Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática

Universidade do Porto Faculdade de Ciências

Universidade do Minho Departamento de Informática

2020

David João Apolinário Simões

Aprendizagem de Coordenação em Sistemas Multi-Agente

Learning Coordination in Multi-Agent Systems

MAP i DOCTORAL PROGRAMME IN COMPUTER SCIENCE

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática

Universidade do Porto Faculdade de Ciências

Universidade do Minho Departamento de Informática

2020

David João Apolinário Simões

Aprendizagem de Coordenação em Sistemas Multi-Agente

Learning Coordination in Multi-Agent Systems

Tese apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Informática, realizada sob a orientação científica de José Nuno Panelas Nunes Lau, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e Luís Paulo Gonçalves dos Reis, Professor Associado do Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto.

Apoio financeiro da FCT e do FSE no âmbito do III Quadro Comunitário de Apoio.

o júri / the jury

presidente / president

Armando Domingos Batista Machado
Professor Catedrático da Universidade de Aveiro (por delegação do Reitor da Universidade de Aveiro).

vogais / examiners committee

Luís Miguel Parreira e Correia
Professor Associado da Universidade de Lisboa

Gabriel de Sousa Torcato David
Professor Associado da Universidade do Porto

Luís Filipe de Seabra Lopes
Professor Associado da Universidade de Aveiro

João Alberto Fabro
Professor Associado da Universidade Tecnológica Federal do Paraná

José Nuno Panelas Nunes Lau
Professor Auxiliar da Universidade de Aveiro (orientador)

Luís Paulo Gonçalves dos Reis
Professor Associado da Universidade do Porto (co-orientador)

agradecimentos / acknowledgements

É com muito gosto que aproveito esta oportunidade para agradecer a todos os que me ajudaram ao longo da escrita desta tese. Em primeiro lugar, aos orientadores Nuno Lau e Luís Paulo Reis, por disponibilizarem o seu tempo sempre que necessário e me guiaram neste percurso. Agradeço também à minha família e à Daniela Sousa por todo o apoio dado ao longo dos anos.

Finalmente, quero agradecer aos amigos e colegas de laboratório, que através de conversas, ideias, e debates, tornaram possível a escrita deste documento.

palavras-chave sistemas multi-agente, aprendizagem máquina, coordenação, comunicação

resumo

A capacidade de um agente se coordenar com outros num sistema é uma propriedade valiosa em sistemas multi-agente. Agentes cooperam como uma equipa para cumprir um objetivo comum, ou adaptam-se aos oponentes de forma a completar objetivos egoístas sem serem explorados. Investigação demonstra que aprender coordenação multi-agente é significativamente mais complexo que aprender estratégias em ambientes com um único agente, e requer uma variedade de técnicas para lidar com um ambiente onde agentes aprendem simultaneamente. Esta tese procura determinar como aprendizagem automática pode ser usada para encontrar coordenação em sistemas multi-agente. O documento questiona que técnicas podem ser usadas para enfrentar a superior complexidade destes sistemas e o seu desafio de atribuição de crédito, como aprender coordenação, e como usar comunicação para melhorar o comportamento duma equipa.

Múltiplos algoritmos para ambientes competitivos são tabulares, o que impede o seu uso com espaços de estado de alta-dimensão ou contínuos, e podem ter tendências contra estratégias de equilíbrio específicas. Esta tese propõe múltiplas extensões de aprendizagem profunda para ambientes competitivos, permitindo a algoritmos atingir estratégias de equilíbrio em ambientes complexos e parcialmente-observáveis, com base em apenas informação local. Um algoritmo tabular é também extendido com um novo critério de atualização que elimina a sua tendência contra estratégias determinísticas. Atuais soluções de estado-da-arte para ambientes cooperativos têm base em aprendizagem profunda para lidar com a complexidade do ambiente, e beneficiam duma fase de aprendizagem centralizada. Soluções que incorporam comunicação entre agentes frequentemente impedem os próprios de ser executados de forma distribuída. Esta tese propõe um algoritmo multi-agente onde os agentes aprendem protocolos de comunicação para compensarem por observabilidade parcial local, e continuam a ser executados de forma distribuída. Uma fase de aprendizagem centralizada pode incorporar informação adicional sobre ambiente para aumentar a robustez e velocidade com que uma equipa converge para estratégias bem-sucedidas. O algoritmo ultrapassa abordagens estado-da-arte atuais numa grande variedade de ambientes multi-agente. Uma arquitetura de rede invariante a permutações é também proposta para aumentar a escalabilidade do algoritmo para grandes equipas. Mais pesquisa é necessária para identificar como as técnicas propostas nesta tese, para ambientes cooperativos e competitivos, podem ser usadas em conjunto para ambientes mistos, e averiguar se são adequadas a inteligência artificial geral.

keywords multi-agent systems, machine learning, coordination, communication

abstract The ability for an agent to coordinate with others within a system is a valuable property in multi-agent systems. Agents either cooperate as a team to accomplish a common goal, or adapt to opponents to complete different goals without being exploited. Research has shown that learning multi-agent coordination is significantly more complex than learning policies in single-agent environments, and requires a variety of techniques to deal with the properties of a system where agents learn concurrently. This thesis aims to determine how can machine learning be used to achieve coordination within a multi-agent system. It asks what techniques can be used to tackle the increased complexity of such systems and their credit assignment challenges, how to achieve coordination, and how to use communication to improve the behavior of a team.

Many algorithms for competitive environments are tabular-based, preventing their use with high-dimension or continuous state-spaces, and may be biased against specific equilibrium strategies. This thesis proposes multiple deep learning extensions for competitive environments, allowing algorithms to reach equilibrium strategies in complex and partially-observable environments, relying only on local information. A tabular algorithm is also extended with a new update rule that eliminates its bias against deterministic strategies. Current state-of-the-art approaches for cooperative environments rely on deep learning to handle the environment's complexity and benefit from a centralized learning phase. Solutions that incorporate communication between agents often prevent agents from being executed in a distributed manner. This thesis proposes a multi-agent algorithm where agents learn communication protocols to compensate for local partial-observability, and remain independently executed. A centralized learning phase can incorporate additional environment information to increase the robustness and speed with which a team converges to successful policies. The algorithm outperforms current state-of-the-art approaches in a wide variety of multi-agent environments. A permutation invariant network architecture is also proposed to increase the scalability of the algorithm to large team sizes. Further research is needed to identify how can the techniques proposed in this thesis, for cooperative and competitive environments, be used in unison for mixed environments, and whether they are adequate for general artificial intelligence.

Contents

Chapter 1

Introduction

Multi-agent systems (MAS) are composed of multiple entities interacting with each other and the environment. A team of distributed agents can solve problems that would otherwise be difficult or even impossible for monolithic systems. A large number of MAS examples can be found, such as robotic or swarm navigation [1, 2, 3], distributed target tracking [4, 5, 6, 7], traffic monitoring [8, 9], spacecraft and satellite formation [10, 11], economics and competitive negotiation environments [12, 13], distributed sensor networks [14, 15], autonomous robots [16, 17], or games [18, 19, 20, 21, 22, 23]. MAS form the basis of most complex systems around us, and while MAS research typically focuses on software agents, it also encompasses robots or even humans.

The goal of MAS research is to emulate the solutions already found in biology and psychology and create autonomous agents that can interact with complex systems. Agents, like humans and other life forms, act with limited capabilities upon local knowledge, and are capable of cooperating to reach a desirable global outcome in cooperative environments. MAS research is focused on finding the optimal individual agent's policy to reach the global objective of the system [24]. It uses tools from the fields of game theory, biology, and artificial intelligence, namely planning, reasoning methods, search methods, and machine learning, in order to achieve coordination.

Coordination is considered a key characteristic of MAS, and an agent's capability of coordinating with others constitutes one of its major qualities [25]. In cooperative environments, coordination consists on harmonizing the interactions of multiple agents, such that a global plan can be carried out. The global plan, possibly composed of the sum of each agent's individual actions, will ideally fulfill the agent' individual goals or the global objective of the MAS, as efficiently as possible. In competitive environments, coordination consists on finding the best strategies that complete an agent's own goals, while also avoiding being exploited by adversaries. Eventually, agents may converge to an equilibrium state where changing their policies will allow others to take advantage of them and eventually worsen their performance.

1.1 Motivation

Recent advancements in deep reinforcement learning have achieved great results in highly complex single-agent environments [26, 27]. These improvements have stemmed from a recent increase in hardware capabilities, the re-emergence of artificial neural networks as universal function approximators, and the development of general reinforcement learning algorithms.

Neural networks can generalize to new unseen states, and thus can handle complex environments without needing to explore the environment's complete state-space [26]. They may also be viable for transfer learning, through the use of appropriate generalization techniques [28, 29, 30]. However, most deep learning algorithms are computationally expensive and sample-inefficient, requiring millions of interactions with the environment to converge to adequate policies. The problem becomes more evident when only commodity hardware is available, on which some algorithms would take years to achieve state-of-the-art performance, or when training environments are not performance-oriented and become performance bottlenecks.

Despite the success of single-agent approaches, various research efforts [31, 32, 33] have shown that achieving coordination in MAS remains a complex challenge with open questions. A popular technique for learning in MAS is to apply single-agent reward-based learning algorithms to each independent agent and demonstrate that successful policies can be learned with implicit coordination [34]. However, theoretical convergence guarantees offered by single-agent algorithms are lost, as the vast majority of reward-based learning algorithms assumes a stationary environment [35]. In a MAS, since agents must take into account the remaining agents' policies, which can also adapt their behavior, and agents face a moving target problem in a non-stationary environment. Another solution to this consists on the use of a central entity that controls all agents simultaneously, effectively making the environment single-agent [36]. Despite this, many environments require agents to be executed in a decentralized manner, independently with only their own local observation of the environment. Not only that, but the complexity of the joint-action-space grows exponentially with the amount of agents in the environment, so this solution is not scalable.

Therefore, when independent learning agents form the basis of the MAS, each agent must handle all the complexity that exists in a single-agent environment, as well as the additional issues that arise in the multi-agent paradigm. This includes the underlying environment's complexity, its partial-observability, its high-dimensional action-spaces, its non-stationarity, the possibility of exchanging information with other agents, the structural credit assignment problem (where each agent must estimate how well it contributed to the task's completion), and the convergence to an equilibrium of rational policies.

In cooperative environments, communication is a flexible and general method to achieve coordination, dependent on inherent communication constraints of the MAS, which allows agents to share low- and high-level information [37, 38, 39, 40, 41]. This helps compensate for the partial observability of the environment, reducing the complexity of the task, regardless of whether the communication protocol was hard-coded [42, 43] or learned by the agents [44, 45]. However, there is no consensus on how best to determine the communication protocol for any given environment. Another solution for cooperative coordination is for a central referee to evaluate the entire team's performance and contribution to the task, stimulating implicit coordination and lessening the structural credit assignment problem. Because this does not allow agents to explicitly share information, they may be unable to handle partially-observable environments.

In competitive environments, agents commonly try to reach the equilibrium that maximizes their own pay-off without being exploited by their opponents. These policies are often stochastic and require a probability distribution over the action-space for each state, whereas many single-agent reward-based learning algorithms can simply exploit a greedy deterministic policy. Many competitive multi-agent algorithms [46, 47, 48, 49, 50] have unrealistic assumptions and require more information than the agent's own local observations and re-

wards. Among those that have been shown to converge to the equilibrium policies with only local information, they are often not general enough [51, 52, 53], have no formal proof of convergence [53, 54], or are biased against deterministic strategies [54]. Algorithms can also focus on opponent modeling to adapt accordingly, but it may lead to a recursive loop where agents sequentially try to adapt to each other's expected response [55]. Most contributions in this area focus on single-state games or environments, rely on reinforcement learning, and are tabular-based, thus becoming unable to handle continuous state-spaces or adapt to new unseen states in complex games.

Our research question is then twofold. Firstly, is it possible to learn high-level coordinated strategies with large numbers of communicating agents in complex partially-observable cooperative environments, using deep reinforcement learning? Secondly, is it possible to learn convergent policies with equilibrium properties in competitive environments that can be learned using deep reinforcement learning in complex environments with continuous state-spaces?

1.2 Objectives

This thesis has multiple goals related to learning coordination in multi-agent systems.

Research is conducted on the use of deep learning algorithms to approximate value and policy functions for complex multi-agent environments, both cooperative and competitive. While using artificial neural networks as non-linear function approximators discards theoretical convergence guarantees, we hope to demonstrate that their benefits outweigh their setbacks. In practical terms, they reduce the complexity of the environment's state-space found in both single-agent and multi-agent systems, by generalizing across similar states. New network architectures that can improve the scalability of deep learning solutions in MAS with large amounts of agents are also evaluated. The adaptation of existing multi-agent algorithms to the deep learning paradigm is also considered.

Methods for achieving coordination in both competitive and cooperative scenarios are researched. In competitive environments, agents are expected to converge to equilibrium strategies based solely on local observations and rewards. Cooperative environments are not so restrictive, and a centralized entity can aggregate information from all agents in the learning phase such that agents converge to coordinated policies. The extension of single-agent deep-learning algorithms with multi-agent coordination techniques is also evaluated.

Inter-agent communication is a flexible and general way of sharing information. Research is conducted on how to include it as part of the learning task. The benefits of hard-coded or self-learned protocols are considered. Agents can then exchange information to improve their coordination, compensate for local observations in partially-observable environments, and take advantage of the distributed environment.

The research topics of this thesis can be summarized as follows:

- Deep learning algorithms for complex environments;
- Deep learning architectures for scalability;
- Adapting tabular multi-agent algorithms to deep learning paradigm;
- Equilibrium methods for competitive environments;
- Centralized coordination methods for cooperative environments;

- Adapting single-agent deep learning algorithms to multi-agent paradigm;
- Learning communication to share relevant information.

Proposals that support, to some extent, the above mentioned properties should prove to be a valuable contribution of knowledge to the scientific community, namely to the MAS and machine learning fields. Tests are conducted in multiple and varied environments, and proposals are evaluated based on their performance, robustness and reliability. Algorithms' source-code, tests, and parameters are published on-line, such that our results can be easily corroborated by others.

1.3 Contributions

The research on multi-agent systems for this thesis led to the publication of multiple scientific papers. These were based on multi-agent extensions of deep learning algorithms, reinforcement learning environments, and improvements to mixed-policy algorithms.

Following reproducibility guidelines, new algorithms described in these publications have had their source-code published in on-line repositories. This allows other researchers to have access to each algorithm's implementation as well as the tests and environments used to evaluate it.

[56] D. Simões, N. Lau, and L. P. Reis, *Multi-agent Double Deep Q-Networks*. In: Progress in Artificial Intelligence - 18th EPIA Conference on Artificial Intelligence, EPIA 2017, Oporto, Portugal, September 5-8, 2017. E. Oliveira, J. Gama, Z. Vale, H. Lopes Cardoso (eds), Lecture Notes in Computer Science, volume 10423. Springer.

This paper focused on the adaptation of a deep learning algorithm to the multi-agent paradigm, through two opposite approaches. Their performance were evaluated, as well as their generality to harder tasks and larger teams. The algorithm's source-code and tests can be found at `https://github.com/david-simoes-93/Multi-agent-Double-Deep-Q-Networks`.

[57] D. Simões, N. Lau, and L. P. Reis, *Mixed-Policy Asynchronous Deep Q-Learning*. In: Third Iberian Robotics Conference, ROBOT 2017, Seville, Spain, November 22-24, 2017. A. Ollero, A. Sanfeliu, L. Montano, N. Lau, and C. Cardeira (eds), Advances in Intelligent Systems and Computing, volume 694. Springer.

This paper focused on the adaptation of several multi-agent tabular algorithms to the deep learning paradigm. The extensions were compared and tested in complex partially-observable environments, which the original algorithms did not support. The algorithms' source-code and tests can be found at `https://github.com/david-simoes-93/Mixed-Policy-Asynchronous-Deep-Q-Learning`.

[58] D. Simões, N. Lau, and L. P. Reis, *Adjusted Bounded Weighted Policy Learner*. In: Robot World Cup XXII, Robocup 2018, Montreal, Canada, June 18-22, 2018. D. Holz, K. Genter, M. Saad, O. von Stryk (eds), Lecture Notes in Computer Science, volume 11374, Springer.

This paper focused on improving a tabular equilibrium algorithm, to address one of its setbacks. It is compared with the original algorithm in different scenarios, matching or outperforming it. The algorithm's source-code and tests can be found at `https://github.com/david-simoes-93/ABWPL`.

[59] D. Simões, N. Lau, and L. P. Reis, *Guided Deep Reinforcement Learning in the GeoFriends2 Environment*. In: 2018 International Joint Conference on Neural Networks, IJCNN 2018, Rio de Janeiro, Brazil, July 8-13, 2018. IEEE.

This paper presented a complex reward-based learning environment, both single- and multi-agent. It also described a set of techniques that can be used to speed up deep learning methods, such as intra-agent parameter sharing, or the use of small incremental rewards to guide training. The environment's source-code can be found at `https://github.com/david-simoes-93/GeoFriends2-v2`.

[60] D. Simões, S. Reis, N. Lau, and L. P. Reis, *Competitive Deep Reinforcement Learning over a Pokémon Battling Simulator*. In: 2020 International Conference on Autonomous Robot Systems and Competitions, ICARSC 2020, Azores, Portugal, April 15-17, 2020.

This paper presented a competitive Pokémon battling environment, both 1v1 battles between two agents. It also described the application of a set of mixed-policy algorithms and how applicable would they be to a complex scenario like this. The environment's source-code can be found at `https://gitlab.com/DracoStriker/simplified-pokemon-environment`.

[61] D. Simões, N. Lau, and L. P. Reis, *Multi-agent Neural Reinforcement-Learning System with Communication*. In: New Knowledge in Information Systems and Technologies, WorldCIST 2019, Galicia, Spain, April 16-19, 2019. Á. Rocha, H. Adeli, L. P. Reis, and S. Costanzo (eds), Advances in Intelligent Systems and Computing, volume 931, Springer.

[62] D. Simões, N. Lau, and L. P. Reis, *Multi-Agent Deep Reinforcement Learning with Emergent Communication*. In: 2019 International Joint Conference on Neural Networks, IJCNN 2019, Budapest, Hungary, July 14-19, 2019. IEEE.

[63] D. Simões, P. Amaro, T. Silva, N. Lau, and L. P. Reis, *Learning Low-Level Behaviors and High-Level Strategies in Humanoid Soccer*. In: Fourth Iberian Robotics Conference, ROBOT 2019, Oporto, Portugal, November 20-22, 2019. M. Silva, J. Lima, L. P. Reis, A. Sanfeliu, D. Tardioli (eds), Advances in Intelligent Systems and Computing. Springer.

[64] D. Simões, N. Lau, and L. P. Reis, *Multi-agent actor centralized-critic with communication*, in "Neurocomputing", 2020.

[65] D. Simões, N. Lau, and L. P. Reis, *Multi Agent Deep Learning with Cooperative Communication*, in "Journal of Artificial Intelligence and Soft Computing Research", 2020. Sciendo.

[66] D. Simões, N. Lau, and L. P. Reis, *Exploring Communication Protocols and Centralized Critics in Multi-Agent Deep Learning*, in "Integrated Computer-Aided Engineering", 2020. IOS Press.

These papers focused on the adaptation of a deep learning algorithm to the multi-agent paradigm through the use of learned communication and a centralized evaluator. Its performance was evaluated in complex environments, which require communication and coordination to be completed, despite said communication channels being noisy. The algorithm's source-code and tests can be found at `https://github.com/david-simoes-93/A3C3`.

[67] A. Abdolmaleki, D. Simões, N. Lau, L. P. Reis, and G. Neumann, *Contextual Relative Entropy Policy Search with Covariance Matrix Adaptation*. In: 2016 International Conference on Autonomous Robot Systems and Competitions, ICARSC 2016, Bragança, Portugal, May 4-6, 2016. IEEE.

[68] A. Abdolmaleki, D. Simões, N. Lau, L. P. Reis, and G. Neumann, *Learning a Humanoid Kick with Controlled Distance*. In: Robot World Cup XX, RoboCup 2016, Leipzig, Germany, June 30 to July 3, 2016. S. Behnke, R. Sheh, S. Sarıel, D. D. Lee (eds), Lecture Notes in Computer Science, volume 9776, Springer.

[69] S. M. Kasaei, D. Simões, N. Lau, and A. Pereira, *A Hybrid ZMP-CPG Based Walk Engine for Biped Robots*. In: Third Iberian Robotics Conference, ROBOT 2017, Seville, Spain, November 22-24, 2017. A. Ollero, A. Sanfeliu, L. Montano, N. Lau, and C. Cardeira (eds), Advances in Intelligent Systems and Computing, volume 694. Springer.

[70] A. Abdolmaleki, D. Simões, N. Lau, L. P. Reis, and G. Neumann, *Contextual Direct Policy Search with Regularized Covariance Matrix Estimation*, in "Journal of Intelligent & Robotic Systems", November, 2019. Springer.

The research on optimization techniques performed within the FCPortugal3D team led to the publication of additional scientific papers. These were based on developing an optimization algorithm that matches the performance of other state-of-the-art black-box optimizers, within a contextual setting, and using it to optimize kick and walking behaviors for agents in the team.

The work performed for the FCPortugal3D team in RoboCup's 3D Simulated Soccer League has also led to prizes in multiple competitions. In 2016, FCPortugal3D ranked first in the Portuguese Roboticos Open 2016 and third in the 20th RoboCup International Competition. In 2017, FCPortugal3D ranked second in the Portuguese Roboticos Open 2017 and seventh in the 21st RoboCup International Competition. In 2018, FCPortugal3D ranked second in the Portuguese Roboticos Open 2018 and third in the 22nd RoboCup International Competition. In 2019, FCPortugal3D ranked third in the Portuguese Roboticos Open 2019 and sixth in the 23rd RoboCup International Competition.

1.4 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 reviews important concepts and background information regarding multi-agent reward-based learning, including single-agent reward-based learning, deep learning, Markov decision processes, and multi-agent learning. It also conducts a thorough analysis of the current state-of-the-art in mixed-policy learning, reward-based deep learning, and multi-agent reward-based learning. Chapter 3 lists adequate test-beds for multi-agent algorithms, including single-state game-theoretic games, complex single-agent environments, and multi-agent environment suits with various differing properties.

Chapter 4 proposes MADDQN, an adaptation of the Deep Q-Networks algorithm (a single-agent deep-learning method) to the multi-agent paradigm, using independent and joint-action approaches. The adaptation is evaluated on its performance and generality to harder tasks and larger teams, on fully-observable environments.

Chapter 5 extends several mixed-policy tabular algorithms for competitive environments to the deep learning paradigm. The original algorithms allow agents to converge to equilibrium strategies with only local information. Their extended versions are evaluated and compared in game-theoretic environments and in a multi-state game with noisy observations, evaluating whether they maintain their convergence properties and how robust they are to hyper-parameter changes. Chapter 6 proposes ABWPL, an extension to the WPL algorithm with a new update rule that avoids asymptotic convergence in specific cases, and maintains WPL's behavior in the remainder of scenarios. The extension is compared against the algorithms described in the previous chapter in a wide set of game-theoretic environments.

Chapter 7 proposes A3C3, a multi-agent deep-learning actor-critic algorithm, where a centralized learning phase allows agents to robustly converge to coordinated policies while simultaneously learning communication protocols. The chapter also describes a permutation invariant network architecture for deep learning algorithms. A3C3 supports distributed execution, partially-observable environments, large teams, and noisy communication, and is robust to hyper-parameter changes. It is evaluated in a large set of multi-agent environments with different properties, outperforming other state-of-the-art options.

Finally, Chapter 8 draws our conclusions and lists future work directions.

Chapter 2

Multi-Agent Reward-Based Learning

There are many successful applications of MAS in the real world. Such systems include a set of autonomous agents with a common environment, independently perceived by each agent, which acts according to its goals. In most situations, agents are required to interact with other agents (e.g., robots interacting with humans or other robots) in order to solve a given problem. MAS research focuses on building a system with multiple independent agents, and how to coordinate them [71]. Agents can have a common goal, and work as team, or conflicting agendas, and work against each other to achieve their own goal.

