamwork in robot soccer. They assume that the team plays using a playbook. This playbook contains plays specifying roles needed to execute them and how to execute them. The ad hoc agent needs to select a play and choose its own role within that play, without communicating with its team. After playing hundreds of games, the ad hoc agent can learn how the team performs role assignment and predict correctly which role it should have in each play. Genter and Stone [22] focus on the problem of leading a flock of agents for which the behavior is assumed to follow the Boid model in order to get them in a specific orientation. They propose an algorithm to coordinate multiple ad hoc agents to influence the remainder of the flock for a desired goal. These two papers focus on very specific domains and do not generalize for other domains but we are more concerned in a wide range of domains.

Liemhetcharat and Veloso [23] deal with the problem of ad hoc team formation, where a group of agents that are unaware of each other's behaviors, and with different capabilities, must cooperate with each other as a team. While this paper focus on selecting teammates, the ad hoc teamwork problem is

Table 3.1: Summary of approaches that tackle the ad hoc teamwork problem

Reference	Domain	Number of teammates	Teammate behavior	Task
[8]	MAB	1	greedy	known
[9]	MAB	1	greedy	known
[10]	MAB	≥ 1	one of a known set	known
[17]	NFG	1	ϵ-greedy best-response	known
[12]	NFG	≥ 1	best-response	known
[14]	NFG	1	Markovian	known
[11]	NFG	≥ 1	best-response	unknown
[19]	MDP	≥ 1	unknown	known
[15]	MDP	≥ 1	unknown	unknown
[24]	BSG	≥ 1	one of a known set	known

more about cooperating with them after they have been selected.

Albrecht and Ramamoorthy [24] model the ad hoc teamwork problem as a Bayesian Stochastic Game (BSG). This concept is quite similar to a MDP, where there are multiple states, a transition function between states and a policy that each agent follows. This policy depends on the history of joint actions and the type of agent. The difference between BSGs and MDPs is this type that each agent has. A type can be seen as the role of the agent, and it can change over time. They show an algorithm to compute a sequence of actions if the behaviors of teammates are known. If these are unknown, then they propose a ”conceptualized type” where the user defines a function that measures similarity between states along with a radius and the behavior is learned from those and adapted as time goes on.

3.5 Summary

In Table 3.1 it possible to see a summarization of some solutions to the ad hoc teamwork problem. Since most authors of these papers were not concerned with scalability, it is hard to report it here. However it is easy to see that all exact optimal algorithms scale exponentially with the number of agents and polynomial with the number of actions available. For approximated algorithms it is harder to see if they scale well, as it depends on the approximation error.

Out of all the approaches, PLASTIC-Model [4] is the one that has objectives most similar to ours, as it explicitly includes planning and models the domain as an MDP. However it still has strong assumptions – such as a perfect model of the domain and pre-existing models of various teammates. In some scenarios it is not realistic to assume we have access to this knowledge. Our approach thus goes

beyond PLASTIC-Model by considering teammate behaviors and environment dynamics are unknown.

4

Solution

Contents

4.1 Problem formalization

We model the problem as an MMDP $\mathcal{M} = (N+1, \mathcal{X}, \mathcal{A}, \{\mathbf{P}_{\boldsymbol{a}}\}, r, \gamma)$, where

- $N+1$ is the number of agents considered. We assume throughout that the ad hoc agent is agent 0 and there are N teammates.
- $\mathcal{X}$ is the (countable) set of possible states of the environment. The state at time step t is denoted as the random variable $\mathrm{x}(t)$ (we use upright letters to denote random variables) taking values in $\mathcal{X}$.
- $\mathcal{A}$ is the joint action space. In other words, $\mathcal{A}$ is the cartesian product of $N+1$ individual action sets, $\mathcal{A}_0, \ldots, \mathcal{A}_N$. Each $\mathcal{A}_n$ is the individual action set of agent n.

  We write a_n to refer to an element of $\mathcal{A}_n$, and $\boldsymbol{a}$ to refer to an element of $\mathcal{A}$, understood as a vector

  $$\boldsymbol{a} = (a_0, a_1, \ldots, a_N).$$

  We write $\boldsymbol{a}_{-0}$ to denote a reduced joint action

  $$\boldsymbol{a}_{-0} = (a_1, \ldots, a_N),$$

  obtained from a joint action $\boldsymbol{a}$ by removing the component corresponding to the action of agent 0. The joint action at time step t is denoted as a random vector $\mathbf{a}(t) \in \mathcal{A}$, and we write $\mathrm{a}_n(t)$ to denote the nth component of $\mathbf{a}(t)$, corresponding to the individual action of agent n at time step t.
- $\mathbf{P}_{\boldsymbol{a}}$ denotes the transition probabilities associated with joint action $\boldsymbol{a}$. We write, interchangeably, $\mathbf{P}_{\boldsymbol{a}}(y \mid x)$ and $\mathbf{P}(y \mid x, \boldsymbol{a})$ to denote the transition probability

  $$\mathbf{P}_{\boldsymbol{a}}(y \mid x) = \mathbf{P}(y \mid x, \boldsymbol{a}) = \mathbb{P}\left[\mathrm{x}(t+1) = y \mid \mathrm{x}(t) = x, \mathbf{a}(t) = \boldsymbol{a}\right].$$

- $r(x, \boldsymbol{a})$ is the expected reward function representing the target task.
- γ is a constant in $[0, 1)$, denoting the discount.

The difference between solving the MMDP in the ad hoc teamwork setting is that we only control one of the agents and therefore only one action is provided at each timestep. Also, the transition function $\mathbf{P}_a$ is not known to the ad hoc agent, which constitutes another obstacle to directly solving the MMDP.

Since we cannot control other agents' actions, we can also see this problem as a MDP where the transition function is a combination of the teammates' behaviors and the environment dynamics. Denoting $\boldsymbol{a}_{-0}$ as the joint action of the legacy agents, and a_0 as the ad hoc agent's action and defining

$\pi(x, \boldsymbol{a}_{-0}) \rightarrow [0, 1]$ as the teammate behavior function, then this MDP transition function would simply be:

$$\mathbf{P}_{a_0}^{\text{MDP}}(x(t+1)|x(t)) = \sum_{\boldsymbol{a}_{-0} \in \mathcal{A}_{-0}} \pi(x(t), \boldsymbol{a}_{-0}) \mathbf{P}_{(a_0, \boldsymbol{a}_{-0})}^{\text{MMDP}}(x(t+1)|x(t)) \tag{4.1}$$

Similarly, the reward function would be:

$$r^{\text{MDP}}(x(t), a) = \sum_{\boldsymbol{a}_{-0} \in \mathcal{A}_{-0}} \pi(x(t), \boldsymbol{a}_{-0}) \mathbf{P}^{\text{MMDP}}(x(t), (a_0, \boldsymbol{a}_{-0})) \tag{4.2}$$

This representation of the problem is much more interesting and allows us to focus on a single action - the ad hoc agent's. So, the new MDP can be described as the tuple $(\mathcal{X}, \mathcal{A}_0, \mathbf{P}_a^{\text{MDP}}, r^{\text{MDP}}, \gamma)$, where:

- $\mathcal{X}$ is the same state space as the MMDP described previously
- $\mathcal{A}_0$ is the action space of the ad hoc agent
- $\mathbf{P}_a^{\text{MDP}}$ is the transition function as computed in Equation 4.1, which is simply the expected transition function for a given teammate behavior.
- $r^{\text{MDP}}(x, a)$ is, similarly to the transition function, the expected reward for a given teammate behavior given by Equation 4.2
- γ is the same discount factor as in the MMDP

It is clear that if we knew the legacy teammates' behaviors and the dynamics of the environment, we could solve the MDP with the classical value iteration solution. It is also tempting to simply use Q-learning to learn a policy without knowing the transition function $\mathbf{P}_{\text{MDP}}$. We show in the Results section that this approach takes a lot of iterations to converge to an acceptable performance, partly due to the huge state-action space of the domain used for testing.

4.2 3-layered architecture for ad-hoc teamwork

Our solution can be segmented in three parts that address the three distinct challenges described in the previous section. These three parts are: model-based reinforcement learning using a deep neural network to model the environment, fictitious-play like model to learn teammate's policies and MCTS planning to select the optimal action given the previous models. In this section we'll go into detail about these segments and how they interact to provide a solution to the ad hoc teamwork problem.

An architecture of the solution proposed can be seen in Figure 4.1. The different boxes represent functions developed in this work. At every timestep the ad hoc agent observes the state and reward given by the environment. It uses the state, along with the previous state and previous joint action to

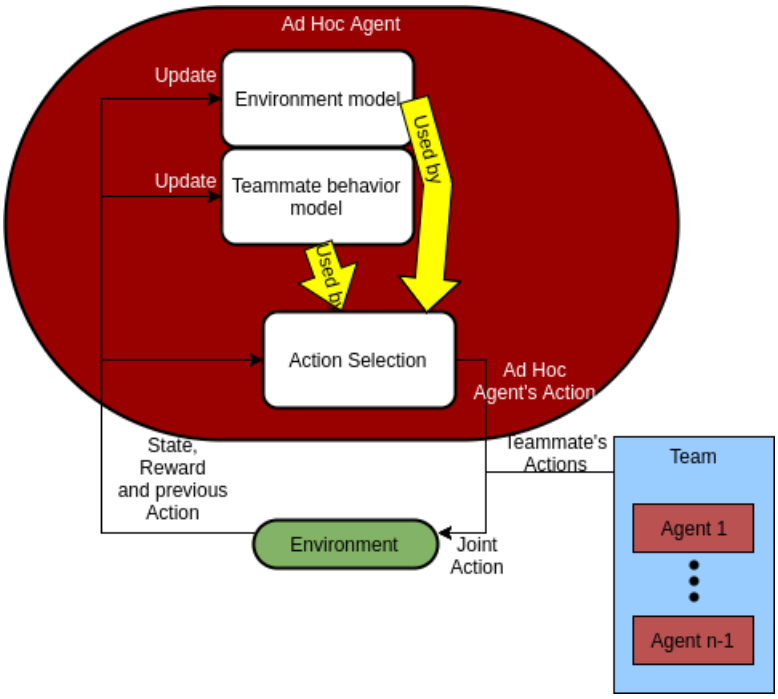

Figure 4.1: Architecture of proposed solution

update the environment model, that is the approximation of the function $P(x(t+1)|x(t), \boldsymbol{a})$. It uses the previous state and joint action to update the teammates model, that is the approximation of their policy, $\pi(x, \boldsymbol{a}_{-0})$. These two models are used to compute a solution to the underlying MDP and then select the ad hoc agent's action, which will act on the environment along with its teammate's actions. The various functions will be modeled by the following:

- Environment model – A neural network receiving as input a state and joint action and outputting the most likely next state.

- Teammate behavior model – A neural network receiving as input a state and outputting a prediction of the distribution of their actions

- Action selection – MCTS receiving as input the models and current state, and outputs the best action for the ad hoc agent. For the rollouts use the teammate behavior model to predict the joint action and use the environment model and predicted joint action to predict the next state.

The environment is assumed to be the same through episodes, and therefore the environment model will be kept even if the team changes. The teammate's model needs to be retrained after the team changes, but previous models can be used as initialization for the new teams.

4.3 Learning teammate's policies with a deep neural network

One of the challenges of the ad hoc teamwork setting is to predict the behavior of the legacy teammates, that is the function $\pi(x, \boldsymbol{a}_{-0})$. We define $\mathcal{F}(x, i)$ as a set of features for a given state x as seen from the agent i. In this work, we make the following assumptions:

- Teammates have a static policy – meaning the action distribution they choose from depends only on the current state. This implies they do not learn over time.
- Actions chosen are visible to the ad hoc agent
- Behaviors are homogeneous among the teammates – meaning they all follow the same policy assuming states are relative to each agent. This means $\mathcal{F}(x_1, i) = \mathcal{F}(x_2, j) \Rightarrow \pi(x_1, a_i) = \pi(x_2, a_j)$

These assumptions allow us to model the teammate policy using the fictitious play algorithm. Using fictitious play, the predicted policy is given by the following equation:

$$\hat{\pi}(x, \boldsymbol{a}_{-0}) = \frac{N(x, \boldsymbol{a}_{-0})}{\sum_{\boldsymbol{a'}_{-0} \in \mathcal{A}_{-0}} N(x, \boldsymbol{a'}_{-0})} \tag{4.3}$$

Where $N(x, \boldsymbol{a})$ represents how many times the team played actions $\boldsymbol{a}$ in the state x in the past. A problem with this approach is that it requires lots of state-action visits if the domain has a sizable state-action space. One way to improve this is to leverage the similarity between different state-action pairs.

We propose looking at this problem as a supervised learning problem, where the input is the state (or state features as seen from one teammate $\mathcal{F}(x, i)$) and the output is the action distribution of a single teammate. Samples to train this model are collected as the ad hoc agent navigates through the environment and observes other agent's actions.

We chose a neural network as the teammate behavior model, with the input being state features and an output for every action estimating the probability that a given legacy teammate chooses that action in the given state. Since in the specified domain in this thesis every teammate has the same policy, we can use the same neural network to predict each of the agent's actions, and train it with all actions observed.

Using a neural network to model the teammate policy allows us to generalize the behavior of the team in a given state to similar states, using state features to measure this similarity. This works in huge domains where mapping every state-action pair to a probability in a non-parametric way, for example when using fictitious play, would be intractable as available memory is limited.