Multi-Agent Reward-Based Learning (MARL) is the discipline that focuses on models where agents dynamically learn policies through interaction with the environment. Some literature uses the term *reinforcement learning* instead of *reward-based learning*, but Panait et al. [72] have noted that the former term is used in two different contexts: the learning class (comparable to supervised and unsupervised learning); and a family of dynamic programming learning algorithms (such as Q-Learning or Sarsa). To avoid confusion, this thesis uses reward-based learning when addressing the learning class.

Common approaches to MARL are based on the concept of rational agents [73]. Russell et al. [74] define an agent as anything that can perceive and act upon the environment, which includes human beings, animals, robots or even software. An agent that optimizes its behavior based on a measure of performance is called *rational*. The goal of MARL research is to find methods to build autonomous rational agents who operate on local knowledge with limited abilities, but are able to learn and solve complex problems in a system composed of multiple agents [24].

MARL has several advantages over the single-agent counterpart. Parallel computation leads to speedups in the learning phase, as well as scalable and robust execution when agents can exploit the decentralized structure of the task [34]. Agents can also use experience sharing for similar agents to learn faster and better, through communication [75], teaching [76, 77], or imitation [78, 79]. Finally, some MAS tasks require decentralized and independent execution, and cannot be completed from a single-agent perspective.

However, there are many open issues and challenges to MARL [80, 81]. In multi-agent or non-static environments, the theoretical guarantees common in most single-agent RL algorithms are lost. Since multi-agent learning consists on learning a policy in the presence of other agents, which are out of our control, and these other agents may also adapt, then the optimal policy may change as learning is performed. This is known as a *moving target* problem [82]. The definition of an adequate learning goal is challenging due to the trade-off

between having a convergent and stable algorithm against one that can adapt to other agents' behavior. The matter of communication helps decreasing the locality of information for each agent, but there is no consensus about what, how, when and why to communicate. Finally, the fundamental issue of MARL is the coordination problem or, in other words, how can multiple agents coordinate to form an optimal joint behavior.

This chapter now describes the Reward-Based Learning class, compares single- and multi-agent systems, and presents the Markovian properties that formally describe MARL environments. It then shows examples of MAS, their taxonomy, some concepts and definitions, and currently open challenges. Finally, it lists related work and state-of-the-art in the field of MARL.

2.1 Reward-Based Learning

Reward-based Learning (RL) is a sub-field of machine learning, whose goal is to control a system that maximizes a numerical-representation of a long-term objective [83]. There is at least one agent, which behaves as the learner, and the environment with which they interact. Agents select and perform actions on the environment, which then reaches a new state, from which agents take an observation, and possibly a reward associated with that transition. This can be seen in Figures 2.1 and 2.2, for both single- and multi-agent systems, respectively.

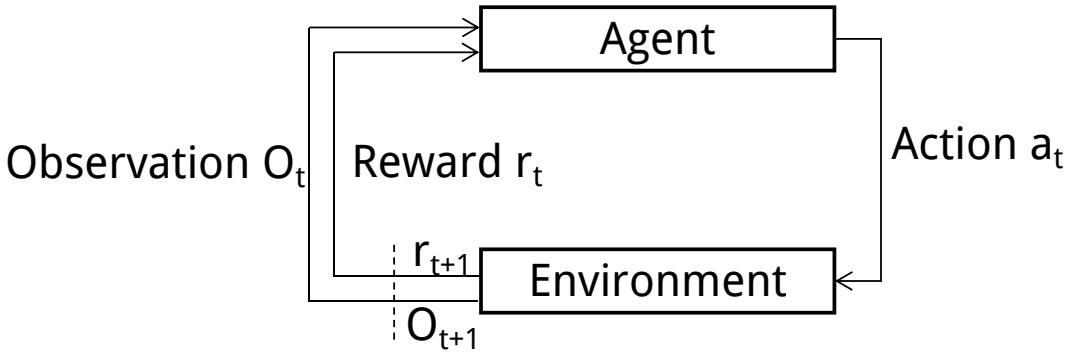


Figure 2.1: The RL cycle for single-agent systems [84]. At time-step t, the agent executes action a_t upon the environment, and obtains reward r_{t+1} and observation O_{t+1}. This process is repeated until the environment reaches a terminal state.

This paradigm has received immense interest in late years, with super-human results on classical games, like *Chess*, *Shogi*, and *Go* [21], and on modern videogames, like *Atari 2600* games [26], *Dota2* [20], and *StarCraft II* [19]. This chapter now describes both single- and multi-agent perspectives on the Reward-Based Learning field.

2.2 Single-Agent Systems

In a single-agent system, there is a single learner trying to find the optimal policy that maximizes the obtained rewards. Two classical approaches have been used in these problems, Value Iteration, and Policy Iteration. Policy iteration relies on evaluating and improving a policy, two processes which are repeated until convergence has been achieved. This is shown in Algorithm 1.

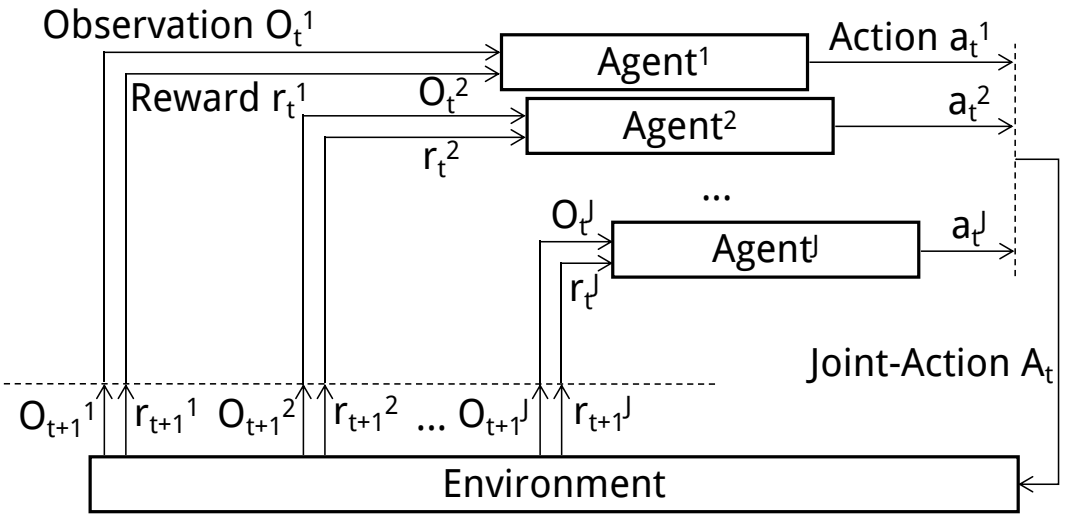


Figure 2.2: The RL cycle for MAS [84], with J agents. At time-step t, each agent j executes action a_t^j upon the environment, composing the joint-action A_t. After this, each agent j obtains reward r_{t+1}^j and observation O_{t+1}^j. This process is repeated until the environment reaches a terminal state.

Value Iteration relies on finding the optimal value function, and extracting a policy from there, as shown in Algorithm 2.

It can be seen that both algorithms require exhaustive sweeps over the complete state-space, which is unfeasible for any complex environment. The environment's state-space is often unknown to agents, who must repeatedly interact with the environment in order to extract an adequate policy.

2.2.1 Markov Decision Process

In single-agent RL, the environment of an agent is commonly described by a Markov Decision Process (MDP). A MDP is a tuple $(S, A, \mathcal{P}, \mathcal{R})$, where S is a finite set of possible states, A is a set of possible actions, $\mathcal{P} : S \times A \times S \to [0, 1]$ is a state transition probability function and $\mathcal{R} : S \times A \times S \to \mathbb{R}$ is the associated reward function.

The state $s_t \in S$ describes the environment at time-step t, and can be changed by the agent through action $a_t \in A$ to state $s_{t+1} \in S$. The transition is based on the state transition probability $\mathcal{P}$, where $\mathcal{P}(s_t, a_t, s_{t+1})$ describes the probability of ending in state s_{t+1} when action a_t is executed on state s_t. Agents receive scalar rewards r_t according to reward function $\mathcal{R}$, where $r_{t+1} = \mathcal{R}(s_t, a_t, s_{t+1})$ measures the immediate effect of an action, and none of its long-term effects. Deterministic models are a specialization of this model, where $\mathcal{P} \to \{0, 1\}$.

Agents behave according to a stationary policy π which describes which action to choose based on a state, $\pi : S \times A \to [0, 1]$. Stationary policies do not evolve over time. The goal of an agent is to maximize, at each time-step t, the expected discounted return $R_t = \mathbb{E}\{\sum_{j=0}^{\infty} \gamma^j r_{t+j+1}\}$, where $\gamma \in [0, 1]$ acts as a future reward discount factor, to decrease the importance of rewards across time, and the expectation $\mathbb{E}$ is taken over the probabilities of the state transitions. The discount factor also bounds the sum to a finite value when $\gamma < 1$.

Input: Set of states S, set of actions A, future reward discount factor γ, state transition probability function $\mathcal{P}(s, a, s')$, reward function $\mathcal{R}(s, a, s')$, randomly initialized value function $V(s)$, and randomly initialized policy $\pi(s)$.

repeat
  {Policy Evaluation}
  repeat
    for all $s \in S$ **do**
      $V(s) \leftarrow \sum_{s' \in S} \mathcal{P}(s, \pi(s), s')(\mathcal{R}(s, \pi(s), s') + \gamma V(s'))$
    end for
  until $V(s)$ converges

  {Policy Improvement}
  for all $s \in S$ **do**
    $\pi(s) \leftarrow \text{argmax}_a \sum_{s' \in S} \mathcal{P}(s, a, s')(\mathcal{R}(s, a, s') + \gamma V(s'))$
  end for
until $\pi(s)$ converges

Output: An accurate value function $V(s)$, and an optimal policy $\pi(s)$.

Algorithm 1: Pseudo-code for the Policy Iteration algorithm [84].

Input: Set of states S, set of actions A, future reward discount factor γ, state transition probability function $\mathcal{P}(s, a, s')$, reward function $\mathcal{R}(s, a, s')$, and randomly initialized value function $V(s)$.

repeat
  for all $s \in S$ **do**
    $V(s) \leftarrow \max_a \sum_{s' \in S} \mathcal{P}(s, a, s')(\mathcal{R}(s, a, s') + \gamma V(s'))$
  end for
until $V(s)$ converges

Output: An accurate value function $V(s)$, and a deterministic policy $\pi(s)$ such that $\pi(s) = \text{argmax}_a \sum_{s' \in S} \mathcal{P}(s, a, s')(\mathcal{R}(s, a, s') + \gamma V(s')$.

Algorithm 2: Pseudo-code for the Value Iteration algorithm [84].

2.2.2 Partially-Observable Markov Decision Process

A Partially-Observable Markov Decision Process (POMDP) is a generalization of the above described MDP, where the agent cannot directly observe the underlying MDP state. Agents maintain a probability distribution over the possible underlying states, which is based on the underlying MDP and on the agents' observations and corresponding probabilities. A POMDP is a tuple $(S, A, \mathcal{P}, \mathcal{R}, \omega, O)$, where S, A, $\mathcal{P}$, and $\mathcal{R}$ have the same definition of the MDP, ω is the set of observations, and $O : \omega \times A \times S \rightarrow [0, 1]$ is a set of observation probabilities.

Like in a MDP, the state $s_t \in S$ describes the environment at time-step t, and can be changed by the agent through action $a_t \in A$ to state $s_{t+1} \in S$. Agents then observe $o_{t+1} \in \omega$ with probability $O(o_{t+1}, a_t, s_{t+1})$. Based on these, agents try to maximize the expected discounted return R_t.

2.2.3 Q-Learning and SARSA

Depending on whether the algorithms learn or use information about the underlying environment model, they can be divided into model-free and model-based algorithms. Model-based algorithm try to model the environment and plan a policy based on that model, while model-free algorithms learn a policy without explicitly modeling the environment, usually being more sample-inefficient.

Q-Learning is a classical example of a model-free algorithm, which learns an action-value policy by exploring the underlying MDP, learning a value-function, and provably converging to an optimal policy. It is shown in Algorithm 3.

Input: Future reward discount factor γ, learning rate η, randomly initialized Q-function $Q(s, a)$, maximum time-step value $T_{\max}$, exploration policy π.

repeat
  Reset the environment and sample initial state s_0
  Time-step $t \leftarrow 0$
  repeat
    Sample action a_t according to exploration policy π
    Take action a_t
    Sample reward r_t and new state s_{t+1}
    $Q(s_t, a_t) \leftarrow (1 - \eta)Q(s_t, a_t) + \eta(r_t + \begin{cases} 0 \text{ for terminal state } s_{t+1} \\ \gamma \max_a Q(s_{t+1}, a) \text{ otherwise} \end{cases})$
    $s_t \leftarrow s_{t+1}$
    $t \leftarrow t + 1$
  until $t > T_{\max}$ or terminal s_t.
until Q converged.

Output: An accurate Q-function $Q(s, a)$.

Algorithm 3: Pseudo-code for the Q-Learning algorithm [84].

Algorithms can be further divided by being on- or off-policy, depending on whether the value function being estimated depends on the policy generating the data or not. While Q-learning is off-policy, algorithms like SARSA, shown in Algorithm 4, are on-policy.

Input: Future reward discount factor γ, learning rate η, randomly initialized Q-function $Q(s, a)$, maximum time-step value $T_{\max}$, exploration policy π.

repeat
  Reset the environment and sample initial state s_0
  Sample action a_0 according to exploration policy π
  Time-step $t \leftarrow 0$
  repeat
    Take action a_t
    Sample reward r_t and new state s_{t+1}
    Sample action a_{t+1} according to exploration policy π
    $Q(s_t, a_t) \leftarrow (1 - \eta)Q(s_t, a_t) + \eta(r_t + \begin{cases} 0 \text{ for terminal state } s_{t+1} \\ \gamma Q(s_{t+1}, a_{t+1}) \text{ otherwise} \end{cases})$
    $s_t \leftarrow s_{t+1}$
    $t \leftarrow t + 1$
  until $t > T_{\max}$ or terminal s_t.
until Q converged.

Output: An accurate Q-function $Q(s, a)$.

Algorithm 4: Pseudo-code for the SARSA algorithm [84].

When the state- or action-space grows very large, function approximation techniques are required to converge to solutions within reasonable time. Models like Artificial Neural Networks can handle high-dimensional state-spaces, and also generalize to new unseen states [26], while Monte-Carlo search methods help explore high-dimensional search-spaces.

2.3 Deep Learning

Deep Neural Networks are powerful non-linear function approximators and have recently become a popular approach to visual domains, such as image classification [85, 86]. In reward-based learning, a network $V(s_t, \theta)$ with weights θ approximates the optimal value function $V(s_t, \theta) \approx V^*(s_t)$ for any state s_t, thus reducing the complexity of the learning problem and generalizing to possibly unseen states. The networks are used as an end-to-end differentiable algorithm, and approximate the value function solely based on the environment's state and the

expected rewards. The universal approximation theorem [87] states that deep neural networks with at least one hidden layer and a non-linear activation function can provably approximate any function with arbitrary precision.

At its core, a neural network is a directed graph where nodes represent neurons and are divided into an input layer, an output layer, and one or more hidden layers [88], as can be seen in Figure 2.3. How many hidden layers are necessary to distinguish between deep and shallow neural networks is not clearly defined. Each edge of the graph between nodes i and j has a weight w_{ij} and each node j outputs value $x_j = \sigma(x_j^-)$, where σ is a predetermined activation function, and $x_j^- = \sum_{i=1}^{I} w_{ij} \times x_i$ is the weighted sum of the node j's I input synapses. These input synapses, in a fully connected layer for example, consist on all the nodes of the previous layer, and possibly an additional node with value 1 called the bias, all multiplied by their corresponding weight.

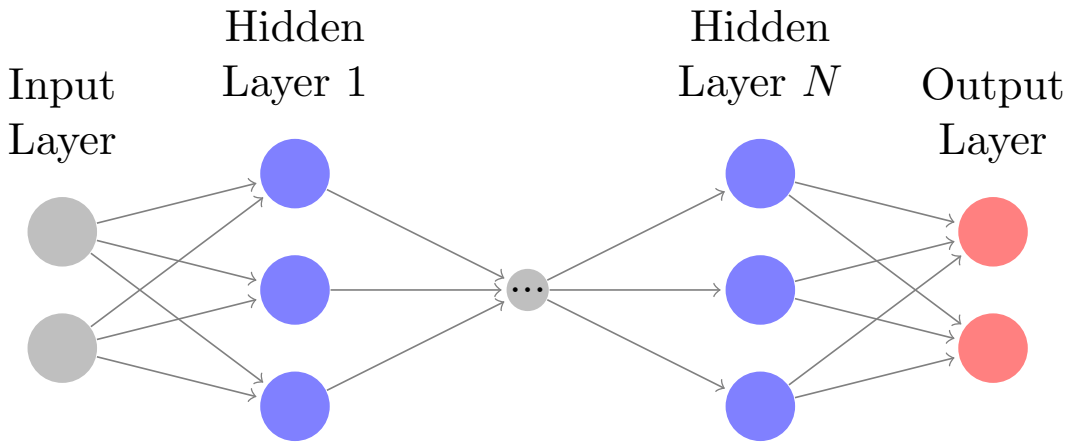


Figure 2.3: An example of a deep neural network with a fully-connected architecture and N hidden layers. Each node is connected to all the nodes in the next layer.

Neural networks are trained through backpropagation, where the weights θ of the network are optimized based on the partial derivatives $\frac{\partial L}{\partial \theta}$ of a loss function L with respect to the weights. They require several hyper-parameters to be defined. These include an amount N of hidden layers, nodes I in each layer, and activation function σ of each layer; a type for each layer, which defines how it is connected to other layers; an initializer, which defines how the initial weights are randomly generated; an optimizer, which calculates the gradients applied to each weight; a loss function, which defines the network's error; a learning rate η, used by the optimizer to define the size of the gradient updates; and a batch size n, which defines the size of batches of samples, thus taking advantage of hardware to speed-up the training. Many of these depend on the problem and are often defined based on intuition.

For layer types, the most common are fully-connected, where all nodes from a layer N connect to all nodes of layer $N+1$; convolutional [85], mostly used in image processing, where a kernel (a small set of weights) processes many smaller portions of the layer in order to capture spacial dependencies; recurrent, mostly used when the network input has temporal dependencies, where the output of a layer N connect to the input of a previous layer $N-i, i \geq 0$; and Long Short-Term Memory (LSTM) [89], a type of recurrent layer, where a state is saved at time-step t and used in time-step $t+1$, and where the formula to save the state is also

Table 2.1: Common neural network activation functions.

Name	Function
ReLU [90]	$\sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ otherwise} \end{cases}$
ELU [91]	$\sigma(x, \alpha) = \begin{cases} \alpha(e^x - 1) \text{ if } x < 0 \\ x \text{ otherwise} \end{cases}$
Logistic	$\sigma(x) = \frac{e^x}{e^x+1}$
Hyperbolic Tangent	$\sigma(x) = \frac{e^x-e^{-x}}{e^x+e^{+x}}$
Linear	$\sigma(x, m) = mx$

Table 2.2: Common neural network loss functions.

Name	Function
Mean Absolute Error	$\frac{1}{n}\sum_{i=1}^{n} \lvert y_i - y_i^* \rvert$
Mean Squared Error	$\frac{1}{n}\sum_{i=1}^{n} (y_i - y_i^*)^2$
Negative Log Likelihood	$-\frac{1}{n}\sum_{i=1}^{n} \log(y_i)$ for all correct classes
Cross Entropy	$-\frac{1}{n}\sum_{i=1}^{n} [y_i^* \log(y_i) + (1 - y_i^*) \log(1 - y_i)]$

optimized through backpropagation. It is possible to extract temporal dependencies with only fully-connected layers by aggregating multiple time-steps as the network input [26].

Non-linear activation functions allow the network to approximate non-linear functions. The most common are described in Table 2.1. Depending on the used functions, different issues may arise, like vanishing or exploding gradients, where gradients in initial layers tend to 0 or to infinity, respectively, or the *dying ReLU* problem, where negative weights have no gradient and cannot be optimized further.

An initializer describes how weights are initially generated for the network. While they are often based on a normal or uniform distribution, the Glorot initializer [92] takes into account the amount of input and output units between layers such that the variance of each layer's weights remains the same. This helps with the vanishing or exploding gradients problems, and allows networks with many hidden layers to be optimized within a reasonable amount of time.

An optimizer is used to define how the network's weights are updated at each step. The most common is Stochastic Gradient Descent [93], where random samples are selected and gradients are computed based on them. Those gradients are then multiplied by a learning rate η and applied to the network's weights θ. The learning rate is a hyper-parameter that depends on the network architecture and problem, and many optimizers [94, 95, 96] use an adaptive learning rate technique.

A loss function L measures the error between a target y^* and the network output y, and is minimized with respect to the network's weights θ. Common loss function are showed in Table 2.2. For continuous unconstrained values (like a Q-function), the mean squared error is a common loss function, while cross-entropy is often used when optimizing probability distributions.

Deep neural networks, however, are prone to over-fitting, and require specific techniques to avoid it, such as dropouts [97], or random experience replay [26]. Their results are also often

difficult to interpret [98], and when the network is not able to approximate a given function, pinpointing the cause of the problem is often not trivial.

2.4 Multi-Agent Systems

The field of Multi-Agent Systems focuses on finding methods for building complex systems with independent and autonomous agents, capable of carrying a desirable global task, despite operating on local knowledge and having limited capabilities [24]. In other words, MAS research takes a description of what a system of agents should do, and transforms it into individual agent behaviors.

MAS approaches the problem with tools and ideas from Game Theory (GT) research (logical decision making, and mixed-strategy equilibria) and Artificial Intelligence research (planning, reasoning methods, search methods, and machine learning). MAS, in comparison with single-agent systems, have the following advantages [71, 73]:

- Parallelization, speedup, and efficiency, particularly in tasks that can be decomposed into several subtasks, which are then handled asynchronously by the agents;
- Fault tolerance and robustness, since the system suffers a gradual degradation as agents fail, instead of completely stopping;
- Scalability, since more agents can be added to the MAS, in order to increase the parallelization of the system;
- Geographical spread-out, when agents are at different locations;
- Better performance over cost ratio, since it is usually cheaper to scale horizontally than scale vertically;
- Simplicity and re-usability, since developing and maintaining modular systems is often easier than monolithic ones.

The inherent complexity of problems in MAS makes hand-tuned solutions overly difficult, as opposed to automated machine learning solutions [72]. These focus on allowing agents to learn on their own how to solve a problem, and have become a popular solution to MAS problems.

2.4.1 Taxonomy

MAS can be classified across several dimensions, and authors have provided different taxonomies [72], including dimensions such as communication topology, range and throughput, team composition and reconfigurability, processing ability of individual agents, agent homogeneity (also known as agent invariance [99]), control architectures, input and output capacities, among others. For a well-organized and flexible taxonomy, this chapter use Vlassis et al.'s [73] categories, Design, Environment, Perception, Control, Knowledge, and Communication, and a final Learning category.

Design

The design category is based on the characteristics of each individual agent. It is related with and encompasses the other categories to some extent. From a design perspective, agents can be either homogeneous or heterogeneous, across several areas: goals, costs of failure and acting, time constraints, and model and action architectures.

Agents can have the same exact goals (cooperative task), completely opposite objectives (competitive task) or simply different rewards (mixed task). Based on the goals, the task can be cooperative, competitive or mixed. Agents can also have static or dynamic goals, which evolve over time, usually based on some condition. Adversarial algorithms can describe their agents as being in a single-agent adversarial environment (because the agent is against all others). However, we follow their most intuitive description, and instead describe agents as being in stationary environments against learning opponents, and governed by multi-agent competitive algorithms.

The cost of failure and acting of agents is a characteristic related with the sampling efficiency of algorithms. While simulated or software-based systems usually have no associated acting cost (aside from time), real-life robotic agents have both maintenance, time and resource costs, which in turn leads to a smaller degree of freedom when learning optimal strategies.

The cost of failure is usually associated with the reward function, in the sense that failing the task will decrease the overall reward. However, real-time or safety-critical systems have failure consequences whose effect extends beyond the reward-function, usually in terms of human lives. This failure cost may not always be modeled in terms of a reward function penalty.

Agents can also have time constraints in their tasks, after which the task's value is decreased or even void. Like the failure cost, time constraints are usually associated with the agent's reward function, but they may not always be modeled in such a way.