4.4 Task identification with model-based reinforcement learning

Task identification can be split into two other tasks: learning the task goal (the reward function) and the environment dynamics (the transition function). Environment dynamics are described by the distribution

$\mathbf{P}_{\boldsymbol{a}}(y|x)$ for every $\boldsymbol{a}, x, y$, where $\boldsymbol{a}$ denotes the joint action, x the current state and y the next state.

To learn the dynamics of the environment, a feedforward neural network is used, as seen in Figure 5.3, similarly to the teammate behavior. In this case, however, we do not have an output node for each possible state, as it would require a huge number of nodes. Instead, we assume the state can be fully described by a reasonable number of features, denoted $\mathcal{F}(x)$, for which a metric distance is defined. In particular, the state transitions can be fully featured in terms of a translation in feature-space. If $\mathcal{F}(x(t))$ is a feature vector that describes the state at time step t then the transition from state $x(t)$ to state $x(t+1)$ can be fully described by a translation vector $\mathcal{F}(x(t+1)) - \mathcal{F}(x(t))$. For example, in a grid world we could have a set of nodes for positions for each object.

The output nodes of the neural network are the difference between the features of the current state and the next state, that is $\mathcal{F}(x(t+1)) - \mathcal{F}(x(t))$. By using this variable change trick, the neural network only needs to output a smaller range of values, making the training fairly easier as validated by our experiments.

The input nodes are the features that describe the current state, $\mathcal{F}(x(t))$, along with the actions of all agents, $\boldsymbol{a}$. Actions are described as one hot encoded vectors [1].

Ideally, the model would output a distribution over all possible next states. However, this would require an output node for every possible state, which is intractable for many domains (e.g. the pursuit domain). Instead, the model predicts the most likely next state given the current state and the actions taken. By using an environment model, we can reuse this information across different teams. That is, even though the teammate behavior changes through episodes, the environment dynamics remain the same, allowing the agent to improve its policy with changing teams.

4.5 MCTS planning

If the agent knew both the teammate behavior and environment models, we could compute $\mathbf{P}^{\text{MDP}}$ as described in Equation 4.1 and solve it using Value Iteration. However, we are interested in problems which have large state spaces and Value Iteration would take a long time in those cases.

We decide to use MCTS to plan the optimal actions for the ad hoc agent as it provides a good approximation of the optimal policy. Given the teammate behavior model and the environment model for the MMDP, it is possible to compute the environment dynamics for the resulting MDP. Of course that the planning will not be optimal since both the teammate behavior model and the environment model are just approximations and we do not have infinite time.

An example of the expansion step of the MCTS is shown in Figure 4.2. In this step, we use the model described in Section 4.3 to predict what actions the ad-hoc agent's teammates will take in a given state.

[1] A one hot encoded vector is a vector where each entry is associated with a possible value of the set we want to describe. For a given value a in the set, all entries are 0 except for the entry corresponding to a, for which the value is 1.

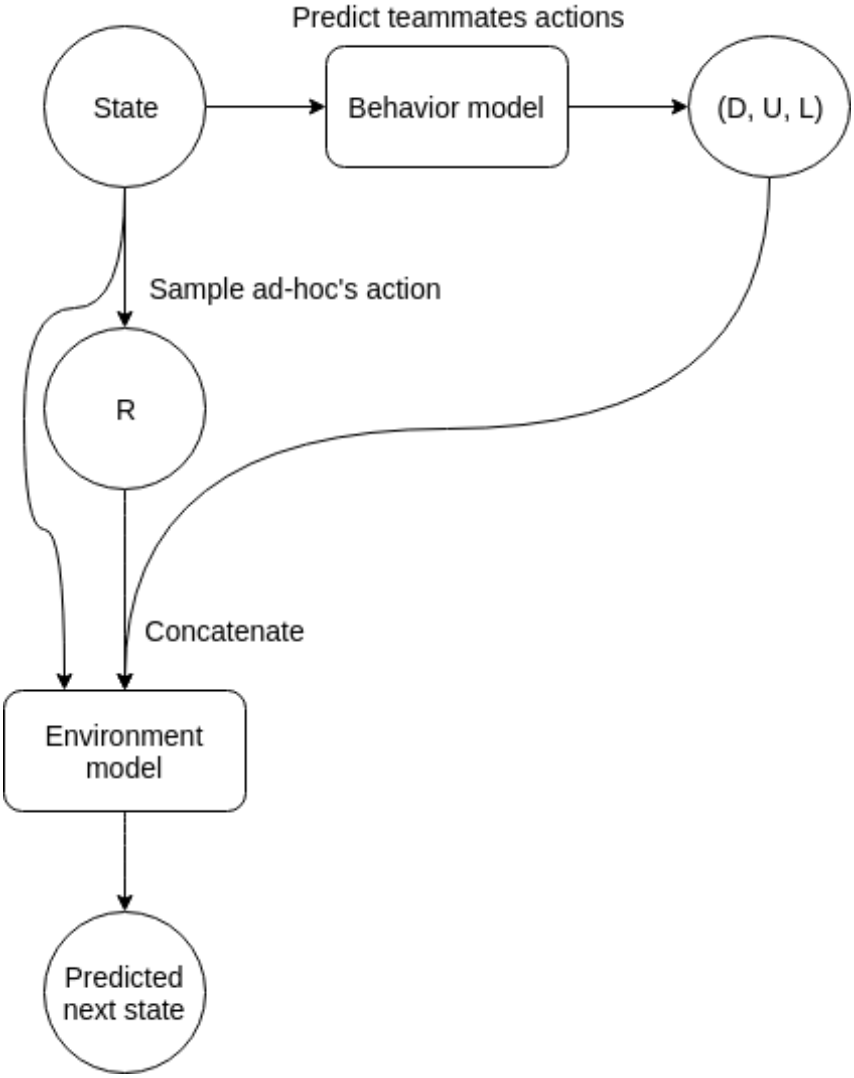

Figure 4.2: An example of the expansion step of MCTS planning using previously learned models of both behavior and environment. (D,U,L) represents an example of actions by the three teammates and R represents an action by the ad hoc agent.

After we have the predicted actions, we concatenate them with the ad-hoc's action to build a joint action. This joint action and the given state is fed into the environment model described in Section 4.4 which outputs the most likely next state, concluding the expansion step of MCTS. The same is done in the simulation step, where we need to generate the next state given the current state and a random action from the ad-hoc agent.

5

Evaluation

Contents

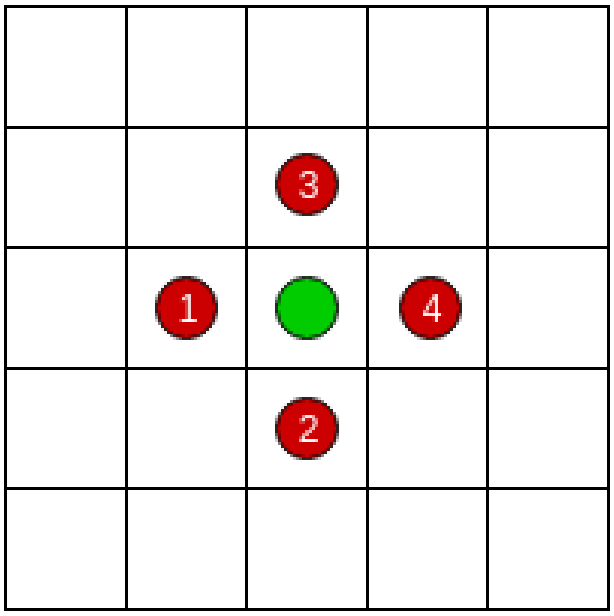

Figure 5.1: An example of a capture position. Prey is represented as a green circle and predators are represented as red circles. In this case, the prey is unable to move in any direction, ending the game and all predators earn a reward.

5.1 Pursuit domain

We evaluate our work in the pursuit domain, similarly to Barrett et al. [4], which is used widely in multi agent reinforcement learning. In the pursuit domain, there is a $N \times N$ toroidal grid world where four predators, starting in random positions, chase a prey. At each timestep, the prey chooses a random action, either Up, Down, Right or Left and tries to execute it. If the cell is blocked, the move fails and the prey remains in the same cell. If the prey is surrounded by all four sides, it is said to be captured and the episode ends – an example can be seen in Figure 5.1. Similarly, predators choose one of the four actions at each timestep with the goal of surrounding the prey. If two agents try move to the same cell, the one who succeeds is chosen at random and the other remains in its cell.

Since the world is toroidal, if an agent is in the top-left corner $(0, 0)$ and moves left, it goes to the top-right corner $(N - 1, 0)$. Similarly, if the prey is in the top-left corner, then it will be blocked if and only if all four adjacent positions – $(0, 1)$, $(1, 0)$, $(0, N - 1)$, $(N - 1, 0)$ – are taken by the predators.

This domain is well suited for the ad-hoc teamwork problem. Some of the reasons are:

- Fully cooperative game, that is, all agents receive the same reward at every timestep
- All agents are needed to achieve the goal, so the ad-hoc cannot play independently of its team
- The optimal strategy of one agent depends heavily on the strategy of the other agents. For example, if the team avoids collisions by giving priority to the agent farthest from the prey, an agent following the opposite strategy will lead to a lot of collisions.

Also, the domain is simple enough to be modeled by a MMDP but complex enough that it is not tractable to solve it with an exact solution. For a $N \times N$ world with four predators and one prey, it has approximately N^{10} different configurations if we consider each agent to be different from its teammates.

Formally, we define this domain as the MMDP $\mathcal{M}_L = (4, \mathcal{X}, \mathcal{A}, \{\mathbf{P}_{\boldsymbol{a}}\}, r, \gamma)$, where:

- L is the height and width of the grid
- $\mathcal{X} = \{(x_0, y_0, \cdots, x_4, y_4) : \forall_{1 \leq i \leq 4}, 0 \leq x_i, y_i \leq L\}$

  Each (x, y) represents the position of one agent
- $\mathcal{A} = \{(a_0, a_1, a_2, a_3) : \forall_{1 \leq i \leq 4}, a_i \in \{U, D, L, R\})\}$

  Where (U, D, L, R) represent the actions of moving Up, Down, Left or Right.
- $r = \begin{cases} 0 & \text{if prey is blocked} \\ -1 & \text{otherwise} \end{cases}$

The transition function $\mathbf{P}_{\boldsymbol{a}}$ is defined by the movements of the agents. If an agent moves towards a direction, the probability of success is always 1 if the destination cell is empty and if no other agent tried to move towards it. If that's not true, then a collision occurs and one of the agents (selected at random) is able to move and all the other ones are unable to move. The prey always chooses each direction with probability 0.25.

5.2 System layout

The architecture of the neural network for the teammates' behavior can be seen in Figure 5.2. It is composed of two hidden layers, each with 128 nodes. Its input are the state features, relative to teammate i and it outputs the predicted distribution of teammate i actions. A relu activation function is used between all layers except the last one, where softmax is used instead. During training, the neural network is optimized using the ADAM [7] optimizer, with a cross-entropy loss function.

The architecture of the neural network for the environment is described in Figure 5.3. It is composed of two hidden layers, an input layer containing the current state features and outputs the predicted difference between the current state and next state. The activation function used is relu for every layer except the last, where we do not use any activation. We optimize, using ADAM [7], the loss function $||\mathcal{F}(x(t+1)) - \hat{\mathcal{F}}(x(t+1))||^2$, which, denoting $\boldsymbol{o}$ as the output of the neural network, corresponds to $||(\mathcal{F}(x(t+1)) - \mathcal{F}(x(t))) - \boldsymbol{o}||^2$.

We use UCT which is a variant of MCTS where the selection step uses a tree policy to balance exploration and exploitation. For the simulation step, we use random rollouts with a cap of 10 steps for the depth. This value was decided to be a good tradeoff between final score and computational performance after some experimentation.