The model of the agent comprises both its hardware and software. Regarding hardware, agents can have different robotic shapes, parts, or mechanisms, ranging from small humanoid Nao robots to large tracked robots. This impacts not only their possible set of actions, but also their perception modules. Regarding software, agents may have different decision logic, behavior policies or even structures, such as reactive or deliberative architectures. This impacts their behavior and their knowledge regarding themselves, other agents, or the world.

The action architecture of the agents is related to their physical model and describes the set of actions of the agent. The same hardware model usually implies the same action model. The opposite, however, is not true, since different hardware models (such has having 4 and 5 wheels) may have identical action models (walking forward and backwards). The action architecture may also be discrete (common in reward-based learning) or continuous (common in physical control tasks).

Environment

The environment category is based on the characteristics of the world and the team of agents.

The agent team has a specific number of agents, which can be static (always the same number of agents) or varying (agents can enter and leave the world at any time). Agents in the team also have specific starting location definitions, such as randomized, identical, or agent-based.

There are two environment types, static and dynamic. A static environment, which is only changed through agent interaction, is assumed in the vast majority of the literature, since it allows single-agent RL algorithms to provably converge. Dynamic environments, on the other hand, can behave against the agents in a competitive manner during the learning phase, which causes no optimal strategy to be found by the agents. In MAS, since other agents' actions (which may not be known) are interacting with the environment, a static environment for the team is not static for each individual agent.

Environments can also be classified based on their observability. Agents can either perceive the full environment state (usually only in simulated or abstract environments), or a partial environment observation. The observation is usually incomplete when compared with the environment state.

Perception

The perception category is based in the input modules of the robot and encompasses the aspects of any observation the agent can make about its surroundings. This includes any input regarding environment state, the other agents' states and its own state.

As discussed in the previous section, agents can perceive complete environment states or their corresponding partial observations. With partially-observable environments, agents may have to rely on others' knowledge. In a POMDP, agents need to maintain a probability distribution about the possible states, based on their observations and their probabilities [34]. This raises issues related with sensor fusion (how to optimally combine the agent's perceptions) and decision making under partial observability (which can be an intractable problem).

Regardless of whether agents observe the environment's state or a partial observation, they can also perceive local perspectives with spatial (at different locations), temporal (arriving at different times) and semantic (with different interpretations) differences. In order to obtain a global perspective, an additional logic step is necessary to merge information with spatial and time differences from other agents.

Examples on the combinations of global/local perspectives and fully/partially observable environments are shown in Table 2.3.

Table 2.3: Examples of environments with different observation and perspective properties.

	Local Perspective	Global Perspective
Fully Observable	Multiagent Particle Envs [23]: agents perceive the game state in relation to themselves	Chess: agents perceive the game state from a global perspective
Partially Observable	Ciber-Mouse [18]: agents sense a small area around themselves	StarCraft II [100]: agents observe revealed parts of the map from a global bird's-eye view

The observed states may also differ to each agent, in the sense that each perceptor usually has an associated noise. Coordination between agents with noisy beliefs can be improved by using communication to increase the consistency of information [101].

Control

The control category relates to how the agents are controlled in the MAS. Agents can be fully autonomous and distributed (each agent controls itself), or, on the other extreme, human-assisted through a central entity. We are interested in the former, where distributed agents are fully autonomous and must learn how to coordinate. Control methods depend particularly on the task type (cooperative, competitive, or mixed), and they tackle the coordination problem as a means to complete a task. Coordination problems commonly include resource management, where actions are interdependent due to limited resources, and schedule coordination, where time and agents are themselves resources, and require a coordinated effort from a MAS.

Coordination can be explicit or implicit [102, 103]. Explicit coordination relies on information sharing through communication of beliefs [39], objectives [40], or intentions [41], to other agents. Beliefs are world state information, objectives (or goals) are high level task objectives, intentions are low level objectives to achieve a goal (they may change while the goal usually remains the same).

In the implicit form of information sharing, agents interact with the environment, and use knowledge about the capabilities of other agents [104] to achieve the desired collective performance. Information is shared through intelligent perception (observing the other agents to interpret their intentions), active interaction (being sensed, pushing agents, or changing their state), or stygmergy (environment interaction). Stygmergy falls under the active category (changing the environment so its sensing is different for other agents), and passive stygmergy (changing the environment so that it behaves differently for other agents).

Though explicit and implicit coordination each have their own benefits and weaknesses, the less an agent depends on shared information, and the more flexible it is when faced with on-line problem-solving and coordination knowledge, the better it can adapt to changing environments [102]. In fact, the combination of implicit coordination with beliefs exchange has been reported to yield better performance than explicit coordination with intentions communication alone [105], on scenarios with communication loss.

Both these coordination methods can rely on social conventions (or roles) to simplify the problem. Roles can either be static, if they are maintained throughout the task, or dynamic, if they can change according to the situation. If all roles are equal and static, the agents are homogeneous; otherwise, if there is some ordering of agents, the agent structure is hierarchical. A particular case of the hierarchical structure is when there is a single leader, which can act as a central decision unit. Roles may also be very specific or general, depending on the agent capabilities, and may be used to assign an order of importance to agents to break ties.

Knowledge

The knowledge category is related to how aware are agents of the task, themselves, and the remaining agents. It includes the definitions of common knowledge, domain knowledge, and agent knowledge.

Just like with perception, knowledge can be specialized or redundant, based on whether there is common knowledge between the agents or not. The fact p is common knowledge (also known as shared knowledge) if everybody knows p, and everybody knows that *everybody knows* p, and everybody knows that *everybody knows that everybody knows* p, infinitely [41].

Domain knowledge is *a priori* knowledge about the task or the environment. It can

be represented as initial solutions or predefined Q-functions to solve a task in model-free approaches [67], and is the basis for model-based approaches, common in the game theoretic perspective [41]. Informative reward functions can also reward promising behaviors rather than only the achievement of the goal [34].

The agents of the system can also be modeled in terms of states, goals, possible actions and knowledge. This knowledge about the remaining agents can be model-based, where agents can learn the allied and opponent's strategy and devise a best response, or be model-free, where agents learn a strategy of their own that does well with team members against opponents, without explicitly learning the allied or opponent's strategy.

With homogeneous agents, modeling is trivial; all agents behave and know exactly the same about each other. With heterogeneous agents, modeling can be done with communication (if agents are honest and understand each other), but it is not a feasible approach, since enemies are not expected to communicate their strategy in common scenarios. Without communication, modeling can only be done through observation. One of the earliest examples of this is Fictitious Play [106], from the Evolutionary Game Theory environment, where different opponent strategies are counted and the opponent is assumed to choose any of the strategies with a given probability (a mixed strategy). The probabilities are estimated based on the frequency of each strategy, and a corresponding counter-strategy is chosen. When agents are non-cooperative, these knowledge sharing techniques and deductions might not be accurate, or even possible. Agents must also consider the other agent's knowledge in their decision making (which may become a recursive problem, where every agent knows a fact, every agent knows that every other agent knows this fact, and so on).

Finally, knowledge can also be extracted and incorporated off-line. After each game, some systems allow agents to extract global information, and enemy agents' actions and strategies, through game replays. Other systems allow agents to fully communicate in off-line situations while acting autonomously with little to no communication during the task. These are known as Periodic Team Synchronization (PTS) systems [40], and allow shared knowledge to be defined.

Communication

The communication category encompasses all communication related characteristics. At one end of the spectrum, agents cannot share information in any way. At the other, agents can communicate all information instantly to all agents (similar to a hive-mind). However, communication usually has a size and spatial limit, as well as delays and reliability issues. When self-interested agents interact, information may be purposely fake. With cooperative agents, communication of beliefs [39], objectives [40], or intentions [41] is a strong coordination mechanism, based on information sharing.

Communication is usually blackboard- [107] or message-based [108]. Blackboard communication uses a common information space (the blackboard) to make information available to all agents without the need for direct communication between them. It is a low-overhead and simple architecture, with central and distributed models. The central blackboard is a unique information space for all agents, which may become a performance bottleneck. The distributed blackboard has several sub-blackboards, each handling a group of agents. Agents are organized according to some algorithm and communicate with members of the same category through their respective sub-blackboard.

Message-based communication, on the other hand, relies on direct communication, by

sending messages from agent to agent. If specific targets can be chosen as the recipients of the message, communication is targeted. If messages are sent to all other agents within range, communication is broadcast. Lau et al. [38] identify four main communication areas, through which to model message-based broadcast algorithms:

- What to communicate?
- When to communicate?
- Who should be heard at each time?
- How will received messages affect player's behavior?

The communication protocol, or language, is another part of the communication taxonomy. Some systems have no defined protocol, which means an optimal one can be directly learned by the MAS, while others have a rigid protocol defined [42, 43], which may lead to inefficient message exchanges but decreases the complexity of the learning task. Even when the protocol is learned, proposals range from learning *tabula rasa* [44, 45] to deriving languages from symbol alphabets [109, 110], which may be simpler and still offer enough flexibility for agents to learn efficient communication.

The category also includes the cost of communication (both in terms of resources and time), the range (agents may not be able to communicate at long distances), the reliability (messages can be lost or corrupted), security (messages can be tampered with), conflicts (agents may not be able to talk at the same time), the size of the message, its delay, among others.

Learning

The learning category encompasses details regarding the optimization algorithms used to learn optimal strategies and behaviors of the MAS. The used algorithm is one of the main features of the category, and falls under one of the previously described types (direct policy search techniques, reinforcement learning algorithm, or a game theoretic algorithm). The homogeneity of the learning algorithm can also be considered (e.g., all agents learn with the same algorithm), although there seems to be no obvious advantages in heterogeneous learning algorithms.

Experience sharing has been a popular approach to allow speed-ups and take advantage of the distributed setting of MARL. Agents can exchange information through communication [75], skilled agents may serve as teachers for the learner [76], or learning agents may imitate skilled agents [78]. The latter two options are only appropriate if there is already an agent with a (near) optimal policy whose knowledge cannot be shared off-line.

The learning model also determines the generality of the learned policy. If agents from a large population are randomly matched and learn in response to the expected play within the population, then a good general policy may be learned. On the other hand, if a fixed set of agents repeatedly interact with one another, then an over-fitted policy may be learned, which will likely behave better against that specific set of agents, but worse against the overall population.

Algorithms can be classified based on how they cope with the non-stationarity of the environment. Hernandez et al. [31] consider five categories.

- **Ignore** - Algorithms that ignore the non-stationary behavior of the environment. If other agents change their behavior, learned policies with these algorithms are no longer optimal.
- **Forget** - Algorithms with the *convergence* property that adapt to the changing (non-stationary) behavior of other agents. They continuously learn (and forget), adjusting their policies to cope with other agents' behaviors.
- **Respond to Target** - Algorithms that know (or assume) behaviors from other agents, and adapt accordingly. However, if assumptions do not hold, these algorithms provide restricted adaptability suboptimal policies.
- **Learn** - Algorithms that model other agents and use that model to derive optimal policies. These algorithms do not consider strategic behaviors of other agents, which may reason about them.
- **Theory of Mind** - Algorithms that assume that other agents are performing strategic reasoning about them, and react accordingly. This reasoning can become recursive *ad infinitum*, and computing optimal policies can be impossible.

Finally, the goals of the learning algorithm is taken into account. If agents are learning against each other, they may not reach a stable policy, but instead adapt and evolve *ad eternum*. Stability and adaptation (or no-regret) are commonly used as algorithm criteria, where stability essentially means the convergence to a stationary policy, and adaptation ensures that performance is optimal against stationary opponents [82, 111, 112]. A different set of criteria is targeted optimality, compatibility, and safety, where targeted optimality is adaptation against a specific class of opponents (stationary, for example), safety implies the algorithm achieves at least a minimum reward value, and compatibility means that the algorithm reaches an optimal Nash equilibrium in self-play [113].

2.4.2 Multi-agent Markov Decision Process

The generalization of a MDP to the multi-agent case is commonly known as a Multi-agent Markov Decision Process (MMDP), or, in game theoretic literature, as a Stochastic Game [34].

A MMDP is a tuple $(S, A^1, \ldots, A^J, \mathcal{P}, \mathcal{R}^1, \ldots, \mathcal{R}^J)$, where S and $\mathcal{P}$ hold the same meaning as before, J is the number of agents, $A^j, j = 1, \ldots, J$ is a set of possible actions for agent j, A is the joint action set $A = A^1 \times \ldots \times A^j$, and $\mathcal{R}^j : S \times A \times S \to \mathbb{R}, j = 1, \ldots, J$ is the associated reward function of agent j.

The state transitions are the result of the joint-action $a_t \in A$ at time-step t. The joint policy π_t is gathered from the policies $\pi_t^j : S \times A^j \to [0, 1], j = 1, \ldots, J$. If $\mathcal{R}^1 = \ldots = \mathcal{R}^j$, all the agents have the exact same goal and the MMDP is fully cooperative. If $\exists j \neq i : \mathcal{R}^j = -\mathcal{R}^i, i = 1, \ldots, J$ (for all transitions, either agents tie or a single agent wins), the agents have opposite goals and the MMDP is fully competitive.

2.4.3 Decentralized Partially-Observable Markov Decision Process

The generalization of a POMDP to the multi-agent case is known as a Decentralized Partially Observable Markov Decision Process (Dec-POMDP). A Dec-POMDP is a tuple

$(S, A^1, \ldots, A^j, \mathcal{P}, \mathcal{R}^1, \ldots, \mathcal{R}^j, \omega^1, \ldots, \omega^j, O)$, where J, $A^1, \ldots, A^j$, $\mathcal{P}$, $\mathcal{R}^1, \ldots, \mathcal{R}^j$, and O hold the same meaning as before, and $\omega^j, j = 1, \ldots, J$ is the set of observations for agent j.

The state transitions are the result of the joint-action $a_t \in A$ at time-step t. The joint policy π_t is gathered from the policies $\pi_t^j : \omega^j \times A^j \to [0, 1], j = 1, \ldots, J$. Agents observe $o_{t+1}^j \in \omega^j, j = 1, \ldots, J$ with probability $O(o_{t+1}^j, a_t, s_{t+1})$. It has been shown that optimal planning in Dec-POMDP is provably intractable [114], and in practical terms, requires approximation methods [115] as a trade-off between accuracy and speed.

2.4.4 Concepts and Definitions

After defining the taxonomy of MARL environments, this chapter introduces some common definitions and concepts of multi-agent learning literature. These include the stability, adaptation, no-regret, targeted optimality, compatibility and safety criteria for learning algorithms, Nash Equilibria and Pareto Optimality concepts from GT, and the definitions of pay-off matrix and mixed strategy.

Nash Equilibria and Pareto Optimality

An important solution in static games, which is often used as a goal for multi-agent algorithms to achieve [34], is the Nash equilibrium [116]. Assume σ_i* to be the best response of agent i to a vector of opponent strategies. A Nash equilibrium is the joint strategy $\{\sigma_1*, \ldots, \sigma_n*\}$ such that each individual strategy is the best response to the others. This describes a balance where no agent can change its strategy as long as the remaining agents maintain theirs. Any static game has at least one Nash equilibrium.

The Nash equilibrium is often associated with the Pareto optimality principle. A joint strategy $\{\sigma_1*, \ldots, \sigma_n*\}$ is Pareto optimal if there is no other joint strategy $\{\sigma'_1*, \ldots, \sigma'_n*\}$ such that the reward $R(\sigma'_i) \geq R(\sigma_i)$ for all agents i and $R(\sigma_j) > R(\sigma'_j)$ for one agent j. In other words, the strategy is Pareto optimal if there is no other strategy that increases the reward of at least one agent j without damaging the rewards of any other agent i. In many strategic games, a Nash equilibrium is not Pareto optimal, and vice-versa (such as the Prisoner's Dilemma). However, in identical pay-off games (fully cooperative games whose rewards are identical for all agents), all Pareto optimal solutions are Nash equilibria [117]. When optimizing multiple objectives, the Pareto Optimality is usually used as a convergence criterion [118].

Stability, Rationality, and No-Regret

Bowling et al. [82, 111, 112] define three criteria for algorithms to achieve: stability and either adaptation or no-regret.

Stability, also known as *convergence to equilibrium* or *equilibrium learning*, is the requirement that algorithms converge to an equilibrium [48, 119]. In other words, the agents' strategies should eventually stabilize in a coordinated strategy. Algorithms focused on stability are typically unaware and independent of the other learning agents. Reaching Nash equilibria is commonly used as a stability goal. Opponent-independent learning is related with stability, since algorithms converge to a strategy that is part of an equilibrium solution regardless of what the other agents are doing [34].

Rationality, also known as *adaptation* or *best response learning*, is the requirement that agents converge to a best response when other agents remain stationary. Algorithms that focus

on adaptation are aware to some extent of the other agents' behavior and usually model it in some manner to keep track of their policies [120, 121]. In extreme cases, if stability concerns are disregarded, algorithms are only tracking and adapting to the behavior of the other agents. Opponent-aware learning is related with adaptation, since algorithms learn models of the other agents and react to them using some form of best response [34]. If all agents in a system are stable and adaptable, they naturally converge on a Nash equilibrium.

An alternative to *rationality* is the *no-regret* concept, present in the GIGA-WoLF algorithm [112], where the agent's expected average reward is at least as large as the expected average reward any static strategy could have achieved, for any set of strategies of the other agents. In other words, the algorithm is performing at least as well as any static strategy.

Stochastic Games

Stochastic Games are dynamic games with stochastic transitions played by one or more players, and can be modeled through MDP. Stochastic games can be specialized into stage and repeated games, whose definitions derive from GT [34].

A stage game, also known as a single shot game or static game, is a stateless game, which can be described through a pay-off matrix if it has only 2 players. Fully competitive stage games are known as zero-sum games, since the sum of their agents' reward matrices is a zero matrix.

A repeated game is a stage game that is played repeatedly by the same agents. Agents can use the previous game iterations to gather information about the other agents.

Mixed Strategy

A Nash equilibrium is often not a deterministic strategy, but a stochastic one, also known as a mixed strategy. It maps states to probability distributions over the agents' actions, as opposed to greedy policies, where the maximum reward action is always chosen, and prevents opponents from exploiting a deterministic strategy.

Stochastic strategies are commonly found in non-cooperative stochastic games, and are more relevant in the MAS environment than in the single-agent one, due to the need for some solutions to be expressed in terms of stochastic strategies, in order to maintain balance conditions, such as Nash equilibria. They arise since deterministic strategies can often be exploited by other agents [82].

Pay-off Matrix

A pay-off matrix represents a game between 2 players where the columns represent n strategies of one agent, and the rows represent the m strategies of the other agent, as shown in Table 2.4. In zero-sum games, the rewards for the enemy player are symmetrical to the friendly player's, such that the sum of both pay-off matrix would yield a zero matrix, and hence the name zero-sum game.

A deterministic strategy simply plays action $j \in \{1, \ldots, m\}$ with probability $p(j) = 1$, while a mixed strategy (also known as non-deterministic or stochastic strategy) randomly samples an action according to a probability distribution where at least 2 actions have non-zero probability.

Table 2.4: A pay-off matrix for a 2-player game, where one agent has m actions, and another has n actions. The value a, b represents the points earned by agents when the corresponding actions are taken. Agents keep a probability distribution over actions, represented by $\{\alpha_1, \ldots, \alpha_m\}$ and $\{\beta_1, \ldots, \beta_n\}$, where $\sum_{j=1}^{m} \alpha_j = 1$ and $\sum_{j=1}^{n} \beta_j = 1$.

	β_1	β_2	...	β_n
α_1	1,0	0,-1	...	-1,0
α_2	-1,0	0,1	...	-1,0
...	...	...	...	...
α_m	1,0	0,-1	...	1,0

2.4.5 Challenges

The MARL field has multiple open problems and challenges, some of which inherited from single-agent learning [34]. These include the trade-off between exploration and exploitation, the integration of domain knowledge, problem decomposition, credit assignment and the curse of dimensionality. The MAS perspective also creates challenges, such as an adequate learning goal, non-stationary environments, communication, and coordination.

- The exploration and exploitation trade-off consists on the choice to either try out new actions to measure their effectiveness or exploit actions that are already known to yield a high reward. If the focus on either one is too strong, learning will yield poor results.
- Integration of domain knowledge is usually in the form of problem modeling or adequate initial solutions, which have been shown to increase the learning speed of agents.
- Credit assignment, from a single-agent perspective, is based on how to properly reward agents for their actions when rewards are not immediate, and from a multi-agent perspective, is based on how to properly reward agents who did not contribute equally to the task completion.
- The curse of dimensionality relates to the exponential growth in complexity seen in POMDP and in MAS.
- Adequate learning goals and non-stationary environments are related to the adaptation and stability properties of MARL algorithms.
- Communication and coordination arise as the main challenges to solve in the field.

This chapter now describes the most relevant challenges found and some of their current solutions.

Curse of Dimensionality

Some model-based approaches assume that optimal coordination solutions can be found by applying a convergent single-agent method to the complete state-action space. Coordination would arise from each agent's individual action towards that solution. The curse of dimensionality is defined as the exponential growth of the joint state-action space, when dealing

with high-dimensional state and action spaces, worsened by the fact that each agent adds its own variables to the joint state-action space.

Another common cause for the curse of dimensionality is partial observability. It has been shown that optimal planning under partial observability is provably intractable [114], which limits the scalability of optimal solutions to low dimensional state and action spaces (e.g., low number of agents, simple environments or simple action models) and to a short-sighted planning model (e.g., only a few time steps). Intractable problems require approximation methods to reduce the search space before calculating a solution.

Fuzzy Learning [101] has been proposed as a solution to the curse of dimensionality, in order to reduce the dimension of the search space, while still maintaining enough information for a high performance algorithm. Coordination graphs [122] have been used in order to reduce the complexity of the global Q-function into local Q-functions that only depend on the actions of a subset of agents [123]. In other words, it is simpler to calculate the best local policy with only agents that can influence it, as opposed to calculating the global effects of all agents in the system. Busoniu et al. [124] propose adaptive state focus Q-learning, where agents expand their state space to incorporate states of other agents as needed by the learning task.

Other approaches include the generalization between similar situations and actions through input generalization and state clustering [125], or the use of parameterized function approximators (such as neural networks) [126, 127].

Shaping [128] or layered learning [129] can also be used to reduce the complexity of the task and iteratively find more complex behaviors. Shaping presents simple tasks for the agents and progressively increases the goal difficulty. Layered learning learns low-level behaviors and uses them to learn higher complexity behaviors, in an iterative process, until high-level strategies emerge.

Credit Assignment

The credit assignment problem in single-agent systems, known as temporal credit assignment, is related with how to handle delayed rewards. For example, games that involve dozens of moves like backgammon only reward the agent at the end of the game, in the form of victory. As a result, the reward only affects weakly the distant states that led to it. Another issue is that the reward is equally attributed to all of the moves, regardless of their actual utility (whether they were good or bad moves).

Q-learning, for example, calculates all the future rewards to take into account how good are moves (whether they will lead to a strong position or not), and uses a discount factor $\gamma \in [0, 1[$, which represents how far should future rewards be taken into account.

The credit assignment problem from a MAS perspective, known as structural credit assignment, is related to how players that have not contributed equally to the task completion should be rewarded. While a part of the team was accomplishing the goal, some agent may have been standing idly, instead of helping its team members or disturbing the enemy team. Mataric et al. [125] use two solutions to measure the contribution of each agent to the overall goal, namely heterogeneous reward functions, in which small sub-goals are recognized and used to frequently reward learners, and progress estimators, evaluation metrics relative to a current goal that the agent can estimate. Chang et al. [130] use a causal Kalman filter to represent the reward due to other agents or external factors, and can separate it from the agent's personal reward, in cases where the reward function is deterministic or Gaussian. Other approaches [131] model a value function and compare, for each agent, the difference between the

received reward and the expected reward when doing any other action. This allows an agent to calculate the contribution of their specific action to the overall reward.

Coordination

To find a coordinated joint action, three methods have been commonly used: communication, social conventions and learning [37].

Communication methods are dependent on the inherent communication constraints of the MAS, but are general and flexible methods, and allow agents to share low- and high-level information [38, 39, 40, 41]. They are based on agents informing other agents of state information (and then assuming their policies), or informing other agents of their intentions, both of which are a very human-like approach.

Social convention methods are based on hierarchical orderings of the agents and their actions. Agents calculate, observe, or are informed about the intentions of all higher priority agents before calculating their own. These methods are general and domain-independent, but imply common knowledge among the agents. Coordination is only ensured when the intentions or goals of higher ranked agents are known [34].

Learning-based methods, on the other hand, are model-free and robust approaches, but may not converge or may take an impractical amount of time to converge to a global optimal strategy [102]. They use repeated interactions to learn other agent's behavior and act accordingly. Several problems arise due to the non-stationarity of the environment (since other agents may be adapting their behavior to the learner's actions), which causes theoretical guarantees to be lost [80].

Communication

Communication is one of three methods commonly used to achieve a coordinated joint plan [37]. It has been shown that it is possible for a MAS to learn communication and a policy simultaneously [44].

The content of the transmitted information falls under two main categories: informational (also known as low-level), where world state knowledge, beliefs, useful events and opportunities are shared [38, 39]; and propositional (also known as high-level), where intentions and goals are shared [40, 41].

Lau et al. [38] use informational communication with the principles *Communicate only when you have something important to say* and *Communicate only what is important*. The authors use utility metrics to define the importance of each piece of information, and communicate accordingly. Protocols have been hand-developed for knowledge transmission in message-based communication (such as KIF [132], KQML [42], or COOL [43]), but communication has also been learned from scratch, where the MAS learns its own protocol [44, 45].

In propositional information exchange, research has focused on goal commitment. Agents commit to dynamic roles and goals, this way coordinating to solve the task. The GPGP algorithm [133] allows agents to make commitments to goals, goal characteristics and even to explicitly negotiate, in order to build domain-independent distributed coordination strategies. Castelfranchi et al. [134] define three types of commitments in multi-agent environments: internal (an agent commits to some task), social (an agent commits to helping some other agent fulfill its goal), and collective (an agent commits to filling some role). Stone et al. [40] compare collective commitments to locker-room agreements (a type of periodic team synchronization

model). Authors agree that there must be some sort of negotiation and tie-breaking in goal commitment, in order to achieve agreement.

2.5 Learning in Multi-Agent Systems

There are many approaches to multi-agent coordination, both with hand-made policies and with multi-agent learning techniques. Hand-made policies usually rely on domain-knowledge, and use both situation- and role-based mechanisms [135, 136, 38] to achieve flexibility among cooperative agents. While they can in fact be very successful, they may be prohibitively expensive to craft with more complex environments, where the underlying model may not be fully known or understood.

That being said, even classical machine learning approaches may not achieve successful policies in MAS. Q-learning, which provably converges to an optimal policy with sufficient exploration on single-agent systems, does not handle the non-stationarity of the environment and the conflict of interests between agents, thus losing its theoretical guarantees. However, it has still been widely used in two main variations to learn successful policies in MAS. The Independent Q-Learners (IL) algorithm [36] is possibly the simplest multi-agent reinforcement learning algorithm, in that it consists on having agents learning with single-agent algorithms and having them learn implicit coordination. Because the environment is not stationary, there are no theoretical guarantees of convergence, despite having been shown to achieve successful policies in many different environments [131]. The Joint-Action Learners (JAL) algorithm [36] instead treats the entire team of agents as a single agent whose action space is the joint-action of the team. While theoretical guarantees are kept with this model, it has scalability issues and also may not support decentralized execution (independent agents with local observations may not compute the same joint-action).

Another issue of Q-learning is its deterministic policies, which may be unable to achieve equilibrium strategies. Many multi-agent game-theoretic algorithms learn mixed-policies and have been shown to converge to equilibrium strategies. However, since many of these algorithms are tabular and store state-action representations in tables, without any generalization to unseen states, they suffer from the curse of dimensionality, and require specific techniques to decrease the state- and action-spaces. Some techniques [137, 138] exhibit stronger convergence properties by relying on state-based or agent-based lists and counters, which worsens the problem even further. Many game-theoretic algorithms have also only been evaluated in the context of single-stage games.

Multi-agent deep learning techniques have been proposed to address many of these issues. By using a neural network as a non-linear approximator, algorithms are shown to handle high-dimensional state-spaces [26, 139] and achieve successful policies in complex single- and multi-agent environments. However, they no longer have theoretical proofs of convergence [140], and only a few algorithms [131, 23, 44, 141, 142] take advantage of the properties of a MAS to learn agent policies.

This chapter now provides an overview of relevant or state-of-the-art contributions in the field, including game theoretic algorithms that achieve equilibrium strategies, single- and multi-agent deep reward-based learning algorithms based on implicit coordination, and multi-agent deep learning algorithms that rely on communication. It formally describes the algorithms that we have extended or have based our work on.

2.5.1 Mixed-Policy Learning

Greedy algorithms like Q-learning cannot converge to a strategy able to play a competitive game as simple as rock-paper-scissors. Such games require stochastic strategies, where each action is played with some probability. These can be achieved by stochastic, or mixed-policy, algorithms with the *rationality* and *convergence* properties described in the previous sections. This section focuses on algorithms in the **Forget** category, as described in Section 2.4.1. They consider the non-stationarity of the environment without having unrealistic assumptions or expectations about other agents. They are also the most well-studied algorithms in the literature, the most common and general, and they are often computationally inexpensive.

This section further focuses on algorithms that do not require additional information besides their own actions and rewards. While several algorithms have been proven to converge to Nash equilibria [51, 52], many have assumed knowledge about the underlying game structure or the optimal Nash Equilibrium [46, 47], or the actions performed by other agents and their received rewards [48, 49]. In most cases, the environment model is unknown or too complex, or access to rewards of other non-cooperative agents is not available.

The majority of algorithms are derived from Q-learning [31], and keep track of both Q-values and of a probability distribution in each state. This probability may tend to a pure strategy, where the algorithms become the original greedy Q-learning. They often update their Q-values in the same manner as Q-learning, and introduce different ways of calculating the policy π. WoLF-PHC [51] introduced the *Win or Learn Fast* (WoLF) principle, where different learning rates are used when the agent is winning or losing, a principle also used by GIGA-WoLF [52]. However, both algorithms have shown problems in more complex games, such as Shapley's Game [143]. WPL [54] instead uses a variable learning rate, but has no formal analysis and proof of convergence. EMA-QL [53] uses two learning rates, and has been shown to outperform WPL, despite having some difficulties learning simpler games with many actions and asymmetric probabilities.

When formally defining algorithms, this section maintains the same notations as used above in the Q-learning algorithm. It additionally uses $\pi_t(s_t)$ as the policy at time-step t for state s_t, representing a vector of probabilities of picking each action, and $\hat{\pi}_t(s_t)$ as the average policy at time-step t for state s_t, representing a policy that changes slower than π. The policy learning rate at time-step t is denoted by δ_t, and is sometimes dependent on the current state s_t and action a_t.

A projection function $P(\pi, \alpha)$, with $0 \leq \alpha \leq \frac{1}{|A|}$ and $|A|$ representing the total amount of actions in the policy, is used to project policies into the valid probability space, where each probability p in the distribution π obeys $p \in [\alpha, 1]$. When used as $P(\pi)$, then $\alpha = 0$.

Policy Hill-Climbing

Policy Hill-Climbing with the WoLF principle (WoLF-PHC) [51] introduces a variable learning rate to achieve convergence in games. The WolF principle consists on having a higher learning speed when the current policy is worse than the average policy (representing the equilibrium policy). WoLF-PHC achieves optimal strategies against static players and has been proven to converge in self-play.

At each time-step, the algorithm increments a state counter $C_t(s_t)$, which counts how

many times the state s_t has been visited, and computes the average policy $\hat{\pi}_{t+1}(s_t)$.

$$\hat{\pi}_{t+1}(s_t) = \hat{\pi}_t(s_t) + \frac{\pi_t(s_t) - \hat{\pi}_t(s_t)}{C_{t+1}(s_t)} \tag{2.1}$$

It then chooses a learning rate $\delta_t(s)$ based on whether the current policy $\pi_t(s_t)$ is better or worse than the average policy, and projects its new policy through an added increment $\Delta_t(s_t, a_t)$.

$$\delta_t(s_t) = \begin{cases} \delta_w \text{ if } \sum_{a' \in A} \pi_t(s_t, a') Q_t(s_t, a') > \sum_{a' \in A} \hat{\pi}_t(s_t, a') Q_t(s_t, a') \\ \delta_l \text{ otherwise} \end{cases} \tag{2.2}$$

$$\forall a \in A \ \Delta_t(s_t, a_t) = \begin{cases} -\frac{\delta_t(s_t)}{|A|-1} \text{ if } a \neq \text{argmax}_{a' \in A} Q_t(s_t, a') \\ \delta_t(s_t) \text{ otherwise} \end{cases} \tag{2.3}$$

$$\pi_{t+1}(s_t) = P\left(\pi_t(s_t) + \Delta_t(s_t)\right) \tag{2.4}$$

Generalized Infinitesimal Gradient Ascent

Generalized Infinitesimal Gradient Ascent using the WoLF principle (GIGA-WoLF) [52] keeps track of two gradient updated strategies, one of which is updated faster than the other. *Regret* measures how much worse a policy performs compared to the best static strategy, and GIGA-WoLF exhibits both no-regret and convergence properties.

At each time-step, the algorithm estimates a new policy $\pi^-_{t+1}(s_t)$ and the average policy $\hat{\pi}_{t+1}(s_t)$.

$$\pi^-_{t+1}(s_t) = P\left(\pi_t(s_t) + \delta_t Q_t(s_t)\right), \hat{\pi}_{t+1}(s_t) = P\left(\hat{\pi}_t(s_t) + \frac{\delta_t Q_t(s_t)}{3}\right) \tag{2.5}$$

It then computes the learning rate δ_{t+1}, whose magnitude is larger when the slower strategy $\hat{\pi}_t$ received higher reward than π_t, and changes policy $\pi_{t+1}(s_t)$ in the direction of the positive gradient.

$$\delta_{t+1} = \min\left(1, \frac{||\hat{\pi}_{t+1}(s_t) - \hat{\pi}_t(s_t)||}{||\hat{\pi}_{t+1}(s_t) - \pi^-_{t+1}(s_t)||}\right) \tag{2.6}$$

$$\pi_{t+1}(s_t) = (1 - \delta_{t+1})\pi^-_{t+1}(s_t) + \delta_{t+1}\hat{\pi}_{t+1}(s_t) \tag{2.7}$$

Weighted Policy Learner

Weighted Policy Learner (WPL) [54] has a variable learning rate, and allows the agent to move towards the equilibrium strategy faster than moving away from it. Despite not having a formal proof of convergence due to the non-linear nature of WPL's dynamics, the authors numerically solve WPL's dynamics differential equations and show that it features continuous non-linear dynamics, while experimentally demonstrating it converges in several more complex games.

At each time-step, the algorithm calculates an increment vector $\Delta(s_t)$ from the gradients of the value function $V_t(s_t)$ and uses it to compute a new policy $\pi_{t+1}(s_t)$, where each action must have a non-zero probability α.

$$V_t(s_t) = \sum_{a \in A} \pi_t(s_t, a) Q_t(s_t, a) \tag{2.8}$$

$$\forall a \in A \ \Delta_t(s_t, a) = \delta \frac{\partial V_t}{\partial \pi_t(s_t, a)} \begin{cases} \pi_t(s_t, a) \text{ if } \frac{\partial V_t}{\partial \pi_t(s_t, a)} < 0 \\ 1 - \pi_t(s_t, a) \text{ otherwise} \end{cases} \tag{2.9}$$

$$\pi_{t+1}(s_t) = P\left(\pi_t(s_t) + \Delta_t(s_t), \alpha\right) \tag{2.10}$$

Despite having been shown to converge to mixed strategies in multiple scenarios, WPL is biased against deterministic strategies. In such scenarios, its policy update rate tends to zero, and theoretically only converges in the limit.

Exponential Moving Average Q-Learning

Exponential Moving Average Q-Learning (EMA-QL) [53] features two learning speeds. The algorithm is experimentally demonstrated to converge faster than WPL in several scenarios with a smaller number of episodes, but it also has no formal proof of convergence.

At each time-step, the algorithm simply calculates an increment vector $\vec{\Delta}(s)$ and uses it to compute a new policy $\pi_{t+1}(s)$.

$$\delta_t(s_t, a_t) = \begin{cases} \delta_w \text{ if } a_t = \text{argmax}_{a'} Q_t(s_t, a') \\ \delta_l \text{ otherwise} \end{cases} \tag{2.11}$$

$$\vec{\Delta}_1(s_t) = (u_0, u_1, \ldots, u_{|A|}) \text{ where } u_a = \begin{cases} 1 \text{ if } a = \text{argmax}_{a'} Q_t(s_t, a') \\ 0 \text{ otherwise} \end{cases} \tag{2.12}$$

$$\vec{\Delta}_2(s_t) = (u_0, u_1, \ldots, u_{|A|}) \text{ where } u_a = \begin{cases} 0 \text{ if } a = \text{argmax}_{a'} Q_t(s_t, a') \\ \frac{1}{|A|-1} \text{ otherwise} \end{cases} \tag{2.13}$$

$$\vec{\Delta}(s_t) = \begin{cases} \vec{\Delta}_1(s_t) \text{ if } a_t = \text{argmax}_{a'} Q_t(s_t, a') \\ \vec{\Delta}_2(s_t) \text{ otherwise} \end{cases} \tag{2.14}$$

$$\pi_{t+1}(s_t) = (1 - \delta_t)\pi_t(s_t) + \delta_t \vec{\Delta}(s_t) \tag{2.15}$$

Minimax-Q

The Minimax-Q algorithm [144], targeted at competitive games, minimizes the loss in worst case scenarios. It is also used to measure the largest reward that the agent can be sure to achieve without knowing the actions of the other players. It incorporates enemy actions in its Q-function.

At each time-step, the algorithm checks all possible enemy actions and determines the worst (for the friendly team) possible combination of actions o that the enemy team can

choose. It then uses linear programming to determine the best strategy π the friendly team can use to maximize the game, such that

$$\pi(s_t) = \text{argmax}_\pi \min_o \sum_a Q_t(s_t, a, o)\pi(s_t, a). \tag{2.16}$$

By assuming perfect rationality from the opponents or allies, Minimax-Q falls under the *Respond to Target* category, according to Hernandez et al. [31].

2.5.2 Single-Agent Deep Reward-based Learning

The previously described algorithms are tabular, and thus unable to handle high-complexity environments, or generalize to new unseen states. Even in single-agent reward-based learning, the curse of dimensionality can be found in complex environments. For example, tabular Q-learning is an impractical approach in high dimensional tasks, because the action-value function is estimated separately for each state, without any generalization.

A recent popular solution for single-agent environments has been the use of artificial neural networks as non-linear function approximators, despite the fact that introducing non-linear function approximators invalidates theoretical convergence properties. DQN [26], DDQN [145], and AnQ [139] handle large state-spaces by approximating the Q-function with a neural network. DDPG [140] and A3C [139] approximate both value and policy functions with neural networks, following Actor-Critic approaches.

DQN introduced the concepts of using an *experience replay* along with *on-line* and *target* networks, while AnQ and A3C introduced the framework for distributed asynchronous updates on *global* networks, which allows for horizontal scaling. Both techniques try to stabilize the learning process and break the correlations between samples used to optimize the networks.

This section now reviews relevant state-of-the-art contributions in the field of single-agent reward-based learning. When formally describing algorithms, it maintains the same notations as before, and additionally use θ and ϑ to represent network weights, $d\theta$ for gradients with respect to weights θ, y for optimization targets, T and $T_{\max}$ for current and maximum iterations, and η for the network's learning rate.

Episodic REINFORCE

Episodic REINFORCE [146] is a reward-based single-agent policy gradient algorithm, which uses a function approximator with weights θ to estimate $\pi(s, a)$. This algorithm, along with other REINFORCE algorithms, makes weight adjustments in a direction that lies along the gradient of the expected reward.

At the end of an episode with T time-steps, the weights are optimized for each time-step t by

$$\theta \leftarrow \theta + \eta \frac{\partial \log \pi_t(s_t, a_t) v_t}{\partial \theta}, \tag{2.17}$$

where $v_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$ represents the discounted reward obtained from step t onward.

Deep Q-Networks

The Deep Q-Networks (DQN) algorithm [26] uses a deep neural network with weights θ, known as the *on-line* network, as a function approximator, where $Q(s, a, \theta) \approx Q^*(s, a)$. The

Input: Learning rate η, mini-batch size k, time-step limit $t_{\max}$, maximum iterations $T_{\max}$, future reward discount factor γ, target network update period τ, replay memory $\mathcal{D}$ with capacity N, on-line network with random weights θ, and target network with weights θ^- copied from the on-line network.

for iteration $T \leftarrow 1, T_{\max}$ **do**
  Sample state s_1
  for time-step $t \leftarrow 1, t_{\max}$ **do**
    Reset gradients $d\theta \leftarrow 0$
    Select random action a_t with probability ϵ, otherwise best action $a_t \leftarrow \text{argmax}_a Q(s_t, a, \theta)$
    Execute a_t
    Sample state s_{t+1} and reward r_t
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random mini-batch of k transitions from $\mathcal{D}$
    for transition $i \leftarrow 1, k$ **do**
      Compute target $y_i \leftarrow r_i + \gamma \max_a Q(s_{i+1}, a, \theta^-)$ with target network
      Compute loss $L_i \leftarrow (y_i - Q(s_i, a_i, \theta))^2$ of on-line network
      Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L_i}{\partial \theta}$
    end for
    Update on-line network weights $\theta \leftarrow \theta + d\theta$
    Update target network weights $\theta^- \leftarrow \theta$ every τ time-steps
  end for
end for

Output: A converged network with weights θ to approximate the value function as $Q(s, a, \theta)$.

Algorithm 5: The Deep Q-Networks algorithm using ϵ-greedy exploration.

agent's experience at each time-step (s_t, a_t, r_t, s_{t+1}) is stored in a data-set $\mathcal{D}$, known as an *experience replay*. Random batches of uncorrelated samples are uniformly drawn from the replay memory, and used to optimize weights θ, where the loss $L(\theta)$ is given by the mean squared error between the network and a target y_t, according to

$$y_t = r_t + \gamma \text{max}_a Q(s_{t+1}, a, \theta^-). \tag{2.18}$$

The target y_t is calculated from a separate target network, whose parameters θ^- are updated much more slowly than the on-line network (i.e., copied from the on-line network every τ time-steps). The algorithm is formally described in Algorithm 5.

Mnih et al. [26] use the Deep Q-learning algorithm to learn how to play Atari 2600 video-games, where a deep convolutional neural network converts raw pixels into a state representation. The authors compute a processed state with the most recent sequence of actions and observations, in order to compensate for the partial-observability of the environments. DQN has since been extended with additional techniques, like prioritized sampling [147].

Double Deep Q-Networks

Double Deep Q-Networks (DDQN) [145] is an extension to DQN, whose intuition is to reduce overestimation of Q-values by decoupling action selection and evaluation in the network update. In the original DQN, the calculation of the optimization target y_i can be written as

$$y_t = r_t + \gamma Q(s_{t+1}, \text{argmax}_a Q(s_{t+1}, a, \theta^-), \theta^-). \tag{2.19}$$

The authors replace the calculation of y_t by

$$y_t = r_t + \gamma Q(s_{t+1}, \text{argmax}_a Q(s_{t+1}, a, \theta), \theta^-), \tag{2.20}$$

so that action selection and evaluation are decoupled and chosen by two separate networks. This trivial modification successfully reduces overoptimism, and results in a more stable and reliable learning process.

Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) [140] is an actor-critic algorithm for continuous action spaces, which borrows the *experience replay* and *target* network techniques from DQN. It explicitly requires noise to be added to its state- or action-space in order to encourage exploration, usually based on a Ornstein-Uhlenbeck process [148].

DDPG keeps target copies of both the actor and critic networks, and updates them every cycle with a much lower learning rate η^{-} (unlike DQN, which periodically makes a full copy from the on-line network). Samples are stored in the *experience replay*, from which batches are uniformly drawn to optimize the networks.

Asynchronous n-step Q-Learning

Mnih et al. [139] have shown that asynchronous methods running on multi-core CPUs not only require less specialized hardware than their GPU counterparts, but they also achieve greater results in a shorter amount of time. Asynchronous methods essentially keep a global network which all worker threads asynchronously update and whose weights are periodically copied by each worker into its own network. There is no replay memory from which to draw samples, and each thread provides its own samples, in order to break the correlations between mini-batches. These algorithms can be horizontally scaled by increasing the amount of workers, which increases the amount of samples and updates per unit of time, and speeds up the learning process.

Asynchronous n-step Q-learning (AnQ) [139] is an asynchronous algorithm that relies on *on-line* and *target* networks for stability and asynchronous updates to break sample correlation, as shown in Figure 2.4.

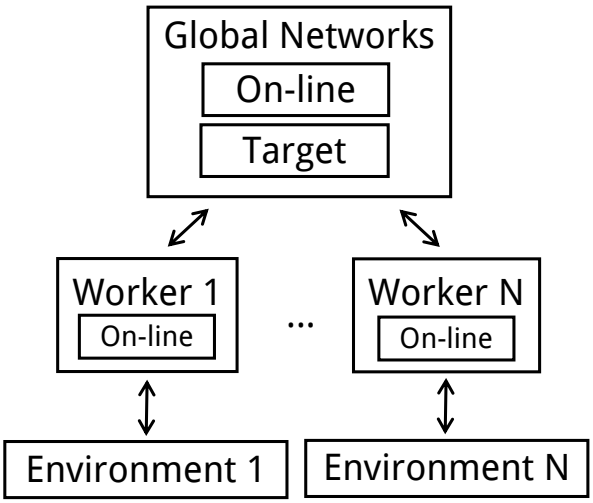


Figure 2.4: The framework for Asynchronous n-step Q-Learning [139], with N workers. Each worker keep a local copy of the on-line network, and interacts with its own environment. Updates are asynchronously performed on the global on-line and target networks.

The algorithm is formally described in Algorithm 6. Each thread computes a gradient of the Q-learning loss. A slowly changing target network is used to stabilize learning, and

gradients are accumulated over up to n time-steps before they are asynchronously applied to the global network.

Input: Globally, shared learning rate η, discount factor γ, target network update period τ, on-line network weights θ, target network weights θ^-, exploration rate ϵ, batch size $t_{\max}$, and maximum iterations $T_{\max}$. Locally, on-line network weights ϑ, and time-step counter t.

$t \leftarrow 0$
for iteration $T \leftarrow 0, T_{max}$ **do**
  Reset gradients $d\theta \leftarrow 0$
  Synchronize $\vartheta \leftarrow \theta$
  $t_{start} \leftarrow t$
  Sample state s_t
  repeat
    Select random action a_t with probability ϵ, otherwise best action $a_t \leftarrow \text{argmax}_a Q(s_t, a, \vartheta)$
    Execute a_t
    Sample state s_{t+1} and reward r_t
    $t \leftarrow t + 1$
  until terminal s_{t+1} or $t - t_{start} = t_{max}$
  Compute target $y \leftarrow \begin{cases} 0 \text{ for terminal state} \\ \max_a Q(s_t, a, \theta^-) \text{ otherwise} \end{cases}$ with target network
  for step $i \leftarrow t - 1, t_{start}$ **do**
    Compute target $y \leftarrow r_i + \gamma y$
    Compute loss $L \leftarrow (y - Q(s_i, a_i, \vartheta))^2$ of local on-line network
    Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L}{\partial \vartheta}$
  end for
  Update global on-line network weights $\theta \leftarrow \theta + d\theta$
  Update target network weights $\theta^- \leftarrow \theta$ every τ time-steps
end for

Output: A converged network with weights θ to approximate the value function as $Q(s, a, \theta)$.

Algorithm 6: Pseudo-code for a worker thread running Asynchronous n-step Q-learning (AnQ) using ϵ-greedy exploration.

Mnih et al. also demonstrate Asynchronous 1-step Q-learning (A1Q), a specific case of AnQ where $n = 1$, shown in Algorithm 7. The algorithm is simpler and closer to the original Q-learning, but it takes longer to converge to successful policies in the demonstrated environments.

Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic (A3C) [139] is another asynchronous algorithm, but it is based in actor-critic methods. Workers keep local copies of both these networks, but asynchronously update their global versions, as shown in Figure 2.5.