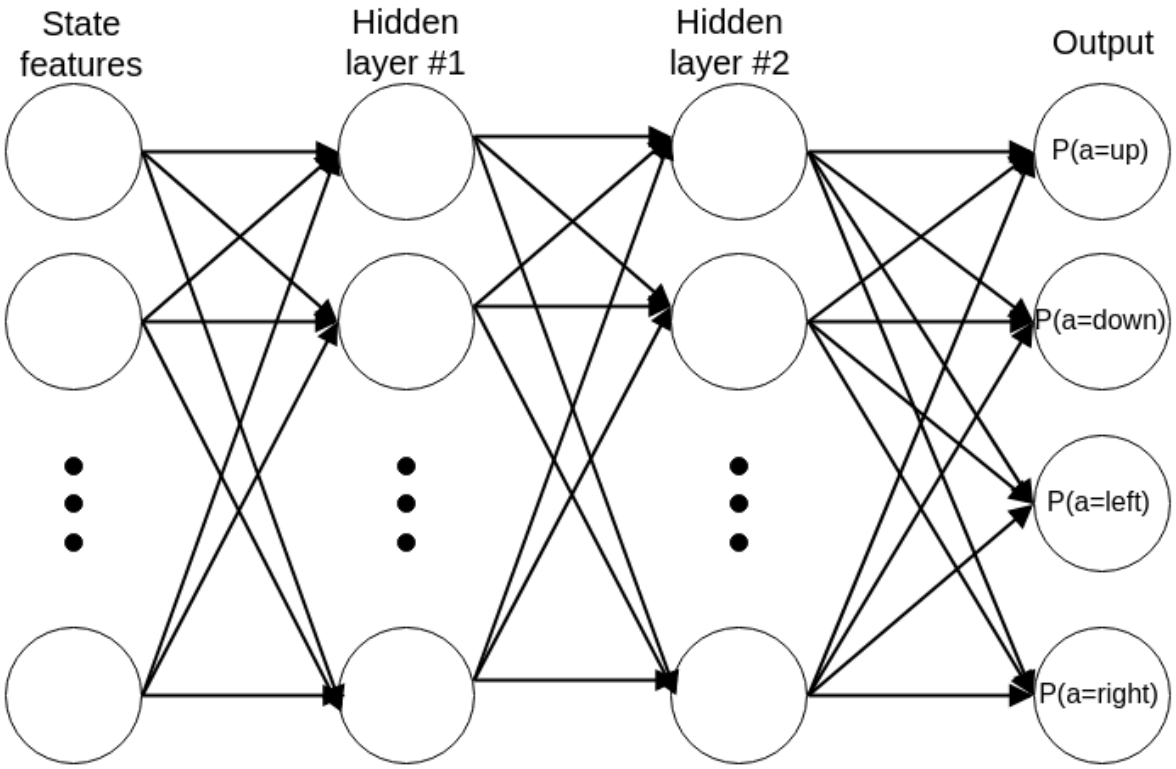

Figure 5.2: Architecture of teammate behavior neural network

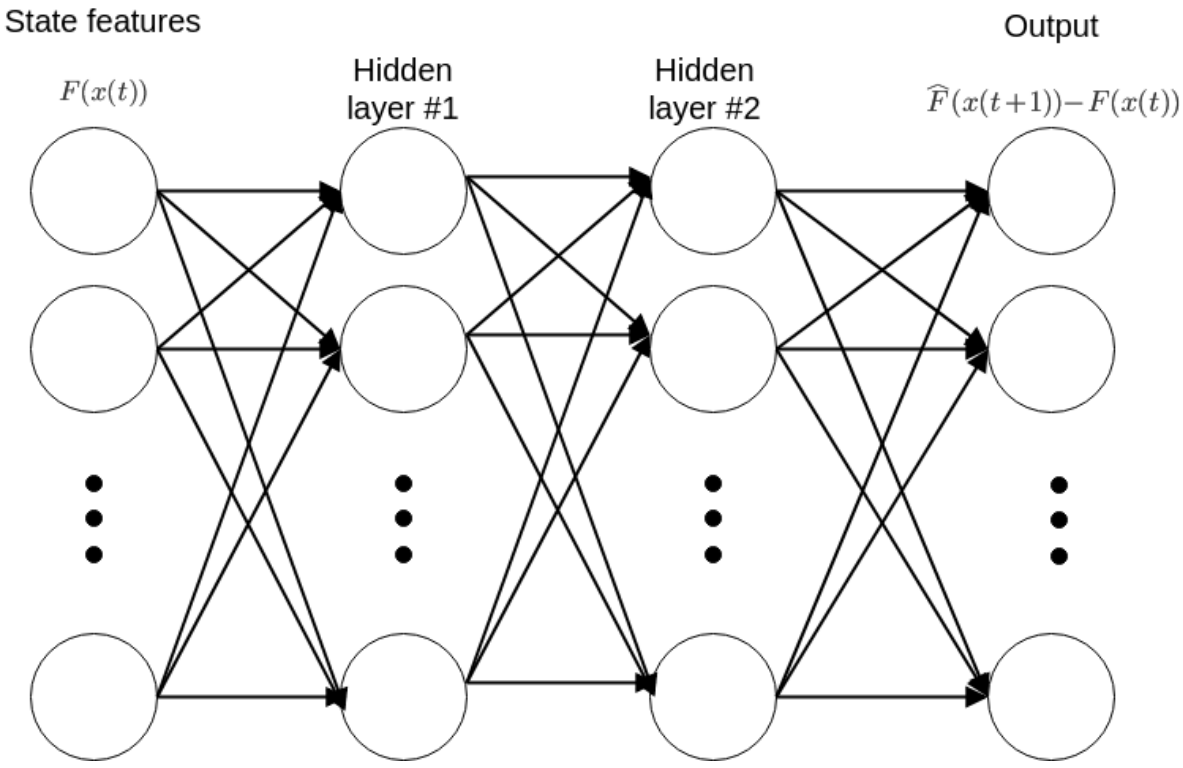

Figure 5.3: Architecture of environment transition neural network

5.3 Setup

We evaluate our work using the framework described in [4]. In such framework, a team of agents plays a game in a given domain obtaining a score s. Then one of the agents is replaced by an ad hoc agent, and the game is played again, obtaining a score s'. By using different ad hoc agents in each game we can compare their scores against each other. Our score corresponds to the amount of steps the team takes to capture the prey, and therefore lower is better.

All results shown in the following sections are taken as an average of 100 runs, using different initial states. The initial state heavily affects the final score, meaning the score has high variance, but we found that running 100 times gives us a small enough confidence interval for the average score. All values are reported along with a confidence interval of 90% .

We use two different kinds of teammates, each with an handcoded policy detailed in the following sections. For the ad hoc agent, we use an ϵ-greedy policy with $\epsilon = 0.9$ and after the 1st, 50th and 200th episode, we evaluate our ad hoc agent in 100 different initial states, with $\epsilon = 0$, and plot the average.

The ad hoc agents updates its models after each episode, by running all observations from the last episode through the neural network. This is done for a single epoch after each episode.

For the UCT, we use 1000 playouts and each rollout has a maximum depth of 10 – in our solution and in PLASTIC-Model. For the exploration constant c, we use 10.0 – as this seemed to be a good tradeoff between exploration and exploitation in our experiments.

We compare our results against the following:

- All agents follow the handcoded policy, no ad-hoc agent is present.
- PLASTIC-Model – PLASTIC-Model with a perfect model of the teammates' policies and the task. This reduces to a planning problem, for which they use a variant of UCT.
- PLASTIC-Model(NN) – PLASTIC-Model with a perfect model of the task but with our learned model of teammates' policies.
- Deep Q-Networks (DQN) – a model-free reinforcement learning algorithm that, similar to our work, does not have a model of the teammates nor the task. It learns for 200 episodes.