A3C is formally described in Algorithm 8, operates in the forward view, and uses n-step returns to update both the policy and the value-function every t_{max} steps or until a terminal state is reached. Actor-Critic methods decouple the value and policy functions into two separate networks. The Critic network with weights θ_v approximates a value function $V(s_t, \theta_v)$ and estimates the expected return at a given state s_t. The Actor network with weights θ_a maintains a policy $\pi(a_t|s_t, \theta_a)$ from which action a_t is sampled for state s_t.

A3C has some advantages over AnQ. It has been shown to achieve better policies with a lesser amount of samples, it supports stochastic policies (as AnQ is greedy), and it requires less hyper-parameters (no target network update period or exploration rate). However, it is also more complex and has a higher computational load with two separate networks to be optimized. This problem can be somewhat mitigated through parameter sharing.

Input: Globally, shared learning rate η, discount factor γ, target network update period τ, on-line network weights θ, target network weights θ^-, exploration rate ϵ, and maximum iterations $T_{\max}$. Locally, on-line value network weights ϑ, and time-step counter t.

$t \leftarrow 0$
for iteration $T \leftarrow 0, T_{max}$ **do**
  Reset gradients $d\theta \leftarrow 0$
  Synchronize $\vartheta \leftarrow \theta$
  $t_{start} \leftarrow t$
  Sample state s_t
  repeat
    Select random action a_t with probability ϵ, otherwise best action $a_t \leftarrow \text{argmax}_a Q(s_t, a, \vartheta)$
    Execute a_t
    Sample state s_{t+1} and reward r_t
    Compute target $y \leftarrow \begin{cases} r \text{ for terminal state} \\ r + \gamma \max_a Q(s_{t+1}, a, \theta^-) \text{ otherwise} \end{cases}$ with target Q-network
    Compute loss $L \leftarrow (y - Q(s_t, a_t, \vartheta))^2$ of local on-line Q-network
    Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L}{\partial \vartheta}$
    $t \leftarrow t + 1$
  until terminal s_{t+1}
  Update global on-line network weights $\theta \leftarrow \theta + d\theta$
  Update target network weights $\theta^- \leftarrow \theta$ every τ time-steps
end for

Output: A converged network with weights θ to approximate the value function as $Q(s, a, \theta)$.

Algorithm 7: Pseudo-code for a worker thread running Asynchronous 1-step Q-learning (AnQ) using ϵ-greedy exploration.

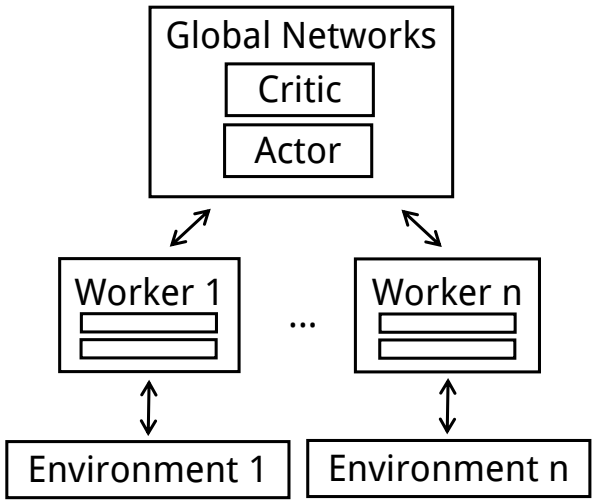


Figure 2.5: The framework for Asynchronous Advantage Actor-Critic [139], with n workers. Each worker keep a local copy of the on-line network, and interacts with its own environment. Updates are asynchronously performed on the global on-line and target networks.

Proximal Policy Optimization

Proximal Policy Optimization (PPO) [149] is a policy search-based method largely inspired on A3C. The policy network uses a new update function which clips policy updates to prevent very large update steps. It has been shown to achieve state-of-the-art results in multiple single-agent environments, with less hyper-parameters and a simpler implementation.

Input: Globally, shared learning rate η, discount factor γ, actor network weights θ_a, critic network weights θ_v, batch size $t_{\max}$, and maximum iterations $T_{\max}$. Locally, network weights ϑ_a, network weights ϑ_v, and step counter t.

$t \leftarrow 0$
for iteration $T \leftarrow 0, T_{max}$ **do**
  Reset gradients $d\theta_a \leftarrow 0$, and $d\theta_v \leftarrow 0$
  Synchronize $\vartheta_a \leftarrow \theta_a$, and $\vartheta_v \leftarrow \theta_v$
  $t_{start} \leftarrow t$
  Sample state s_t
  repeat
    Take action a_t according to policy $\pi(a_t|s_t, \vartheta_a)$
    Sample reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
  until terminal s_{t+1} or $t - t_{start} = t_{max}$
  Compute target $y \leftarrow \begin{cases} 0 \text{ for terminal state} \\ V(s_t, \vartheta_v) \text{ otherwise} \end{cases}$
  for step $i \leftarrow t - 1, t_{start}$ **do**
    Compute target $y \leftarrow r_i + \gamma y$
    Compute cross-entropy loss $L_a \leftarrow \log \pi(a_i|s_i, \vartheta_a)(y - V(s_i, \vartheta_v))$ of local actor network
    Compute loss $L_v \leftarrow (y - V(s_i, \vartheta_v))^2$ of local critic network
    Accumulate gradients $d\theta_a \leftarrow d\theta_a + \frac{\partial L_a}{\partial \vartheta_a}$
    Accumulate gradients $d\theta_v \leftarrow d\theta_v + \frac{\partial L_v}{\partial \vartheta_v}$
  end for
  Update network weights $\theta_a \leftarrow \theta_a + \eta d\theta_a$ and $\theta_v \leftarrow \theta_v + \eta d\theta_v$
end for

Output: Converged critic and actor networks.

Algorithm 8: Pseudo-code for a worker thread running Asynchronous Advantage Actor-Critic (A3C).

2.5.3 Multi-Agent Deep Reward-based Learning

Some of the previously mentioned algorithms have been adapted to the multi-agent paradigm, often based on the IL or JAL approach, or through the means of the *centralized learning, distributed execution* technique. Its point is to augment the training phase with additional information that is not commonly available, without disturbing the execution phase, where agents are run in a distributed and independent manner.

Foerster et al. [150] introduce a set of techniques for multi-agent DQN, such as *inter-agent weight sharing*, where the same network is used by all agents, instead of several separate ones. This speeds-up learning, and agents behave differently through different local observations. The authors also suggest feeding each agent's last action to its input, which helps with partially-observable environments, and disabling the *experience replay* feature of DQN, to avoid outdated samples. This approach does not use communication, but instead is based on implicit coordination. Due to the lack of communication, agents may not be able to account for a partially observable environment.

This chapter now describes the most relevant multi-agent deep-learning algorithms in the literature.

Independent Deep Q-Networks

Tampuu et al. [151] use the *Pong* video-game and adjust the rewarding schemes of the game to range from cooperative to competitive behaviors. The authors use DQN with the IL approach, and report great results, such as cooperative agents learning to hit the ball parallel to the x-axis.

Egorov [152] has also adopted the original DQN algorithm to a multi-agent scenario using

the Pursuit environment and the IL technique. The author demonstrates how the algorithm can generalize to similar tasks, with a different number of agents, or different obstacles. Finally, the author uses a Transfer Learning technique, by transferring the network weights between similar scenarios, to speed-up learning.

Gupta et al. [153] compare Concurrent DQN with centralized DQN, with and without inter-agent parameter sharing. The authors also apply the same approaches to TRPO and DDPG, and conclude that the *experience replay* found in DQN and DDPG negatively impacts training. The authors suggest asynchronous training as a solution to this problem.

Counterfactual Multi-Agent Policy Gradients

The Counterfactual Multi-Agent Policy Gradients (COMA) [131] algorithm is an actor-critic extension that supports distributed execution, but requires centralized training. This *centralized-learning, distributed-execution* framework follows the intuition that algorithms (the value network, in this case) can be augmented with extra information regarding the other agents during the learning phase, while during execution only local information is required, thus allowing agents to run in a decentralized manner. Agents use network sharing for the critic network, so COMA does not support heterogeneous reward functions.

COMA uses the same centralized value network for all agents, with the shared agent observations and their actions as input. The use of agent actions as inputs for the value networks means the environment is now stationary for the critic, even as policies change. COMA addresses the credit-assignment problem by comparing how each agent's action effectively affects the expected value (using the critic network to estimate this).

Since the critic's architecture depends on the amount of agents being trained (as it incorporates their actions and observations), then COMA does not support dynamic amounts of agents. Using the same centralized critic for all agents also means the algorithm does not support different reward functions for different agents (like in non fully cooperative games). Finally, it is unclear how the network scales to large numbers of agents.

Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments

The Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments (MADDPG) [23] is a DDPG extension that also follows the *centralized-learning, distributed-execution* approach. Similarly to COMA, the algorithm has a critic network with the shared agent observations and their actions as input. However, MADDPG uses a value network for each agent, which allows for agents with different reward functions to learn together (any non fully cooperative environment, for example).

MADDPG can also suffer from scalability issues, and does not support dynamic amounts of agents. The approach is based on implicit coordination. The authors propose a suite of multi-agent environments, the Multi-agent Particle Environment (MPE), and compare their work with an IL version of DDPG.

Value-Decomposition Networks

Sunehag et al. present Value-Decomposition Networks (VDN) [154], where agents learn a factorized joint-action value function based on their independent observations, and the sum of each agent's estimation approximates the centralized joint-action function. Agents can communicate by concatenating the output of their layers at some points, thus assuming noiseless

communication without constraints once more. VDN disregards any additional information available from the environment, and limits the complexity of the centralized joint-action function to a simple sum.

Rashid et al. present QMIX [99], a VDN-extension, where each agent's value function is no longer summed to approximate the centralized joint-action function. Instead, an additional *mixing network* is used to combine each individual value function in a more complex manner, which is also able to incorporate additional environment information. QMIX does not use communication between agents, and thus relevant information in partially-observable environments is not shared.

2.5.4 Communication Learning

The previous algorithms have shown good results though implicit coordination. In other words, agents assume the policies of others and act accordingly. The lack of an information-sharing mechanism makes it so that partially-observable environments which require (or where it is beneficial for) agents to share local information are hard or even impossible for such algorithms.

Some authors have recently tackled this problem, by learning communication protocols alongside policies through deep learning. Information exchange allows algorithms to better handle partially-observable environments and improve agent coordination. We do not consider proposals that simply replace the action space by the communication alphabet [155, 150].

CommNet

The CommNet algorithm [44] proposes a single network in the multi-agent setting, passing the averaged message over the agent modules between layers. It uses a fully differentiable communication channel to learn explicit continuous communication between agents, learned concurrently with the agent's policy. The communication channel at time-step t is the summed transmission of messages sent by other agents at time-step $t-1$, and each environment state undergoes multiple communication steps (a value defined a priori).

The authors demonstrate good results in multiple cooperative environments (a traffic simulator, a combat environment, and a Q&A game). The sum of transmissions allows for varying numbers of agents, but multiple cycles of communications among agents is an uncommon assumption. In many environments, actions usually have an equal or higher rate as message transmissions. Not only that, but the amount of cycles with which to communicate with is a hyper-parameter of the algorithm with no intuitive value. The model outputs actions for all agents simultaneously, similarly to the JAL, which does not support distributed execution of the policies. It also makes it unclear how the network handles varying numbers of agents with this shared observation, and how it scales to large numbers of agents. The authors assume perfect communication between agents.

Multi-Agent Bidirectionally Coordinated Network

The Multi-Agent Bidirectionally Coordinated Network (BiCNet) [141] is an actor-critic extension based on Minimax-Q [144]. Using as an input the local view of an agent, and a shared view of all agents, a policy network outputs the action for an agent, and a value network the expected Q-value for that state. Agents are organized in a hierarchical order, and communicate with their neighbors, which allows a variable number of agents to use the same

policy. Through the use of the RNN structure [156], agents have a local memory, and they share information between them while calculating their actions, by sharing the RNN state with their neighbors.

The authors show good results in the StarCraft II [100] environment. However, it is unclear what are the constraints of this sharing methodology and its robustness when communication channels can fail. The use of a shared observation for the policy network is also reminiscent of JAL, which does not support distributed execution of the policies. It also makes it unclear how the network handles varying numbers of agents with this shared observation, and how it scales to large numbers of agents. Finally, requiring the RNN structures in the policy and value networks is a strong requirement, since not all problems require complex structures like RNN, which are notoriously hard to train [157].

Differentiable Inter-Agent Learning

The Differentiable Inter-Agent Learning (DIAL) algorithm [142] uses a Q-network and a neural network that outputs messages through an end-to-end differentiable channel. Agents send messages at each cycle, and these messages are used as inputs for other agents' next cycles, along with their state observations. This approach requires centralized learning, although authors have also proposed an experience-replay based approach that supports decentralized learning [158]. Gradients are then pushed through the communication channels in order to optimize the messages to send.

The authors discretize the sent messages during execution, assume perfect communication between agents, and test their proposal in a limited set of short-horizon environments.

Others

Another end-to-end differentiable learning communication algorithm is found in the methods of Mordatch et al. [110]. Agents learn to communicate by learning a Gumbel distribution, later used on a set of discrete symbols, while simultaneously learning to act in an fully cooperative environment, using a joint reward function. Policies are based in neural networks with recurrent modules and support different numbers of agents. The algorithm requires fully cooperative environments, and the authors also assume perfect communication.

Das et al. [109] propose an algorithm for a one-on-one cooperative game. Using Hierarchical Recurrent Encoder-Decoder neural networks to model policies, and the REINFORCE algorithm [146] for learning a communication policy, agents learn to communicate using a predetermined vocabulary consisting of natural-language symbols. Eventually, one of the agents guesses what image the remaining agent was shown.

D'Ambrosio et al. [45] use neural networks to learn communication in a hive-mind style. Certain neurons are shared among all agents, and the network learns how to set the weights in order to achieve coordination. However, this approach does not allow agents to run in a distributed manner.

Some authors also show how communication can arise in a mix of multi-agent reward-based learning frameworks and supervised learning techniques. By training agents to maximize a goal, and interspersing the training with supervised learning, Lewis et al. [159] demonstrate agents that learn natural language protocols. Using dialogue rollouts, the models plan ahead in bargaining tasks, and fake interest to take advantage of high-value goals.

The Multi-Step, Multi-Agent Neural Network (MSMANN) algorithm [160] uses supervised learning for decentralized agents to learn to imitate a centralized strategy. Agents learn action and communication policies simultaneously during centralized training, despite requiring a JAL strategy to be learned *a priori*. Authors leave a reward-based approach for future work.

2.6 Conclusion

This chapter presented an overview of relevant or state-of-the-art contributions in the field of MARL.

It started by describing game-theoretic multi-agent algorithms that consider the non-stationarity of the environment without having unrealistic assumptions or expectations about other agents, and that do not require additional information besides their own actions and rewards. These algorithms can converge to Nash equilibria in self-play and many are based on Q-learning. However, these algorithms are mostly used in single-state environments, and in fact, due to their tabular nature, cannot handle high-dimensional state-spaces.

Deep learning algorithms have thus been presented as a solution, since they can approximate the value functions and generalize to new unseen states. This chapter described some single-agent algorithms that achieved super-human results in complex environments. However, critical components of these algorithms may not be adequate for MAS. For example, DQN's *experience replay* can lead to outdated samples being considered to optimize a policy. In single-agent environments, the environment is considered stationary, but in MAS, it may lead to policy divergence.

Multi-agent deep learning algorithms thus present a new set of techniques that take advantage of the multi-agent nature of the environment. Some techniques are not general, like interspersing supervised learning techniques during the learning phase, while others are unrealistic, like outputting joint-actions for the team of agents through a central entity. Other techniques, on the other hand, are adequate and flexible for a wide variety of environments, and range from augmenting the learning phase with additional information, to learning differentiable communication protocols.

We can identify multiple shortcomings with the described proposals. For example, few game-theoretic algorithms have been adapted to the deep learning paradigm, which could possibly enable them to handle high-dimensional or continuous state-space environments. Some multi-agent deep learning algorithms are not thoroughly tested in complex environments, or do not scale well to high amounts of agents. Others compensate for partially-observable environments through message passing, but authors assume perfect communication conditions. Many of these shortcomings can be resolved, as it will be shown in the following chapters.

Chapter 3

Applications and Test Beds

Through the course of this thesis, our proposals were tested in multiple environments to evaluate their performance, scalability, robustness, and flexibility. These included single- and multi-agent environments, fully- or partially-observable, with global or local observations, continuous or discrete state and action spaces, pixel- or value-based representations, and single- or multi-state games.

This chapter lists and describes all the environments used to test our proposals. Many of these were specifically developed for the work conducted during this thesis. All environments were developed or adapted to work with our frameworks, using the Gym API [161] in Python 3, with an emphasis on performance. The Gym API is a *de facto* standard in the reward-based learning community, and features an HTTP interface that allows any learning framework to easily use its environments.

3.1 Applications

MAS have a large number of applications in several domains [72], involving, among other, logistics, planning, and constraint-satisfaction with real-time distributed decision making. Due to their complexity and highly distributed features, many of these applications are challenging problems in multi-agent learning.

3.1.1 Cooperative Navigation and Tracking

Cooperative Navigation [1, 2] consists on having a team of agents moving to a certain goal, possibly specific to each agent, usually as fast as possible, and without colliding with obstacles or other agents. A more specific version of this task, known as the Opera Problem [162, 163, 164], consists on having the agents in a small enclosed area and having them leave that area as fast as possible through a small exit. Cooperative Navigation also includes swarm environments [3] with large amounts of agents, where the team must complete a specific goal by combining the efforts of each individual agent.

Cooperative Tracking [4, 5, 6, 7], also known as target observation, consists on having a team of agents keeping several moving targets under observation, being measured by the amount and duration of targets being tracked. Another version of this task, known as Cooperative Surveillance [165, 166], consists on having a team of agents exploring a region without colliding, with the goal of exploring high interest areas or as much as possible of the available

map. Another version, known as multi-agent patrolling [167, 168], involves the continuous exploration of a non-static environment.

3.1.2 Traffic, Vehicle Monitoring, and Transportation

Traffic simulators [169, 170] have recently become popular environments, mainly due to the popularity of autonomous driving fields. Multiple vehicles need to navigate lanes, roads, and intersections, with possibly conflicting goals, and usually with adherence to road rules. On the opposite side of the spectrum, the Vehicle Monitoring task consists on a set of intersections with traffic lights (the agents of the system) maximizing the throughput of vehicles, where the traffic volume fluctuates [8, 9]. A more complex version of this task is where both vehicles and intersections vehicles are agents of the system [171, 172], and can negotiate with each other.

The transportation problem consists on several delivery companies transporting goods between locations. Agents can represent companies or trucks (or both), and customers have different constraints (delivery cost, speed) [173]. A more specific version of this task is the Loading Dock problem, where forklifts load and unload trucks [174]. Another version is the Air Traffic Control task [175, 176], which consists on guiding planes across three-dimensional sectors as fast as possible, without exceeding each sector's maximum capacity. These systems are usually under real-time constraints, regardless of system load, due to their mission critical properties.

3.1.3 Electricity Grid

This task is an electricity distribution management problem where all customers must be supplied with minimal energy losses, while still handling network damage, variable customer demand, scheduled maintenance, or equipment failure [177, 178].

3.1.4 Supply Chains

The Supply Chains [179, 180, 181] task is a classic planning and scheduling algorithm [72], which involves a set of customers requesting different items. The items must be created in a series of steps (with different constraints and costs) and delivered to the customers while minimizing the production costs.

3.1.5 Games

Competitive games are a natural setting for MAS. These include classical board-games, like *Chess*, *Shogi*, and *Go* [21], where two players compete with a structured set of rules, as well as on modern videogames, like *Atari 2600* games [26], *Dota2* [20], Geometry Friends [22], StarCraft II [100], and competitive Pokémon battling [182]. They are complex environments that require reasoning and, in the case of video-games, possible visual processing to extract information from raw pixels.

Other games that have been commonly used in MAS for demonstrating the working of algorithms include the Foraging, Herding, and Pursuit tasks. The Foraging task is a task where agents are tasked with foraging items and bringing them back to specific places. The problem can be solved by a single robot, and multiple agents parallelize the effort [183]. It becomes more complex if good or bad regions to forage changer over time, or if agents are selfish. The Herding task [184, 185] consists on a group of agents (the shepherds) herding

another group of agents (the sheep) into a designated area. The sheep avoid obstacles and the shepherds, but otherwise avoid the herding area or move randomly. The Pursuit task [186], also known as the Predator/Prey game, consists on a team of predators capturing the elements of the team of prey. Prey can either move randomly or actively try to escape the predators, and the environment can be fully cooperative for a team if the opposing team has stationary policies, or mixed if both teams are learning.

3.1.6 Autonomous Robotics

Robotic soccer is one of the most popular MAS domains [16, 71, 72]. Two teams of 5 to 11 agents coordinate in the field and try to score and defend goals. A team's performance is usually measured in goal difference, but metrics such as ball possession time or successful interceptions have also been used. The strong interest in this domain led to the annual RoboCup competition [16], with several different leagues, both simulated 2D and 3D [187], and with real robots [188].

In Keep-Away Soccer, a simpler version of the soccer domain [189], there are three defensive players trying to keep the ball out of reach of a single offensive player. The allowed area for the ball shrinks along time, thus guaranteeing a finite game. A fully cooperative version of this task is the Passing challenge, where agents just pass the ball among themselves as many times as possible over a per-determined interval.

The Ciber Mouse competition [18, 190] is a simulated environment where several agents need to coordinate to solve a maze as fast as possible. Agents need to reach a target area and return to the initial area, and can communicate with restrictions among themselves.

3.1.7 Others

Other applications have been found for MAS, including in the fields of telecommunications (whose agents are nodes in the network and manage it, handle failures and balance link loads) [191, 192], spacecraft and satellite formation [10, 11], economics and competitive negotiation environments [12, 13], or distributed sensor networks [14, 15].

3.2 GeoFriends 2

The Geometry Friends [22] game is a popular simulator developed at GAIPS INESC-ID, with two distinct agents, a circle and a rectangle. The agents have distinct movement patterns, as the circle can rotate and jump, while the rectangle can grow or shrink, and slide. By moving across several 2D scenarios, the agents need to collect all the rewards in the shortest amount of time.

Figure 3.1 shows an example of the scenarios available for the agents. Agents may have specific rewards to collect, or may need to coordinate in order to capture all the rewards. There are scenarios where only one of the agents plays. The environment's multi-agent nature makes it suitable for cooperative algorithms, and it integrates communication between the agents. Its puzzle-like structure makes it complex enough that agents need to reason and need to learn complex policies in order to complete the game. However, despite its advantages, the published environment has a number of properties that make it inadequate to our requirements.

First of all, it is built specifically for the Windows platform, using C# and the .NET framework. It requires the use of Mono, or other .NET implementation, to be run in other

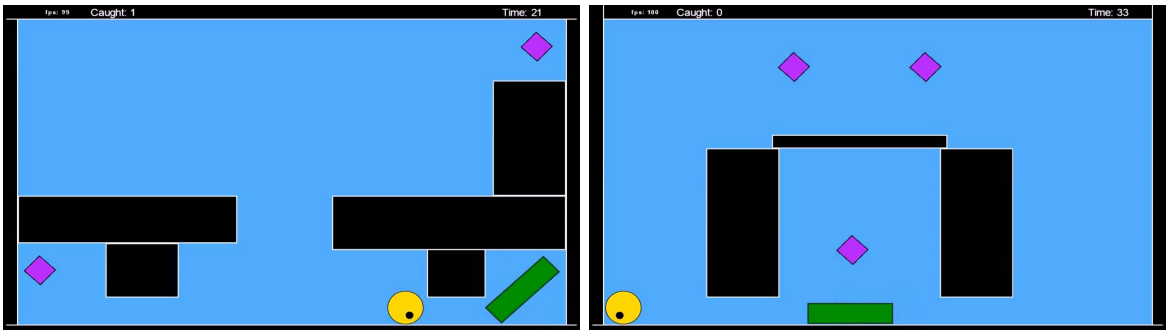


Figure 3.1: Two exemplary scenarios in Geometry Friends, with both the circle (yellow) and rectangle (green) agents. The rewards are represented by the dark gray diamonds.

operating systems, as well as some specific packages for graphical operations. The source code has not been published, so the simulator cannot be optimized or improved by the community. It is also hard to use it with other languages (most high-level languages allow some form of integration of different-language source-code into their own code). Agents are expected to be coded in the original C# language and compiled as DLLs which are later imported by the simulator. A workaround to allow researchers to use different languages and tools for the learning process is to create agents that open communication channels (like sockets), thus being able to communicate with a learning framework developed in another language.

Even with this workaround, the fact that agents cannot be run with specific arguments (as the simulator is the one that runs the agents, they are not called by a learning framework), also implies a set of additional measures to customize agents. This happens in a variety of situations, like when trying to create several parallel workers and have each worker connect to specific channels of the learning framework. Many solutions exist for this problem, ranging from using a global mediator to inform agents where they should connect (additional overhead), to trying several ranges of values (trial-and-error), to running agents from different folders with different configuration files or different source codes (memory inefficient).

While the simulator supports a graphical interface, this interface can be disabled with a command-line argument. In machines where a graphical interface is not deployed to avoid unnecessary resource consumption, applications can only be run in text mode. However, the simulator's graphical interface is not completely disabled with the command-line argument (a button with the word *EXIT* is shown), and it cannot be run unless the application's graphics are piped somewhere (through a remote X Server, if working in Linux).

Regarding the simulator speed, the simulator runs at a fixed 100 asynchronous cycles per second. The agents cannot waste more than 10 milliseconds in each cycle or they will lose sensor information. This is a real-time requirement that makes the environment much more complex, but one that should be optional, as during the training phase, losing cycles due to processing time is highly undesirable, and the learning phase typically has very high processing times (orders of magnitude higher than the deployment phase). A workaround for this issue is to use specific method calls in the agents, which force the simulator to wait, and move all processing to those methods. This is a non-intuitive solution that makes debugging and testing somewhat harder.

The Geometry Friends simulator has a speed command-line argument that controls the

speed at which the simulation is run and the time length of each game cycle. In other words, if the learning framework's processing time were half the environment's cycle time, running at twice the speed would remove excess wasted waiting time. However, the speed argument is closer to a skip-frame argument (where an action is repeated multiple times), than it is to a speed one. The amount of simulator interactions per second is constant, regardless of the speed value. That is, if the simulator speed is doubled, it will still process the same amount of interactions per second in the learning framework, but each interaction behaves as repeating the given action twice. Therefore, if the processing time of our learning framework is faster than the environment's cycle time, the framework is still hindered by the environment's speed, and it is now losing simulation cycles.

We implemented a new version of the Geometry Friends simulator, called GeoFriends2, where the above issues were addressed. It focused on performance and on flexibility, and its code was published as a Gym environment at `https://github.com/david-simoes-93/GeoFriends2-v2`. The environment was also partially published in the IJCNN18 [59]. Figure 3.2 shows exemplary single-agent scenarios.

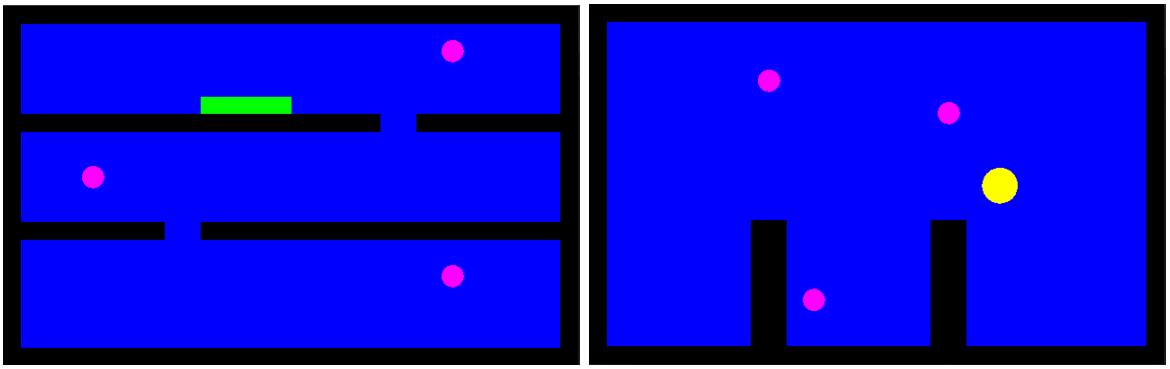


Figure 3.2: Two exemplary scenarios in GeoFriends2, with the circle (yellow) and rectangle (green) agents. The rewards are represented by the pink circles.

GeoFriends2 has no dependencies, aside from Python3, the Gym framework, and the optional PyGame library for display, and agents can either run directly in Python3 or communicate through an HTTP interface. As the code is open-sourced, it is flexible enough to be imported and optimized for any learning framework.

It supports an optional graphical interface, peaks at 1500 cycles per second on an Intel i5-4210U CPU @ 1.70GHz when the GUI is disabled, and the simulator interaction is synchronous, so no cycles are lost.

Agent observations are different for each agent. The circle observes its position and velocity, while the rectangle observes its position, width, and whether it is growing sideways. Both agents also continuously receive the absolute reward positions, and in the first cycle of the simulation, they receive a list of all obstacles' absolute positions and sizes. Another possible state representation is the graphical one, where the learning algorithms process the actual pixels of the simulation's current frame. This representation can be scaled down to decrease the complexity of the environment.

The actions available to each agent differ. Both can stand still and move sideways, but the circle can jump, while the rectangle can decide to grow sideways or not. If the rectangle is not the maximum size, it gradually changes shape. These actions are the same as in the

original environment.

There are some differences from the original environment, however. The Geometry Friends' rectangle can tumble to its sides, which GeoFriends2 forbids, for performance reasons. Keeping the rectangle from tumbling allows the simulator to compute all obstacle detection calculations with simple geometric formulas, and also means it can ignore physics regarding acceleration and angular momentums of the tumble. GeoFriends2 also allows the circle's horizontal speed to change while on air, while Geometry Friends keeps it constant.

GeoFriends2 has also purposefully changed some behaviors which were unexpectedly found in the original environment, such as gravity and bouncing effects when very close to the ground. In Geometry Friends, the circle was never actually on the ground, just infinitely bouncing very close to it, meaning it would have non-linear speed increments without actually jumping. In our environment, a threshold was set that causes the circle to behave as if on the ground, such that bouncing eventually stops when the acceleration is low enough.

Both environments also supports multi-agent interaction and learning. The circle and rectangle act as obstacles to each other and collide, such that one can act like *ground* to the other. GeoFriends2 also supports new teams of agents (like two circles, or two rectangle and a circle), or the implementation of new types of agents. Examples are shown in Figure 3.3, where agents need to coordinate in order to catch all the existing rewards.

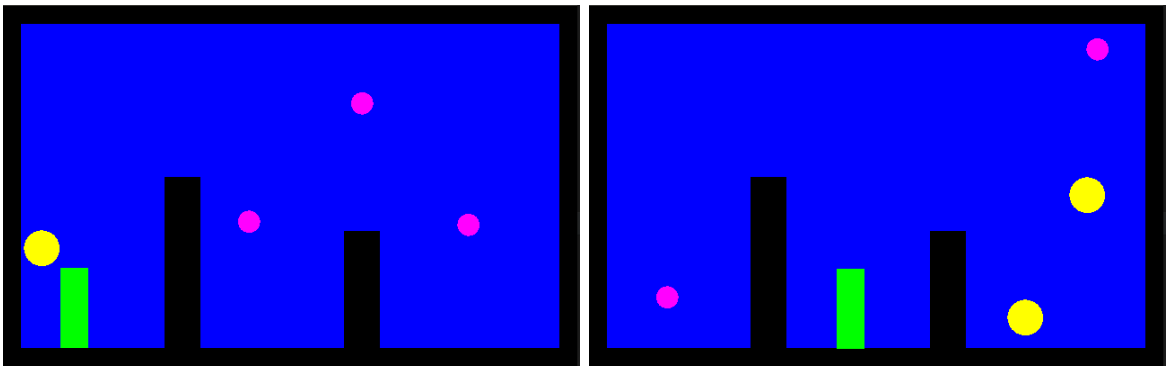


Figure 3.3: Two exemplary scenarios in GeoFriends2, with multiple agents (yellow and green). The rewards are represented by the pink circles. Agents must cooperate in order to finish the map.

GeoFriends2 provides a ready-tu-use set of maps emulating the ones found in Geometry Friends, and also provides an interface for new maps to be implemented by users. These simply require obstacles, rewards, spawn points for agents, and a set of conditions to determine whether the environment was completed (usually, when no more rewards are available).

GeoFriends2 does not address communication between agents in our environment, as we feel communication works best when used directly in agent implementations. In other words, the simulator leaves it to researchers to decide whether and how to communicate between agents, forcing no architecture, timing, reliability, or size constraints.

The environment, even in single-agent mode, is complex enough that reward-based learning techniques like AnQ and A3C are incapable of learning adequate policies within reasonable time [59]. Additional techniques were necessary for agents to achieve complex policies, including intra-agent weight sharing (using a single network with multiple output layers instead of separate networks), breadcrumbs (small heuristic rewards when agents perform adequate

actions), and input shaping (simplifying the agent's state-space and filtering out irrelevant observations).

3.3 Game-Theoretic Environments

This chapter now lists the pay-off matrices of multiple Game-Theoretic environments. Despite being single-state games, these are known and common benchmarks, with uniques properties, which represent a well-rounded test suite for any MARL algorithm.

The competitive version of Matching Pennies is a standard 2-action competitive game with balanced strategies (actions should be played with the same probabilities). Its cooperative counterpart has two deterministic equilibria, one where both players pick the first action, and another where both players pick the second action. The Tricky Game has balanced strategies, like Matching Pennies, but is much harder for algorithms to achieve them [51]. The Biased Game is a competitive game with unbalanced actions, whose Nash equilibrium is $(.75, .25)$ and $(.25, .75)$. All are shown in Figure 3.4.

$1, -1$	$-1, 1$
$-1, 1$	$1, -1$

(a) Matching Pennies.

$1, 1$	$-1, -1$
$-1, -1$	$1, 1$

(b) Coop Matching Pennies.

$0, 3$	$3, 2$
$1, 0$	$2, 1$

(c) Tricky Game.

$1, 1.75$	$1.75, 1$
$1.25, 1$	$1, 1.25$

(d) Biased Game.

Figure 3.4: The pay-off matrices of multiple 2-action 2-player games.

Rock-Paper-Scissors (RPS) is a 3-action game with balanced strategies, while its variant Null Rock-Paper-Scissors (NRPS) is a 4-action derivation with a dominated action (an action that should never be played) and a positive average reward, whose Nash equilibrium is $(0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ for both players. The 4-Action Cooperative Game is a 4-action game with a dominant action, whose Nash equilibrium is $(1, 0, 0, 0)$ for both players. The 1v1 Kick game models players as an attacker that can hesitate, or kick the ball in a direction, and a defender that can hesitate or defend some direction. Hesitating is a dominated action, and the game's Nash equilibrium is $(0, \frac{1}{2}, 0, \frac{1}{2})$ for both players. These games' pay-off matrices are described in Figure 3.5.

$0, 0$	$1, -1$	$-1, 1$
$-1, 1$	$0, 0$	$1, -1$
$1, -1$	$-1, 1$	$0, 0$

(a) Rock-Paper-Scissors.

$-1, -1$	$-1, -1$	$-1, -1$	$-1, -1$
$-1, -1$	$2, 2$	$3, 1$	$1, 3$
$-1, -1$	$1, 3$	$2, 2$	$3, 1$
$-1, -1$	$3, 1$	$1, 3$	$2, 2$

(b) Null Rock-Paper-Scissors.

$1, 1$	$0, 0$	$0, 0$	$-1, -1$
$0, 0$	$1, -1$	$-1, 1$	$-1, -1$
$0, 0$	$-1, 1$	$1, -1$	$-1, -1$
$-1, -1$	$-1, -1$	$-1, -1$	$-1, -1$

(c) 4-Action Cooperative Game.

$-1, -1$	$-1, 1$	$-1, 1$	$-1, 1$
$1, -1$	$1, -1$	$1, -1$	$-1, 1$
$1, -1$	$-1, 1$	$1, -1$	$-1, 1$
$1, -1$	$-1, 1$	$1, -1$	$1, -1$

(d) 1v1 Kick Game.

Figure 3.5: The pay-off matrices of multiple 2-player games with more than two actions.

Prisoner's Dilemma models players as prisoners that can either cooperate or betray each other. If both cooperate, their rewards are better than if they both betray each other. However, if one betrays and the other does not, the betrayer gets the best reward possible. The Nash equilibrium is for both to betray (since their strategy cannot be taken advantage of), but a Pareto-optimal solution would be for both to cooperate with each other. In fact, the Nash equilibrium is the only joint-strategy which is not Pareto-optimal. Stag Hunt models two hunters that can either hunt rabbits (where they always catch something) or stags (where they can get better rewards, but only if both decided to do it). Despite having three Nash equilibria, the one with the highest rewards is for both agents to always catch stags. Battle of the Sexes has a worse stochastic equilibrium and two better deterministic equilibria, but one of the agents will obtain a smaller reward than the other. All these have deterministic equilibria, and are shown in Figure 3.6.

$-1, -1$	$-3, 0$
$-3, 1$	$-2, -2$

(a) Prisoner's Dilemma.

$2, 2$	$0, 1$
$1, 0$	$1, 1$

(b) Stag Hunt.

$3, 2$	$0, 0$
$0, 0$	$2, 3$

(c) Battle of the Sexes.

Figure 3.6: The pay-off matrices of multiple 2-action 2-player games with deterministic equilibria.

Robust algorithms are expected to converge in a wide array of scenarios like this.

3.4 Fully-Observable Environments

This section lists environments with non-singular state-spaces, represented as MMDP, whose state can be directly sampled by agents.

3.4.1 Competitive Grid Games

Maze-related games focus on having agents explore and determine an optimal path to traverse a maze and reach a goal position. In the multi-agent setting, agents either find each other or independently try to find the exit. MazeRPS [58] is a multi-state scenario, derived from Game Theory, where two agents cross a labyrinth to find each other and play a single round of NRPS. Since the average reward of NRPS is positive, agents are incentivized to complete the scenario as fast as possible. Variations of this game are partially-observable with continuous-valued states [57], which make tabular algorithms like Q-learning unable to learn policies without some sort of state approximation function.

The grid Soccer Kick environment [58] is another scenario with multiple states. It has two agents, an attacker and a defender, who compete for a match point in a zero-sum game. If the attacker reaches the goal, it wins, so the defender's goal is to reach him first. If both agents reach each other, they play a 1v1 Kick game to determine the winner.

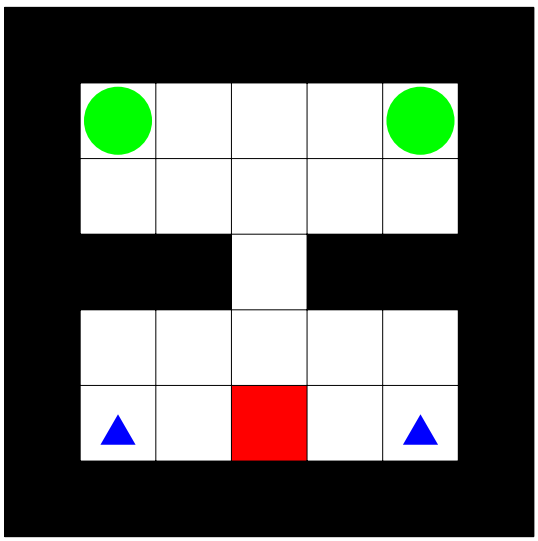
Keep-away Soccer games essentially focus on two teams of agents where an attacker team tries to take the ball from the defender team. There is usually a limited area (which may shrink over time) where the defenders can keep the ball. The grid 3v2 Keep-Away Soccer Environment [58] has three defenders who protect the ball from two attackers, and two variants where defenders can or cannot move. While one of the attackers tags a defender, the other

chases the ball. A good strategy for defenders is often to keep passing the ball between un-tagged members, keeping it away from attackers.

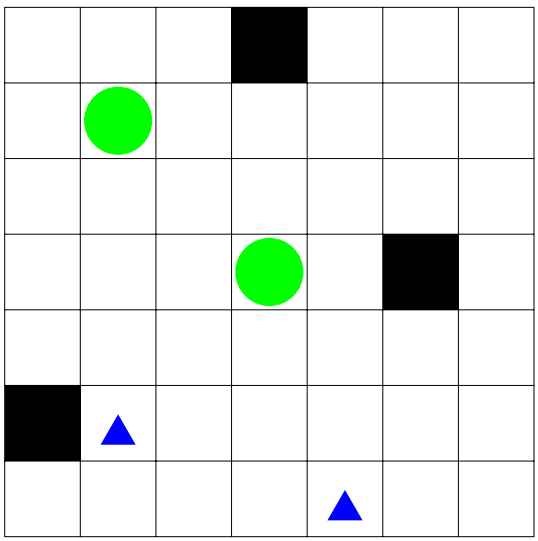
3.4.2 Cooperative Grid Games

Foraging environments can often be solved by a single-agent, but the task's completion can be parallelized in a MAS. Agents can communicate good foraging regions in partially-observable environments, or they can compete for resources. An environment [56] with complete global vision was implemented, shown in Figure 3.7(a), where spawn locations are randomized in different halves of the map and agents cooperate to collect berries. The action-space consists on movement in any of four directions, catching or releasing berries, and standing still. Actions occur simultaneously, and agents can carry one single berry at a time, and release it at their base.

Pursuit games consist on a team of predators capturing a team of prey. If prey are rational and move at the same speed or faster than predators, the task cannot be completed by a single predator. An environment [56] with complete local vision over a toroidal map was implemented, shown in Figure 3.7(b), with random spawn locations. The action-space consists on movement in any of four directions, and standing still. A prey is considered caught when a predator occupies its space. Actions occur sequentially for predators and prey, but are simultaneous for members of the same team. The prey move at the same speed as the predators and escape from their closest enemy, thus requiring a coordinated effort to successfully complete the task.



(a) Foraging Task. Agents (blue) need to collect berries (green) and bring them to their base (red).



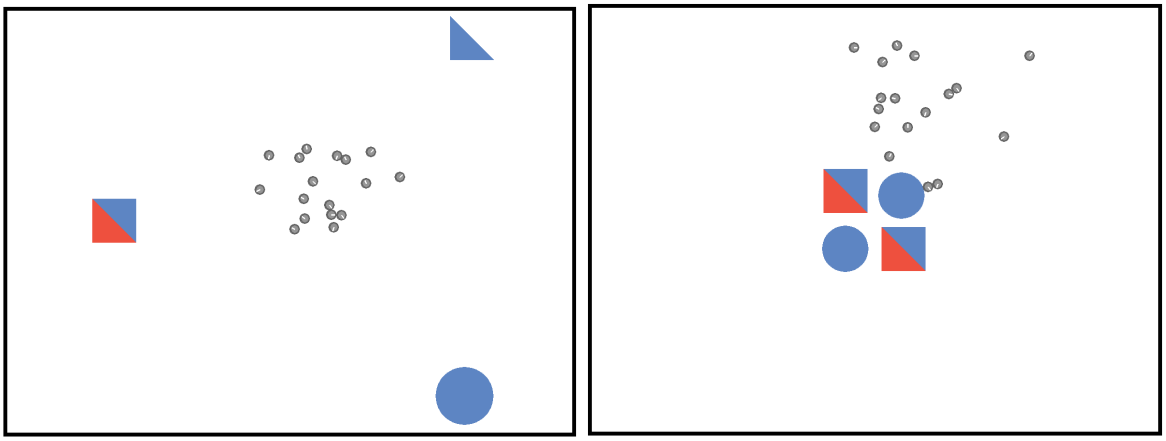
(b) Pursuit Task. Predators (blue) need to surround and catch prey (green) in a toroidal map.

Figure 3.7: The Foraging and Pursuit tasks, two fully-observable multi-agent environments [56].

3.4.3 KiloBots Environment

The KiloBots environment [3] is a set of local continuous observation- and action-space scenarios with simulated physics, emulating the KiloBot robot [193]. Each robot has a diameter of 3 centimeters and moves using two vibration motors. The environment is a great testbed for swarm learning algorithms, which can later be demonstrated with physical KiloBots. The scenarios include:

- *Push Objects* - Agents push an object to a target location.
- *Assemble Objects* - Agents push and assemble multiple objects together.
- *Segregate Objects* - Agents push and separate multiple objects apart.



(a) Assemble Objects. Agents must push and join the objects together at any point in the map.

(b) Segregate Objects. Agents must push and separate the objects as far away from each other as possible.

Figure 3.8: Two scenarios of the KiloBots environment. Agents (grey) know their own poses and the relative polar coordinates of other agents and objects (colored).

Agents observe their own pose and the coordinates of the obstacles in the environment, relative to their own position. They receive no additional information regarding the shape or size of obstacles. Each environment rewards the team at every cycle based on the distance obstacles were pushed.

3.5 Partially-Observable Environments

This section lists environments with partially-observable state-spaces, represented as Dec-POMDP, where agents can only sample incomplete, noisy, or incorrect observations of the underlying state.

3.5.1 POC Suite

A suite of partially-observable multi-agent environments was proposed, which we call the POC suite, where communication is crucial to overcome the partial observability of the environments. It has been partially published in the WorldCIST19 and IJCNN19 conferences

[61, 62]. The following sections describe a Hidden Reward challenge, a Traffic Intersection simulator, a Pursuit game, and a Navigation task, as shown in Figure 3.9. The Hidden Reward game focuses on classic exploration, and agents need to learn how to efficiently explore the environment and alert team members when the target is found. The Traffic Intersection is a close-horizon game, with large amounts of agents, where agents need to learn to adhere to rules and overcome multiple indistinguishable intersections. The Pursuit game is a classical benchmark where agents need to explore and coordinate in order to capture the prey, and the Navigation task focuses on goal assignment, and agents learn how best to distribute themselves in order to complete the task.

These environments are performance-oriented and provide a controlled environment with which to test multiple aspects of a multi-agent algorithm. While agents receive observations about the environment, it is possible to access the environment's underlying state directly, without any additional computations (aside from those already performed by the actual environment). This allows algorithms with the *centralized learning, distributed execution* method to augment their learning phase with additional information in an efficient manner. The environments are also targeted at cooperative teams, and built in such a way that a team of agents must use an information-sharing method to achieve successful coordinated strategies. For example, agents in the Traffic environment must share their intent to turn in order to avoid collisions.

Hidden Reward

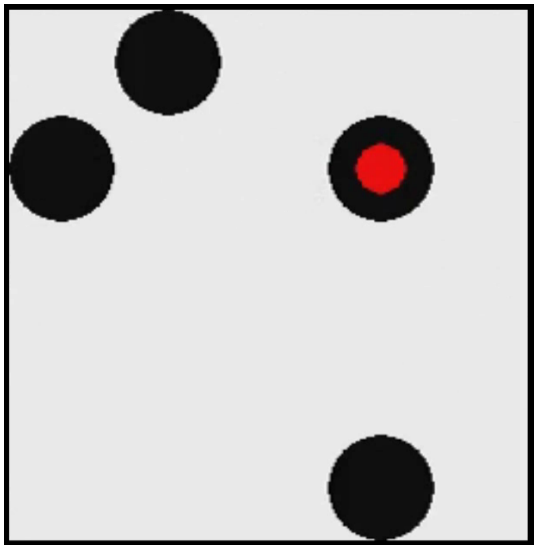
The Hidden Reward challenge consists on several agents having to move across a toroidal map until they find a reward zone. Each agent has a local partial observation of the environment, with their own coordinates and whether they're in that zone or not. The complete state-space consists on a concatenation of all the agents' partial observations. There's both a global time limit since the challenge starts, and a smaller one since any agent finds the reward zone. In other words, agents have some time to explore the map and find the hidden reward, and a short time to gather there once it has been found. Because the time is not enough for a single agent to fully explore it, this not only forces spread coordinated exploration, but also an alert protocol when the reward is found.

Agents receive individual rewards each cycle, 0 points if not on the reward zone, and n points if on the reward zone, where n is the total amount of agents there, so cooperation is encouraged. At each time-cycle, agents can move in four directions or remain in the same position. Agents can broadcast messages to all other agents. An adequate strategy is to explore the map until the reward zone is found, and then broadcast its position to other agents.