5.3.1 Features selected for teammate policy learning

In the pursuit domain, teammates can select one of four actions at every timestep: up, down, left or right. Our goal is to find out the distribution of each teammate's action, $\pi(x, a_i)$. Using the neural network described in section 4.3 requires us to build features of the state relative to each teammate. We selected the following feature set:

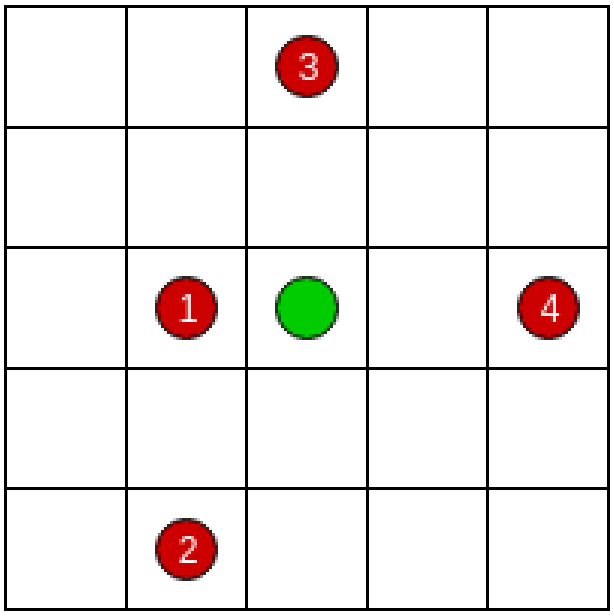

Figure 5.4: Example state in the pursuit domain

1. Difference of positions in the x-axis between the teammate and the prey
2. Difference of positions in the y-axis between the teammate and the prey
3. Difference of positions in the x-axis between the teammate and its closest teammate
4. Difference of positions in the y-axis between the teammate and its closest teammate
5. Difference of positions in the x-axis between the teammate and its 2nd closest teammate
6. Difference of positions in the y-axis between the teammate and its 2nd closest teammate
7. Difference of positions in the x-axis between the teammate and its 3rd closest teammate
8. Difference of positions in the y-axis between the teammate and its 3rd closest teammate

This set allows to describe the state relative to a teammate. If, in a given pursuit world, we switch the position of 2 agents, their feature set is identically switched. This only happens because this feature set is completely independent of teammate's identifiers. Since teammate behaviors are homogeneous and we have features relative to each teammate, we can use the same neural network to predict actions for all teammates.

In figure 5.4, we can see an example of a valid state in the pursuit domain. In that state, the feature set for teammate 1 is $[1, 0, 0, 2, -2, 0, 1, -2]$. Firstly, we have the distance to the prey $(1, 0)$, then the closest teammate is teammate 2, which is distanced $(0, 2)$, then teammate 4 for which the distance is $(-2, 0)$ since the world is toroidal and then finally teammate 3, which is $(1, -2)$ blocks apart from

teammate 1. If we switch the numbers 2, 3 and 4 in the figure, the feature set for teammate 1 remains the same, as it is independent of the teammate identifiers.

5.3.2 Features selected for environment dynamics learning

In the pursuit domain, the reward given is -1 for every state except for the final state, which is 0. Environment dynamics regarding the teammates positions are deterministic as long as there are no collisions, but the prey acts randomly. In case of collision, ties are selected randomly as each cell can only be occupied by a single agent. To represent the state, we select the following features:

- Position in the x-axis of the prey
- Position in the y-axis of the prey
- Position in the x-axis of teammate 1 (ad-hoc agent)
- Position in the y-axis of teammate 1 (ad-hoc agent)
- Position in the x-axis of teammate 2
- Position in the y-axis of teammate 2
- Position in the x-axis of teammate 3
- Position in the y-axis of teammate 3
- Position in the x-axis of teammate 4
- Position in the y-axis of teammate 4

This feature set contains all information needed to reconstruct the state and also the metric distance between states is very well defined, which is useful to compute similarity between states.

5.4 Results

5.4.1 Greedy teammates

In this section we evaluate our work using greedy teammates, as defined in [2]. Greedy teammates behave according to a fixed set of rules:

- If already adjacent to the prey, move towards it
- Pick the closest free cell adjacent to the prey

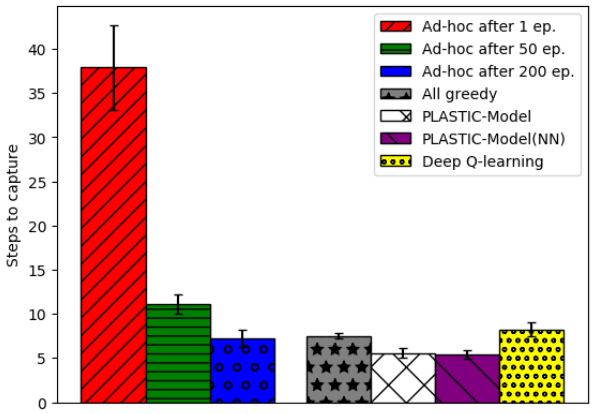

Figure 5.5: Results for our ad-hoc agent after 1, 50 and 200 episodes in a 5x5 world

- If distance in x-axis is bigger than the y-axis, move either left or right, whichever gets it closer to the prey, else move either up or down, whichever gets it closer to the prey.

- If both directions blocked, move randomly

This is a very simple teammate, almost deterministic and does not take into account its teammates unless they are already neighboring the prey. As a result, it works well in small (e.g. 5×5) worlds, but gets worse as we increase the world size. Our ad hoc agent will play with these teammates for some episodes, and learn as it plays.

In figure 5.5 we show its score after playing 1, 50 and 200 episodes in a 5×5 world. We can see that it improves with more episodes of training and even surpasses a team of all greedy agents.

Compared to PLASTIC-Model and PLASTIC-Model with our teammate behavior model, it performs somewhat worse. This is expected as those two solutions have a perfect model of the environment.

DQN has the worst result of all solutions, presumably because it requires more data to converge to a good performance.

In figures 5.6 and 5.7 we show its score for bigger worlds. It's clear that its performance can keep up with bigger worlds and in all three world sizes it has similar results to using all greedy teammates.

Even though it scores worse than PLASTIC-Model, this can be easily explained as PLASTIC-Model has the correct model of the greedy agents and knows the task exactly, while our approach learns it with time. When using our teammates' behavior model in PLASTIC-Model(NN), it gets closer to our results.

Looking at the results in figure 5.7, we conclude that the DQN approach gets even worse with bigger worlds, due to having only a small number of episodes for training. Our solution can cope with this increasing state-space size while also having only 200 episodes to train.