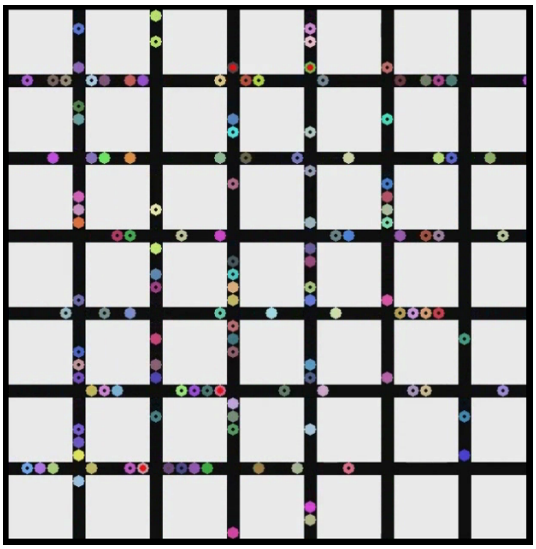
Traffic Intersection

The Traffic Intersection simulator consists on several road intersections, which must be crossed by multiple vehicles. Each agent has a local partial observation of the environment, knowing their desired direction and sensing close vehicles. The complete state-space consists on the positions and intended directions of all vehicles.

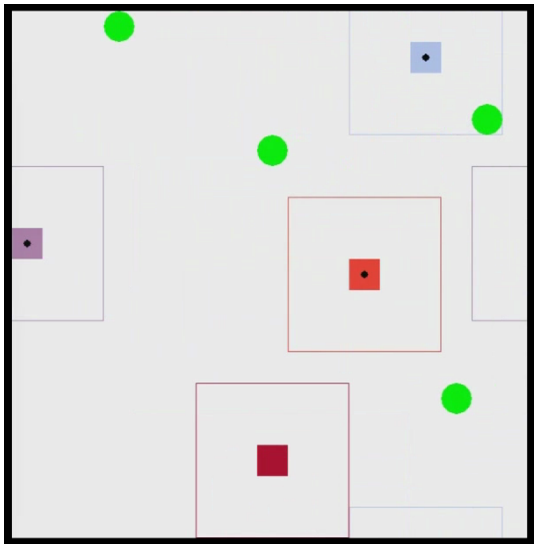
Agents get small penalties for stalling traffic, big penalties if they crash, and even larger penalties if they crashed without having priority. At each time-cycle, agents can move or remain in the same position. The communication range of agents is geographically limited to



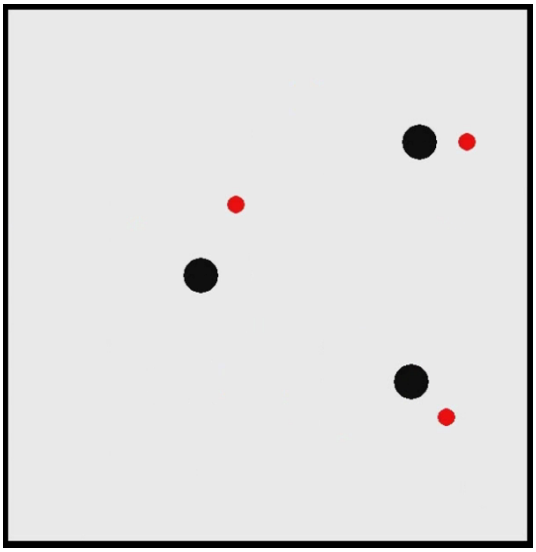
(a) Hidden Reward. Agents (black) only know their own position and explore the map until the reward zone (red) is found.



(b) Traffic Intersection. Agents (colored) must cross intersections without colliding. They know their desired direction, sense other vehicles around them, and are penalized if they collide (red marker).



(c) Pursuit. Predators (squared) see only a small local area, and must chase, surround and capture the prey (green circles), which are hard-coded to run from the closest predator.



(d) Navigation. Agents (black) know the beacons' coordinates, but not each others' positions. They must cover all the beacons (red), and are rewarded by how close any agent is to each beacon.

Figure 3.9: The POC suite, consisting of four environments: Hidden Reward, Traffic Intersection, Pursuit, and Navigation. All are partially-observable multi-agent environments, where agents benefit from sharing information and coordinating as a team.

close vehicles (agents do not broadcast messages to all others, just to other agents in the same intersection). An adequate strategy is for a vehicle to inform others at intersections whether it needs to turn or not, and allow the vehicle with priority to cross the intersection.

Pursuit

The Pursuit game is based on the one shown in Section 3.4.2, and consists on two teams of agents, where one team must capture the other. The prey team is hard-coded, has global vision, and each prey runs from the closest predator. Predators have local partial observations of the environment, sensing a small local area equivalent to less than 10% of the total map and their own global coordinates. The complete state-space consists on the positions of all predators and prey.

Agents get small penalties as time passes, and get penalized and randomly placed if they collide. At each time-cycle, agents can move in four directions or remain in the same position. Agents can broadcast messages to all other agents. A high-level strategy is for predators to explore the map until a prey is found, and then broadcast the prey's position so that all predators can converge and capture it.

Navigation

The Navigation task consists on several agents having to cover all the beacons spread throughout the map. Each agent knows only its own position and the beacon positions. The complete state-space consists on the positions of all agents and beacons. There is a time-limit, but the episode ends early if all beacons are covered.

The team gets points at the end of each time-limited episode, based on how close an agent was to each beacon. At each time-cycle, agents can move in four directions or remain in the same position. Agents can broadcast messages to all other agents. An adequate strategy is for each agent to broadcast its own position and for the team to decide which agent should cover which beacon.

3.5.2 Multi-Agent Particle Environment

The Multi-Agent Particle Environment [23] suite is a set of local continuous observation- and discrete action-space scenarios with simulated physics, which may incorporate communication. The action-space for an agent consists on increasing or decreasing its velocity in one of two axes, both of which decay over time. An agent's observation space usually consists on the relative positions of all entities on the map, as well as the velocities of all allied agents. In environments with agent-specific targets, observations also comprise the relevant information (possibly of a different agent's target, thus requiring communication to complete the task).

The scenarios, some of which are shown in Figure 3.10, include:

- *Physical Deception* - One adversary, N allies and landmarks. One landmark is a target, known by the allies, but unknown to the adversary. Allies receive points by how close one is to the target, and lose points by how close the adversary is to the target.

- *Covert Communication* - One adversary, two allies. Allies share a private key and use it to encrypt and decrypt a value, while the adversary tries to decipher it.

- *Keep-Away* - One adversary, one ally, one landmark. Agents are rewarded for being close to the landmark, but the adversary receives more points if the ally is farther away.
- *Cooperative Reference* - Two allies, three landmarks. Allies know the target landmark of the other agent, and not their own, which they are rewarded for being close to.
- *Cooperative Communication* - The same as *Cooperative Coverage*, but only one ally can move, while the other knows its target.
- *Cooperative Navigation* - N allies and landmarks. Agents are rewarded by being close to each landmark.
- *Tag Challenge* - One adversary, N allies and landmarks. Allies try to touch as many times as possible the faster adversary.

In order to successfully complete these tasks, mechanisms to handle partial-observability are required. These may range from memory of previous states to communication protocols. In addition, tasks also require strong coordination skills, and Cooperative Reference and Cooperative Communication both require relevant information sharing to successfully be completed. We refer the reader to the original publication [23] for further information.

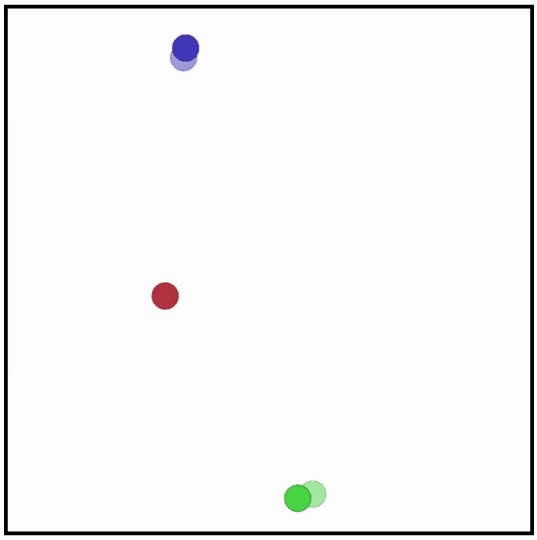
3.5.3 3D Soccer Simulation League

The RoboCup initiative [194, 16] is an annual international robotics competition, whose goal is to have a team of fully-autonomous physical robots winning a soccer match against the world-champion human team, using FIFA standard rules, by the year 2050. The 3D Soccer Simulation League, a part of the RoboCup initiative, is a complex multi-agent environment where two teams of humanoid simulated robots play a ten-minute soccer match using realistic rules. Each team is comprised of eleven NAO robots [195] with multiple different models, each with different physical characteristics.

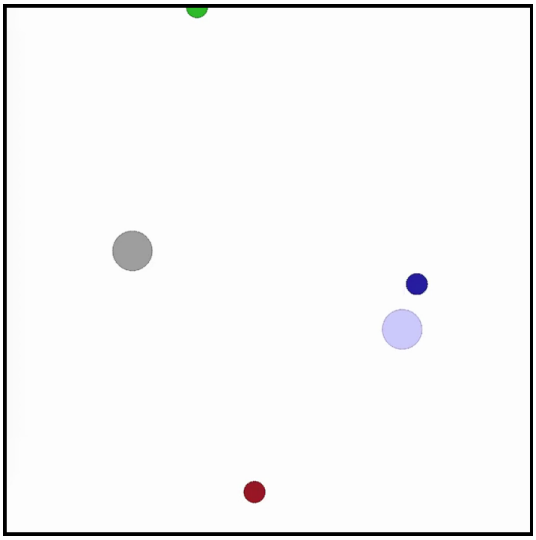
Each agent perceives the environment through local partial observations consisting on spherical coordinates of other elements in the environment. These include landmarks like the goal posts, field lines, other agents, and the ball. Agents then act upon their own local joints, by sending commands to the environment simulator. The observation rate is different from the action rate, as agents only sample new observations every three cycles. Agents can communicate limited-size messages to each other during the match, but only a single message can be heard by each agent per cycle.

Using low-level controllers that abstract simple tasks, like kicking or walking, has been the *de facto* standard in the league, controllers which are then used by high-level decision-making modules. In other words, agents have a set of behaviors which are chosen according to their strategy. The behavior acts upon the agent's joints, and the strategy defines which behavior to execute and with which parameters.

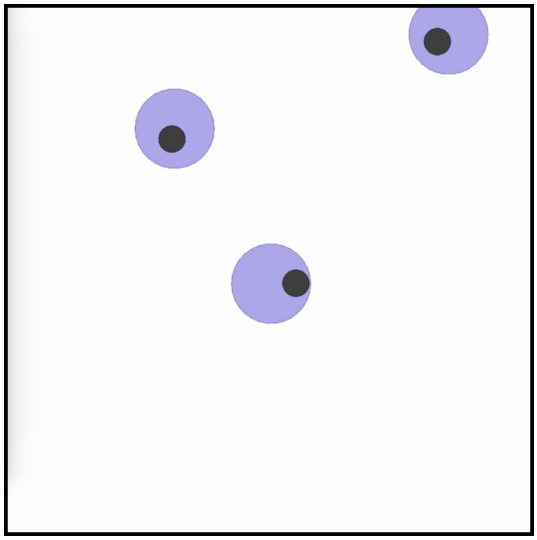
The 3D Soccer Simulation League features multiple game types. The most common is the actual soccer match, as described above, and an example is shown in Figure 3.11. Another is the Penalties match, where a single kicker is pitted against the goal-keeper and has a limited time to score a goal. This match is used as a tie-breaker in competitions. Another game type is the Keep-Away Soccer, where three players keep the ball away from a single opponent for as long as possible. It was used as an additional challenge for the teams in the 2017 RoboCup competition.



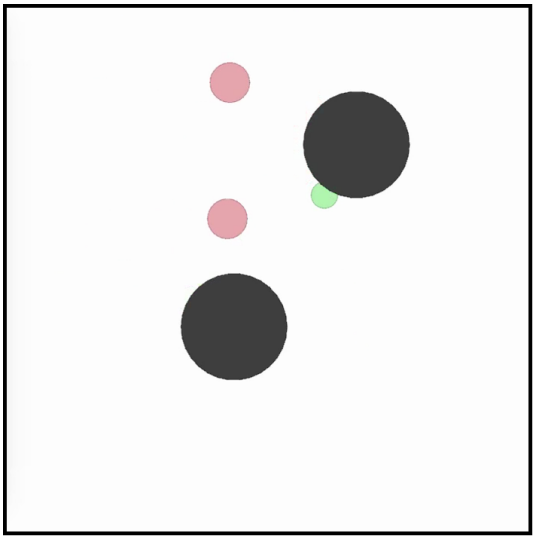
(a) Cooperative Reference. Agents (light colored) know their positions and their ally's target landmark (heavy colored). They must share that information and converge on their corresponding targets.



(b) Cooperative Communication. The gray agent cannot move and knows its ally's (light blue) target (dark blue). The grey agent must share that information for the light blue agent to converge on it.



(c) Cooperative Navigation. Agents (blue) must cover all landmarks (black), by deciding which agent should cover which landmark.



(d) Tag Challenge. Agents (red) try and touch the prey (green) as many times as possible, while dodging the black obstacles and remaining within map boundaries.

Figure 3.10: Some environments from the MPE suite, consisting on Cooperative Reference, Cooperative Communication, Cooperative Navigation, and Tag Challenge.

Figure 3.11: A soccer match in the 3D Soccer Simulation League.

While the 3D Soccer Simulation League is an open-source project that has been developed by the community for a number of years, we have designed and implemented a framework to allow for learning algorithms to use it. The 3d Soccer Simulation Learning (3dSSL) framework, as shown in Figure 3.12, defines the environment with a Gym interface, and allows multiple environments to run simultaneously with fault-tolerance mechanisms enabled. It supports parallel learning with asynchronous deep learning or genetic algorithms, through socket communication implemented within the FCPortugal3D team. The framework has been published on the ROBOT2019 conference [63].

3dSSL supports both single- and multi-agent scenarios. In the single-agent paradigm, it can be used to optimize *get-up*, *kick*, and *walking* behaviors, while in the multi-agent paradigm, 3dSSL allows a team to develop policies for *Passing* and *Keep-Away* games. The 3dSSL Passing scenario consists on three agents passing the ball between them as many times as possible within a time interval, while the 3dSSL Keep-Away scenario has an additional opponent that tries to steal the ball, ending the game.

For the 3dSSL single-agent scenarios, an agent observes its current joints and estimated position, orientation, and gyroscope information. Its actions-space consists on low-level continuous commands to all its joints. The estimated information is directly sampled by the mechanisms currently employed by the FCPortugal3D team. If the agent crashes, the simulator is killed and the framework reports a terminal state.

For the 3dSSL multi-agent scenarios, agents observe the estimated positions of all players and the ball, as well as the estimated location the ball will stop at, their orientation, and the distances of all players to the ball. All estimated information is again sampled by the mechanisms used in FCPortugal3D. The action-space is no longer a low-level continuous joint-control, but instead a discrete selection of high-level behaviors used in the team. The behaviors used were *standing*, where an agent simply stands in the same place, *getting up*, used when the agent falls on the ground, *kicking to an ally*, with the choice of which teammate to kick the ball towards, and using a kick for the adequate distance to the target, *moving to position*, where each agent has a specific default location based on the team's formation. Agents may fall when changing behaviors abruptly, and all behaviors take multiple time-steps to complete.

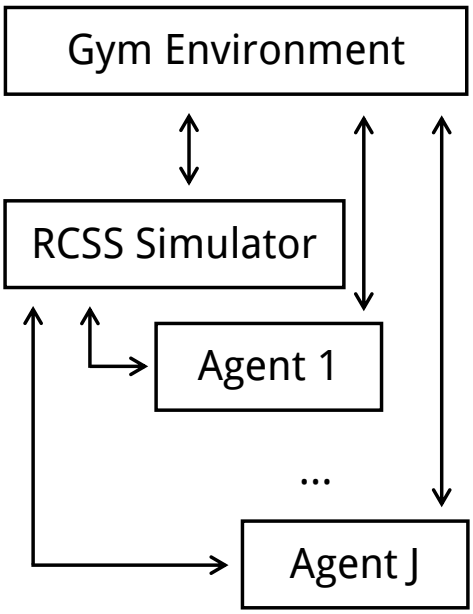


Figure 3.12: The framework for the 3dSSL framework. The framework deploys an RCSS Simulator and J agents, all of which connect to the deployed simulator. Both the simulator and agents remain connected to the framework, which informs agents of what actions they will execute upon the environment. Agents then report their new observations.

In the event of a crash by any agent, all agents and the simulator are killed, and the framework reports a terminal state.

3dSSL also provides additional information to be used by learning algorithms or on tests, but which is not normally accessible by agents in regular play. This includes the agent's real position, orientation, acceleration, as well as the ball's real current position. This information can be used during the learning phase of algorithms that benefit from data that are usually unaccessible to agents.

3.5.4 Simple Pokémon Environment

Pokémon [196] is the largest entertainment franchise in the world, having at its core a role-playing video game series. In a Pokémon battle, a player competes with a roster of up to six Pokémons, and each Pokémon possesses a set of statistics such as hit points (HP), attack, defense, and speed, along with four moves. In 1v1 battles each player controls one active Pokémon fighting against the opponents' active Pokémon and holds the remaining on the bench. Each turn the player may select one of four moves of the Pokémon to attack the opponent's active Pokémon, or the player may switch the active with one benched Pokémon. When a Pokémon's HP are depleted, the Pokémon faints, is forcibly switched out, and is unusable for the remainder of the battle. A core component of Pokémon is typing, each Pokémon and move possesses a type, and types have different effectiveness against different types. A Fire type move is effective against Grass type Pokémon, and Water type Pokémon resists to Fire type moves. The effectiveness of move typing is translated to a modifier chart, an asymmetrical matrix shown in Table 3.1, where effectiveness doubles an attack's damage,

resistance halves it, and immunity negates it.

	Attacking Pokémon																	
	No	Fi	Fl	Po	Gr	Ro	Bu	Gh	St	Fr	Wa	Gr	El	Ps	Ic	Dr	Da	Fa
Normal	1.0	2.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Fighting	1.0	1.0	2.0	1.0	1.0	0.5	0.5	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	0.5	2.0
Flying	1.0	0.5	1.0	1.0	0.0	2.0	0.5	1.0	1.0	1.0	1.0	0.5	2.0	1.0	2.0	1.0	1.0	1.0
Poison	1.0	0.5	1.0	0.5	2.0	1.0	0.5	1.0	1.0	1.0	1.0	0.5	1.0	2.0	1.0	1.0	1.0	0.5
Ground	1.0	1.0	1.0	0.5	1.0	0.5	1.0	1.0	1.0	1.0	2.0	2.0	0.0	1.0	2.0	1.0	1.0	1.0
Rock	0.5	2.0	0.5	0.5	2.0	1.0	1.0	1.0	2.0	0.5	2.0	2.0	1.0	1.0	1.0	1.0	1.0	1.0
Bug	1.0	0.5	2.0	1.0	0.5	2.0	1.0	1.0	1.0	2.0	1.0	0.5	1.0	1.0	1.0	1.0	1.0	1.0
Ghost	0.0	0.0	1.0	0.5	1.0	1.0	0.5	2.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0
Steel	0.5	2.0	0.5	0.0	2.0	0.5	0.5	1.0	0.5	2.0	1.0	0.5	1.0	0.5	0.5	0.5	1.0	0.5
Fire	1.0	1.0	1.0	1.0	2.0	2.0	0.5	1.0	0.5	0.5	2.0	0.5	1.0	1.0	0.5	1.0	1.0	0.5
Water	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.5	0.5	2.0	2.0	1.0	0.5	1.0	1.0	1.0
Grass	1.0	1.0	2.0	2.0	0.5	1.0	2.0	1.0	1.0	2.0	0.5	0.5	0.5	1.0	2.0	1.0	1.0	1.0
Electric	1.0	1.0	0.5	1.0	2.0	1.0	1.0	1.0	0.5	1.0	1.0	1.0	0.5	1.0	1.0	1.0	1.0	1.0
Psychic	1.0	0.5	1.0	1.0	1.0	1.0	2.0	2.0	1.0	1.0	1.0	1.0	1.0	0.5	1.0	1.0	2.0	1.0
Ice	1.0	2.0	1.0	1.0	1.0	2.0	1.0	1.0	2.0	2.0	1.0	1.0	1.0	1.0	0.5	1.0	1.0	1.0
Dragon	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.5	0.5	0.5	1.0	2.0	2.0	1.0	2.0
Dark	1.0	2.0	1.0	1.0	1.0	1.0	2.0	0.5	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	0.5	2.0
Fairy	1.0	0.5	1.0	2.0	1.0	1.0	0.5	1.0	2.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0	0.5	1.0

Table 3.1: The Pokémon type effectiveness chart.

The game features a number of properties that make it a challenging environment for machine learning algorithms. It is partially-observable, since each trainer only knows information about its own team, and some visible stats regarding the opponent's active Pokémon. Unlike chess or Go, the environment is stochastic, and attacks have a chance to hit and do damage based on an interval formula. It is a competitive multi-agent system with a zero-sum reward scheme. The state-space is continuous and high-dimensional, with six Pokémon for each trainer, each with one or two of seventeen different Pokémon types, four attacks per Pokémon, each with its own typing as well, HP and statistics for each Pokémon, and power and accuracy values for each attack.

Multiple Pokémon environments have been developed over the years, both officially released [196], or fan-made [197, 198]. The fan-made battle simulators don't provide adequate API for machine learning algorithms, and are instead focused on human user-experience. To the best of our knowledge, proper API are only available for information mining about the games [199, 200], such as Pokémon lists or maps. Due to this, an AI competition has recently been proposed for the IEEE Conference on Games, known as Showdown AI Competition [201]. The authors state that most current competitions are based on real-time video games with perfect information, and that there is a need to explore other types of games with competitive nature. They identify some properties that contrast Pokémon battling with other games:

- Branching Factor - with (on average) nine possible actions per turn (there are four possible moves and five possible switches), planning multiple steps ahead is computationally heavy;

- Turn Atomicity - although Pokémon is a turn based game, choices are made simultaneously, and agents only observe both moves after selection;

- Categorical Dimensions - although in a clean state, HP and statistics are sufficient to evaluate the actions' reward, over-time conditions like *burned* or *paralyzed*, or other field effects like *sandstorm* or *stealth rock* are hard to quantify (delayed rewards);

- Stochasticity - moves' damage calculation have random parameters, and may miss and do zero damage;

- Hidden Information - this environment is partially-observable at two levels. The opponent's active Pokémon's move set, statistics and abilities (although this can be minimized with domain knowledge), and at the team level, the unawareness of the opponent's roster makes it harder to plan ahead and predict best long-term strategy;

The authors argue that these properties motivate the existence of a Pokémon battle simulator that is compliant with standard machine learning interfaces.

We implemented the Simplified Pokémon Environment (SPE), shown in Figure 3.13, where two agents conduct a Pokémon battle and each team is composed of an active and a bench Pokémon. Pokémon have 300 initial HP, a single typing, and moves of semi-random types, but are otherwise not identified. Each move has a power uniformly distributed between 50 and 100 (multiplied by 1.5 if the move's type is the same as the Pokémon's, a feature known as STAB). Each Pokémon has at least one move of its type, and three moves of random types that are not effective against the Pokémon and that the Pokémon is not effective against. For example, a Fire Pokémon will not have a Water move (effective against Fire) or a Grass move (which Fire is effective against). While some specific Pokémon have attacks with these unconventional typings, Pokémon are not identified in SPE, and removing unintuitive type combinations from SPE leads to more strategical policies learned by agents.

Agents have a reward function $R = R_d + R_f - R_t$, where $R_d \in [0, 1]$ represents the damage dealt as a fraction of the opponent's maximum HP, $R_f \in 0, 1$ is a bonus if the opponent fainted, and $R_t \in [0, 1]$ represents the damage taken as a fraction of the Pokémon's maximum HP. With the feedback provided each turn, this results in non-sparse rewards. If a Pokémon faints, it automatically switches to the bench Pokémon, and a battle ends when a trainer's Pokémon have all fainted. In complete Pokémon battles, the move order is deterministically determined by move priorities and Pokémon speed. Because SPE has no speed statistics, the order of attacks is random each turn, leading to stochastic state transitions. The switch action, similarly to real Pokémon games, always occurs before attacks.