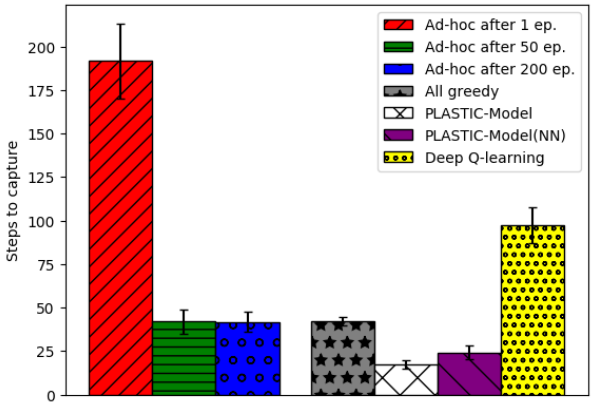

Figure 5.6: Results for our ad-hoc agent after 1, 50 and 200 episodes in a 10x10 world

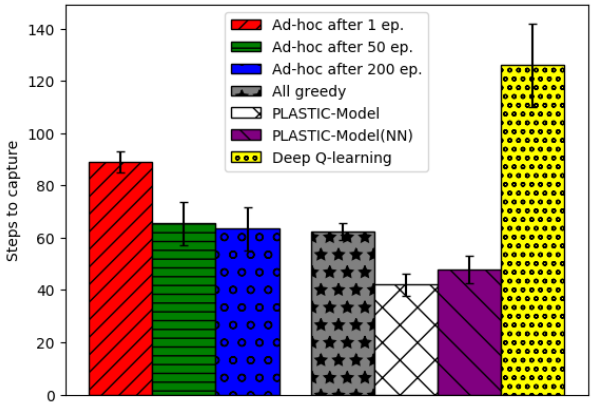

Figure 5.7: Results for our ad-hoc agent after 1, 50 and 200 episodes in a 20x20 world

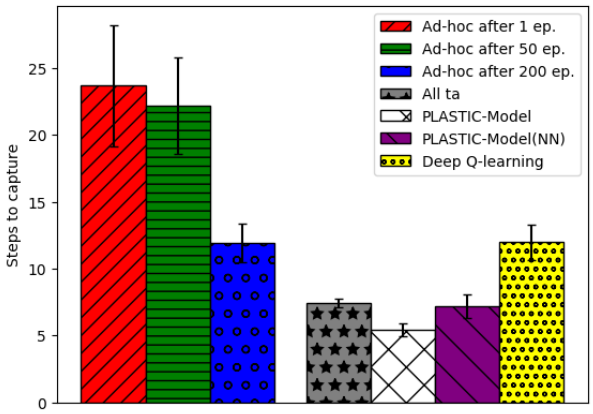

Figure 5.8: Results for our ad-hoc agent, playing with team aware agents, after 1, 50 and 200 episodes in a 5×5 world

5.4.2 Teammate aware teammates

In this section we evaluate our work using teammate aware agents, again defined in [2]. Teammate aware agents try to prioritize farther from the prey agents so they can use the fastest path to chase the prey. They also use A* to plan for the optimal path assuming teammates are static obstacles. Formally, they follow the following set of rules:

- Sort the predators based on the largest distance to the prey
- For each predator, pick the free cell that is neighbor to the prey, that is closest to them and has not been picked yet
- If the predator is already at the picked cell, move towards the prey
- Else, Use A* to pick an action, assuming teammates are static obstacles

This behavior works much better in bigger worlds, as can be seen in Figure 5.10 when compared to Figure 5.7. It is also a behavior that is not as easy to learn due to the avoidance of obstacles done by A* and the pre-coordination used in the destination picking.

Again, in figure 5.8 we show its score after playing 1, 50 and 200 episodes in a 5×5 world. When compared to greedy agents in Section 5.4.1, we can see that our solution does not work as well. This is explained with 2 reasons: team aware agents have a more complex behavior, which is hard to learn and team aware agents rely a lot on their team (to avoid collisions and prioritize agents farther from the prey), making mistakes more heavily penalized.

In figures 5.9 and 5.10 we show our score in 10×10 and 20×20 worlds. The results are similar to the previous section regarding greedy agents: the ad-hoc agent still manages to get a good score in big worlds when following our solution, but it degrades heavily when using DQN. However, all results are

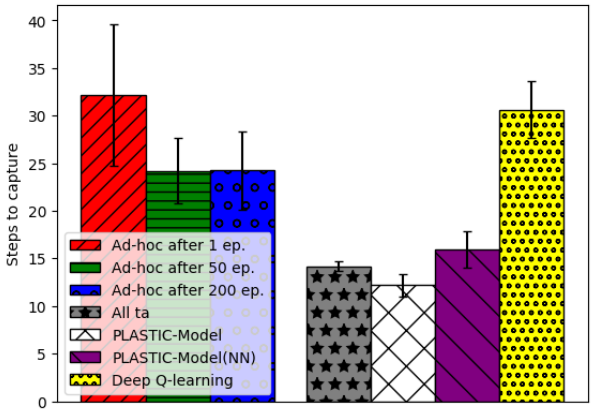

Figure 5.9: Results for our ad-hoc agent, playing with team aware agents, after 1, 50 and 200 episodes in a 10×10 world

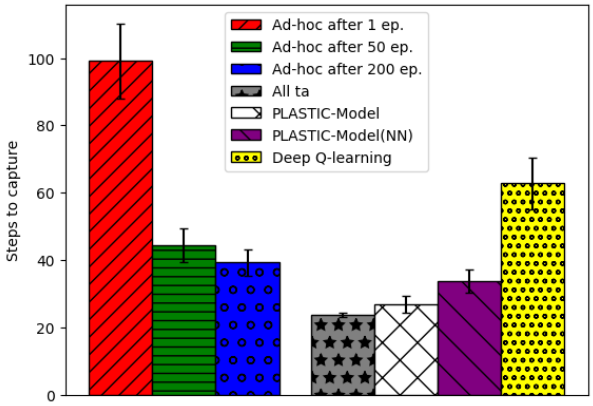

Figure 5.10: Results for our ad-hoc agent, playing with team aware agents, after 1, 50 and 200 episodes in a 20×20 world

overall worse than when the ad-hoc agent played with greedy teammates, and again this is explained with the increased complexity of playing with team aware agents.

6

Conclusion

Contents

Ad hoc teamwork is an emerging research field which intends to create an agent that can cooperate with a team of other agents without pre-coordinating with them. As more agents enter the real world, it is expected that they need to cooperate with others that they have never worked with and with unknown behaviors, making ad hoc agents necessary and useful.

Most of existing work in this area focuses on theoretical results with very strong restrictions regarding number of teammates, complexity of behavior and available information. Some work focuses on empirical results such as PLASTIC-Model and PLASTIC-Policy, where our inspiration comes from.

We propose a novel architecture combining deep neural networks to model the teammates and the task and MCTS for planning. This differs from both PLASTIC-Model, where the task is known and teammates behaviors are either known or learned beforehand with decision trees, and PLASTIC-Policy, where the policy is directly learned without learning teammate's behaviors and environment dynamics.

By comparing our solution with state-of-the-art PLASTIC-Model, we show that we can get comparable performance while using less information regarding the team and the task. Looking at PLASTIC-Model results with our learned teammate behavior, we can see that the difference in performance comes from the approximation error in learning teammates' policies.

We also compare it with DQN, a solution widely used in reinforcement learning, with similar assumptions to our work. DQN is quite similar to PLASTIC-Policy for our intents and purposes, as we are more interested in online learning of teams that the ad hoc agent has never seen before. We show that DQN, when compared to our work, takes a long time to learn, as usual for model-free approaches. Additionally, explicitly modeling the task as we do, allows the agent to reuse this information when teams change, unlike DQN and PLASTIC-Policy.

6.1 Future work

In the future, it would be interesting to adapt the algorithm to work in continuous state and action-spaces such as the Half Field Offense domain, so it would be possible to compare it against the PLASTIC-Policy [5] algorithm.

Another point of improvement would be to allow the environment model to output a distribution of possible next states, instead of the most likely one. This could be done using, for example, Mixture Density Networks [25] for continuous state spaces or large ones where a metric distance is defined (such as the pursuit domain). For small discrete spaces, a simple neural network with an output for each possible state would be the best fit.

Regarding planning, MCTS turned out to be quite slow. A faster planning algorithm would make this work more useful and interesting. One way would be to use a well-studied optimization of MCTS such as RAVE [26] or to simply use an alternative to MCTS.

It would also be interesting to try other supervised learning techniques instead of neural networks as they might be more data efficient.

Bibliography

[1] P. Stone, G. A. Kaminka, S. Kraus, and J. S. Rosenschein, "Ad Hoc Autonomous Agent Teams : Collaboration without Pre-Coordination," *Twenty-Fourth AAAI Conference on Artificial Intelligence*, no. July, pp. 1504–1509, 2010.

[2] S. R. Barrett, P. Stone, and R. Mooney, "Making Friends on the Fly : Advances in Ad Hoc Teamwork," 2014.

[3] M. Benda, V. Jagannathan, and R. Dodhiawala, "On Optimal Cooperation of Knowledge Sources - An Empirical Investigation," Boeing Advanced Technology Center, Boeing Computing Services, Seattle, WA, USA, Tech. Rep. BCS–G2010–28, 1986. [Online]. Available: http://www.cs.utexas.edu/~shivaram

[4] S. Barrett, P. Stone, and S. Kraus, "Empirical evaluation of ad hoc teamwork in the pursuit domain," *Autonomous Agents and Multiagent Systems (AAMAS)*, no. May, pp. 567–574, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2031678.2031698

[5] S. Barrett and P. Stone, "Cooperating with Unknown Teammates in Robot Soccer," *AAAI Workshop on Multiagent Interaction without Prior Coordination (MIPC 2014)*, no. July, pp. 2–7, 2014.

[6] Y. Zhan, H. B. Ammar, and M. E. Taylor, "Theoretically-grounded policy advice from multiple teachers in reinforcement learning settings with applications to negative transfer," *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2016-Janua, no. 7540, pp. 2315–2321, 2016. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[7] D. P. Kingma and J. L. Ba, "Adam: A Method of Stochastic Optimization," in *International Conference on Learning Representations 2015*, 2015, pp. 1–15.

[8] P. Stone, R. Gan, and S. Kraus, "To teach or not to teach? Decision making under uncertainty in ad hoc teams," *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, no. May, pp. 117–124, 2010.

[9] S. Barrett and P. Stone, “Ad Hoc Teamwork Modeled with Multi-armed Bandits : An Extension to Discounted Infinite Rewards,” *Teacher*, no. May, 2011.

[10] S. Barrett, N. Agmon, N. Hazon, S. Kraus, and P. Stone, “Communicating with unknown teammates,” *Frontiers in Artificial Intelligence and Applications*, vol. 263, no. May, pp. 45–50, 2014.

[11] F. S. Melo and A. Sardinha, “Ad hoc teamwork by learning teammates’s task (Extended Abstract),” *Autonomous Agents and Multi-Agent Systems*, pp. 577–578, 2016.

[12] N. Agmon and P. Stone, “Leading ad hoc agents in joint action settings with multiple teammates,” *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, no. June, pp. 341–348, 2012.

[13] N. Agmon, S. Barrett, and P. Stone, “Modeling uncertainty in leading ad hoc teams,” *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, no. May, pp. 397–404, 2014.

[14] D. Chakraborty and P. Stone, “Cooperating with a markovian ad hoc teammate,” *International Conference on Autonomous Agents and Multiagent Systems,*, pp. 1085–1092, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2485091

[15] S. Barrett and P. Stone, “Cooperating with Unknown Teammates in Complex Domains : A Robot Soccer Case Study of Ad Hoc Teamwork,” *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, no. January, pp. 2010–2016, 2015.

[16] R. I. Brafman and M. Tennenholtz, “On Partially Controlled Multi-Agent Systems,” vol. 4, pp. 477–507, 1996.

[17] P. Stone, G. A. Kaminka, and J. S. Rosenschein, “Leading a best-response teammate in an ad hoc team,” *Lecture Notes in Business Information Processing*, vol. 59 LNBIP, no. May, pp. 132–146, 2010.

[18] N. Cesa-Bianchi and G. Lugosi, *Prediction, Learning, and Games*. New York, NY, USA: Cambridge University Press, 2006.

[19] S. Barrett, P. Stone, S. Kraus, and A. Rosenfeld, “Learning teammate models for ad hoc teamwork,” *AAMAS Adaptive Learning Agents (ALA) Workshop*, no. June, pp. 57–63, 2012. [Online]. Available: http://u.cs.biu.ac.il/~sarit/data/articles/ala2012.pdf

[20] S. Kalyanakrishnan, Y. Liu, and P. Stone, “Half Field Offense in RoboCup Soccer: A Multiagent Reinforcement Learning Case Study,” *RoboCup 2006: Robot Soccer World Cup X*, pp. 72–85, 2007.

[21] M. H. Bowling and P. McCracken, “Coordination and Adaptation in Impromptu Teams,” in *AAAI*, 2005.

[22] K. Genter and P. Stone, “Ad hoc teamwork behaviors for influencing a flock,” vol. 00, pp. 1–8.

[23] S. Liemhetcharat and M. M. Veloso, “Weighted Synergy Graphs for Effective Team Formation with Heterogeneous Ad Hoc Agents,” pp. 41–65, 2013.

[24] S. V. Albrecht and S. Ramamoorthy, “A Game-Theoretic Model and Best-Response Learning Method for Ad Hoc Coordination in Multiagent Systems,” no. January, 2014.

[25] C. M. Bishop, “Mixture Density Networks,” *The effects of brief mindfulness intervention on acute pain experience: An examination of individual difference*, vol. 1, pp. 1689–1699, 2015.

[26] S. Gelly and D. Silver, “Monte-Carlo Tree Search and Rapid Action Value\nEstimation in Computer Go,” pp. 1–33, 2011. [Online]. Available: http://www.cs.utexas.edu/~pstone/Courses/394Rspring11/resources/mcrave.pdf