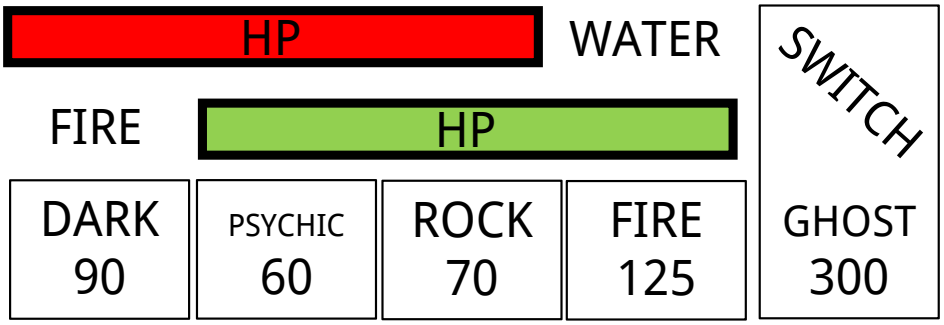


Figure 3.13: The SPE environment. The opponent has an active Water Pokémon, and the local agent has an active Fire Pokémon, with Dark, Psychic, Rock and Fire moves. The trainer can also switch to the benched Ghost Pokémon with 300HP.

The five-dimensional action space of each agent corresponds to the four moves from the active Pokémon, and the switch option. If the bench Pokémon has fainted, the switch action does nothing. Agents sample actions simultaneously, and sample an observation composed of the status (HP and type) of both his Pokémon, the status of the opponent's active Pokémon, and the HP of the opponent's bench Pokémon. Additionally, the state space contains the power

and type of all four moves of the active Pokémon. With 18 different types, and identifying them through a one-hot vector, this is equivalent to a 134-dimensional state space for each agent.

The goal for each agent is to learn typing combinations and optimizing a long-term strategy that maximizes the chances of winning the battle. For example, the Switch action will give the agent an immediate reward $R \leq 0$, but switching to a Pokémon with better typing and move pool often increases the overall reward obtained. Comparing SPE with an official Pokémon game, the branching factor was decreased from nine to five and there are no long-term conditions or mechanics, which creates a simpler environment. However, stochasticity is exacerbated with a random set of generated Pokémon and moves, instead of a fixed set of viable combinations and teams. SPE is compliant with the Gym [161] API and its source code can be found at `https://gitlab.com/DracoStriker/simple-pkm-env`.

3.6 Conclusion

This chapter described a diverse set of multi-agent environments, ranging from cooperative to competitive environments, fully-observable single-state games to partially-observable long horizon tasks, and continuous to discrete action- and state-spaces. Table 3.2 summarizes general properties of all the multi-state environments shown above.

General multi-agent algorithms are expected to allow agents to achieve successful policies across these, and as such, they will be used across the remainder of this thesis as test beds on which proposals will be evaluated.

Environment	Design	Action Sp.	State Sp.	Team	Vision	Persp.
GeoFriends2	Coop	Discrete	Continuous	Static	Full	Global
MazeRPS	Comp	Discrete	Discrete	Static	Full	Global
Grid Soccer Kick	Comp	Discrete	Discrete	Static	Full	Global
Grid K.A. Soccer	Comp	Discrete	Discrete	Static	Full	Global
Grid Pursuit	Coop	Discrete	Discrete	Static	Full	Local
Grid Forager	Coop	Discrete	Discrete	Static	Full	Global
POC Hid. Rew.	Coop	Discrete	Continuous	Static	Partial	Global
POC Traffic Sim.	Mixed	Discrete	Discrete	Varying	Partial	Local
POC Pursuit	Coop	Discrete	Discrete	Static	Partial	Local
POC Navigation	Coop	Discrete	Continuous	Static	Partial	Global
MPE Navigation	Coop	Discrete	Continuous	Static	Full	Local
MPE Comm.	Coop	Discrete	Continuous	Static	Full	Local
MPE Reference	Coop	Discrete	Continuous	Static	Full	Local
MPE Phys. Dec.	Mixed	Discrete	Continuous	Static	Full	Local
MPE Tag	Mixed	Discrete	Continuous	Varying	Full	Local
KiloBots Light	Coop	Continuous	Continuous	Static	Full	Local
KiloBots Join	Coop	Continuous	Continuous	Static	Full	Local
KiloBots Split	Coop	Continuous	Continuous	Static	Full	Local
3dSSL Passing	Coop	Continuous	Continuous	Static	Partial	Local
3dSSL Keep-Away	Coop	Continuous	Continuous	Static	Partial	Local
SPE	Competitive	Discrete	Continuous	Static	Partial	Local

Table 3.2: Comparison of multiple multi-agent multi-state environments, regarding their design (whether agents behave cooperatively, competitively, or in a mixed manner), their action- and state-spaces (discrete or continuous), how the team size behaves during each episode (if the amount changes or remains static), and how agents observe the environment (full or partial observations, from a global or a local perspective).

Chapter 4

Multi-Agent Double Deep-Q-Networks

The use of DQN's deep representations has shown great promise in both single- and multi-agent settings. In the former, despite losing theoretical guarantees, an agent was shown to achieve superhuman-level performance across the Atari 2600 environments [26], and the algorithm has since been extended and improved with different network architectures, sampling strategies [147], and even overestimation techniques [145].

In the multi-agent setting, IL adaptations of DQN [151, 152] that exhibit implicit coordination strategies have been presented, and new techniques, like inter-agent weight sharing [150], have also been proposed. None of these works take advantage of the single-agent extensions of DQN.

This chapter describes the extension of a DQN single-agent variation to the multi-agent paradigm. The extension is tested in two environments against tabular Q-learning, with two different approaches. Its adaptability to new unseen tasks and scenarios is also evaluated, and conclusions are drawn regarding its properties. These findings were published in the ROBOT2017 conference [57]. The algorithm's source-code and tests were published at `https://github.com/david-simoes-93/Multi-agent-Double-Deep-Q-Networks`.

4.1 Problem Statement

DDQN is an extension of DQN which decouples action selection and evaluation, and reduces the overestimation of DQN's value function. In single-agent environments, this leads to a more stable and reliable learning process, but the extension has not yet been evaluated in the multi-agent context. Overestimating a value function in a MAS with implicit coordination can lead to unstable learning, and reducing this overestimation can significantly improve policies [202].

It is also unclear how advantageous the IL approach is against the JAL version of multi-agent DQN, where the agents are controlled by a centralized entity, effectively creating a single-agent environment. The JAL approach inherently has the disadvantage of scalability and flexibility, since outputting a joint-action implies a network has to be trained for a fixed amount J of agents, and its output layer will have $|A|$ possible joint-actions, where $|A| = |A_1| \times \ldots \times |A_j|$ is the amount of joint-actions, scaling exponentially with the amount of agents.

Input: Learning rate η, mini-batch size k, time-step limit $t_{\max}$, maximum iterations $T_{\max}$, future reward discount factor γ, target network update period τ, replay memory $\mathcal{D}$ with capacity N, on-line network with random weights θ, and target network with weights θ^- copied from the on-line network.

for iteration $T \leftarrow 1, T_{\max}$ **do**
  Sample state s_1
  for time-step $t \leftarrow 1, t_{\max}$ **do**
    Reset gradients $d\theta \leftarrow 0$
    Select random joint-action a_t with probability ϵ, otherwise best joint-action $a_t \leftarrow \text{argmax}_a Q(s_t, a, \theta)$
    Execute joint-action a_t
    Sample state s_{t+1} and reward r_t
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random mini-batch of k transitions from $\mathcal{D}$
    for transition $i \leftarrow 1, k$ **do**
      Compute target $y_i \leftarrow r_i + \gamma Q(s_{i+1}, \text{argmax}_a Q(s_{i+t}, a, \theta), \theta^-)$ with target and on-line networks
      Compute loss $L_i \leftarrow (y_i - Q(s_i, a_i, \theta))^2$ of on-line network
      Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L_i}{\partial \theta}$
    end for
    Update on-line network weights $\theta \leftarrow \theta + d\theta$
    Update target network weights $\theta^- \leftarrow \theta$ every τ time-steps
  end for
end for

Output: A converged network with weights θ to approximate the value function as $Q(s, a, \theta)$.

Algorithm 9: The Joint-Action Learners variant of the Multi-agent Double Deep Q-Networks algorithm, using ϵ-greedy exploration.

4.2 Proposal

This chapter describes an extension of DDQN to the multi-agent paradigm, Multi-agent Double Deep Q-Networks (MaDDQN) [56]. It describes both JAL and IL versions, and reports their results in two cooperative MMDP, with fully-observable states. It also compares the performance of both versions of MaDDQN with inter-agent parameter sharing, and evaluates how well the networks can generalize to new unseen states and tasks. MaDDQN is a more general version of DDQN, which can be obtained by setting the amount of agents $J = 1$.

An ϵ-greedy exploration strategy in a multi-agent environment must be tuned to compensate for the fact that agents can explore independently, which leads to a higher than desired exploration rate. This can either be tuned by adjusting the exploration rate ϵ_j for each agent j such that $(1 - \epsilon_j)^J = 1 - \epsilon$, or by using the same random number generator and seed for each agent, such that they explore or exploit simultaneously.

The JAL variant of MaDDQN is described in Algorithm 9. Agents can be run in a distributed manner since they have full observations of the environment. These allow, even with local perspectives, a global complete state to be computed. Each agent then calculates the joint-action for the team, and executes its corresponding action. A single replay memory is used to store state- and joint-action transitions. This variant does not support varying numbers of agents without re-optimizing a new network, and scales poorly to large amounts of agents.

The IL variant of MaDDQN is described in Algorithm 10. Agents can run in a distributed manner even with partially-observable environments. A single replay memory can store the state and action transitions of all agents if all agents evenly populate it with the same amount of samples. Otherwise, it is best to use a replay memory $\mathcal{D}_j$ for each agent j and draw the same amount of samples from each, such that networks are optimized with the same amount of samples from each agent. Unlike Egorov [152], who fixes the network weights of all-but-one agents during training and periodically distributes the learned weights to the remaining

Input: Learning rate η, mini-batch size k, time-step limit $t_{\max}$, maximum iterations $T_{\max}$, future reward discount factor γ, target network update period τ, replay memory $\mathcal{D}$ with capacity N, number of agents J, on-line network with random weights θ, and target network with weights θ^- copied from the on-line network.

for iteration $T \leftarrow 1, T_{\max}$ **do**
  Sample state s_1^j for all agents j
  for time-step $t \leftarrow 1, t_{\max}$ **do**
    Reset gradients $d\theta \leftarrow 0$
    for agent $j \leftarrow 1, J$ **do**
      Select random action a_t with probability ϵ, otherwise best action $a_t \leftarrow \text{argmax}_a Q(s_t, a, \theta)$
    end for
    Execute action a_t^j for all agents j
    Sample state s_{t+1}^j and reward r_t for all agents j
    Store transition $(s_t^j, a_t^j, r_t, s_{t+1}^j)$ in $\mathcal{D}$ for all agents j
    Sample random mini-batch of k transitions from $\mathcal{D}$
    for transition $i \leftarrow 1, k$ **do**
      Compute target $y_i \leftarrow r_i + \gamma Q(s_{i+1}, \text{argmax}_a Q(s_{i+t}, a, \theta), \theta^-)$ with target and on-line networks
      Compute loss $L_i \leftarrow (y_i - Q(s_i, a_i, \theta))^2$ of on-line network
      Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L_i}{\partial \theta}$
    end for
    Update on-line network weights $\theta \leftarrow \theta + d\theta$
    Update target network weights $\theta^- \leftarrow \theta$ every τ time-steps
  end for
end for

Output: A converged network with weights θ to approximate the value function as $Q(s, a, \theta)$.

Algorithm 10: The Independent Learners variant of the Multi-agent Double Deep Q-Networks algorithm, using ϵ-greedy exploration.

---

agents, we train all our agents simultaneously. Together with inter-agent parameter sharing, this speeds up the training phase by a factor proportional to the amount of agents.

4.3 Evaluation

Our proposal was tested in two multi-agent environments, a Foraging task and a Pursuit game, both described in Section 3.4.2.

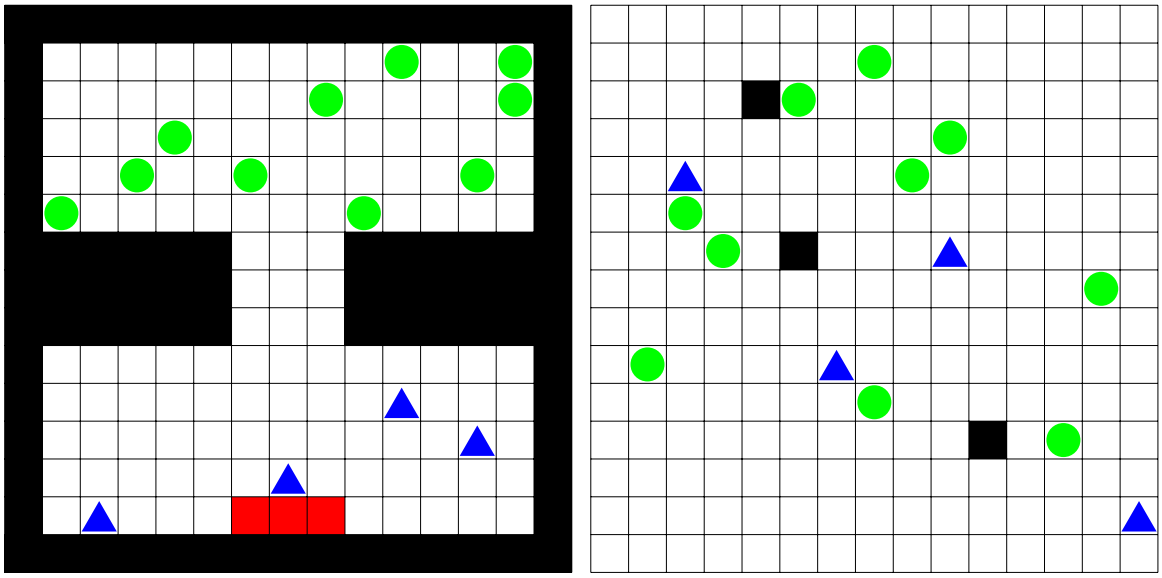
The Foraging task is an environment where agents are tasked with foraging items and bringing them back to specific places. The problem can be solved by a single robot, but multiple agents parallelize the effort. The environment provides homogeneous agents with full global observations. The starting positions of agents and berries are randomized across the lower and upper parts of the map, respectively. Each agent can only carry one item at a time, and can only release it in the base. Agents move simultaneously and collisions prevent movement.

The Pursuit game features a team of homogeneous predators that must capture the elements of the team of semi-randomly moving prey in a toroidal grid. The prey escape from the nearest predator, which must move to its position in order to capture it. Starting positions are randomized across the map, and movements occur alternatively for predators and prey, but simultaneously for members of the same team. Collisions incur penalties and randomly place predators. The environment provides homogeneous predators with full local observations. A single agent cannot complete the task, since predators and prey move at the same speed. Instead, a coordinated behavior by at least two predators is necessary for each capture.

Tests were conducted on 7-unit wide maps, with $J = 2$ agents and two items or prey, as shown in Figure 3.7, where tabular Q-learning can still converge to a successful policy within reasonable time. The tests used a fully-connected neural network with two ReLU-activated

hidden layers of 50 nodes, a replay memory $\mathcal{D}$ with a capacity of $N = 5000$ transitions, a discount factor $\gamma = 0.9$, a network update period $\tau = 500$, a mini-batch size $k = 32$, and learning rate $\eta = 0.001$. The exploration rate ϵ was annealed from 1 to 0.1 during the first 75% steps. The Adam Optimizer [96] was used for optimization, and Glorot initialization [92] for the weights' initial values. The networks were trained for 200 thousand steps in the Foraging task and 100 thousand steps in the Pursuit game. The parameters for tabular Q-learning were a discount factor $\gamma = 0.9$ and a learning rate $\eta = 0.01$.

Tests were also conducted on 15-unit wide maps, too large for tabular Q-learning, with $J = 4$ agents, and up to ten items or six prey, as shown in Figure 4.1. For these tests, two ReLU-activated hidden layers of 250 neurons and a learning rate $\eta = 0.0001$ were used. Training took ten times longer, and a trainer was used to guide the initial exploration phase, by guiding the agents or handicapping the prey with some probability. We could not find parameters for tabular Q-learning that led to successful policies.



(a) Foraging Task. (b) Pursuit Task.

Figure 4.1: An example of the 15-wide maps for the Foraging and Pursuit environments.

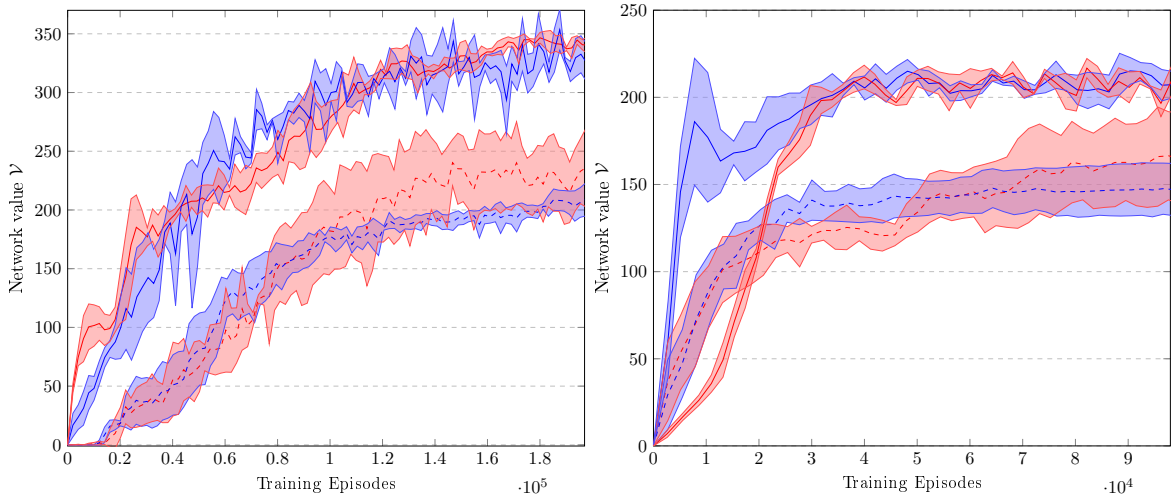
4.3.1 Joint-Action Learners and Independent Learners

This section starts by comparing the performance of policies learned with the JAL and IL approaches in both environments. Agent observations are handled as 1-hot global grids representing the map, with dimension $w \times h \times m$, where w is the map width, h the map height, and m the amount of entity types (obstacle, base, item/prey, agent). Agents have $|A| = 7$ possible actions in the Foraging task and $|A| = 5$ in the Pursuit game, as described in Section 3.4.2, and all agents receive 100 points when an item is caught or delivered, or when a prey is caught.

Based on the DQN analysis [26], a value $\mathcal{V} = \frac{1}{T}\sum_{t=1}^{T} \max_a Q(s_t, a; \theta)$ metric was used to determine the learning performance for our tests, which corresponds to the average Q-value of the best action in all T steps of a test simulation. The initial state of the test simulation

was chosen from a set of randomly-generated initial states, and represents a commonly found scenario. Maintaining a fixed test simulation allows for a controlled analysis of the evolution of the expectations of the network's value function over-time.

In the 7-wide maps, both tabular Q-learning and MaDDQN converged to policies with more than 99% success rate over 1000 games, and each algorithm's value estimation $\mathcal{V}$ is shown in Figure 4.2. Both IL and JAL have similar learning curves in the Foraging task, which does not require strong coordination among agents (a single agent can complete the task). In other words, there are no major benefits over using the more complex and restrictive JAL approach. On the Pursuit game, however, JAL has a steeper curve, due to the strong coordination requirements of the environment. Despite this, both algorithms converge to policies where agents surround each prey until it is eliminated, before moving on to the next.



(a) Foraging task.

(b) Pursuit game.

Figure 4.2: Evolution of MaDDQN policies with 7-wide maps and $J = 2$ agents. The plots represent the average (over three episodes) $\mathcal{V}$ and standard deviation estimated by the team during the training phase, given by IL (red) and JAL (blue) strategies with MaDDQN (solid) and tabular Q-learning (dashed).

In the 15-wide maps, the state-space was intractable for tabular Q-learning algorithms, which did not converge to successful policies. IL MaDDQN, on the other hand, achieved a success rate of more than 80% within the environments' time-limits, over 1000 games. This demonstrates that the IL variant of MaDDQN is adequate for environments with high-dimensional state-spaces, and can correctly approximate the Q-function in such cases. However, with $J = 4$ agents, JAL MaDDQN was unable to learn successful policies, as the joint-action becomes too complex at this point, due to growing exponentially with the larger amount of agents. To achieve successful policies with the JAL variant, the amount of agents was decreased to $J = 2$, as in the 7-wide maps.

4.3.2 Generalization - Harder Tasks

While it is a reasonable assumption that MaDDQN did not explore all the possible states while learning the policies, and can still generalize to new similar states during execution, how

well the policies adapt to similar, and previously unseen, tasks is also evaluated. To do so, the team's performance is directly measured when the task conditions are more complex.

Transfer learning approaches allow learning in one task to improve the learning performance in a related, but different, task [203]. They have been shown to be effective at speeding-up learning of new tasks, and a possible metric to measure their benefits is known as *jumpstarting*, where the initial performance of an agent in a target task is improved by transferring knowledge from a source task. In our case, we adapt agents on a harder task by copying the network weights of agents trained on a simpler task, providing a more accurate initial value estimation of each state-action pair.

The Foraging agents are evaluated on 7-wide maps, with the same amount of agents $J = 2$, and more items to catch. Pursuit agents are evaluated on 15-wide maps, with $J = 4$ or $J = 2$ predators for IL and JAL, respectively, and more prey to capture. This forces Foraging agents to coordinate better in order to efficiently navigate without colliding in a narrow map, and forces Pursuit agents to collectively decide which prey to chase based on the positions of their team. A Foraging episode is considered to be successful when all items are collected within the time-limit, and a Pursuit game to be successful when all prey are captured within the time-limit and no collisions between predators are found.

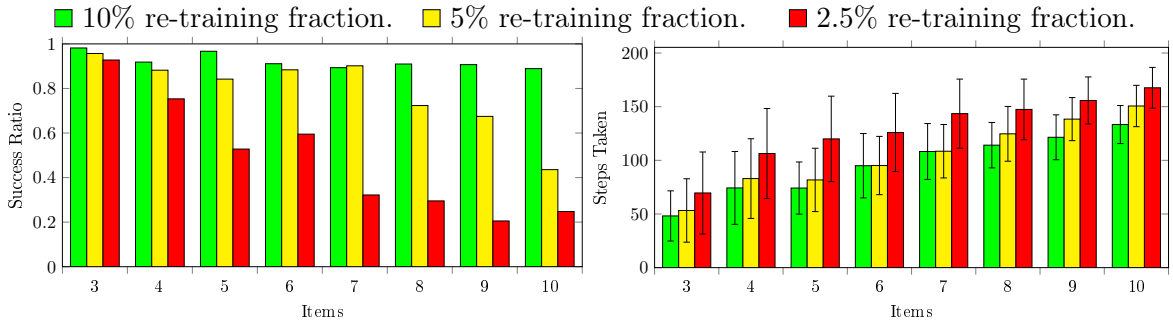
The previously learned policies' performance were evaluated after re-training them with a small fraction of the original training steps, 0% to 10% for each new task. For example, the original Foraging agents were trained with 200 thousand steps on environments with two items to forage, and re-trained with less than 20 thousand steps for environments with three to ten items to forage. Tabular Q-learning cannot generalize to new unseen tasks, since each new state generates a new row in the table that represents the Q-function. While deep learning methods can estimate the Q-value for any state (although inaccurately if the network generalizes poorly), Tabular Q-learning requires complete re-learning to explore most or all of the new states and achieve a similar success rate.

We found that, without any re-training, policies were overfit and could not successfully complete the new tasks, in either Foraging or Pursuit environments. However, Figure 4.3 shows that even a 10% re-training fraction allows Foraging agents to achieve an acceptable success rate, maintaining a success rate above 80% catching ten items in the same time-limit, in both IL and JAL variants.

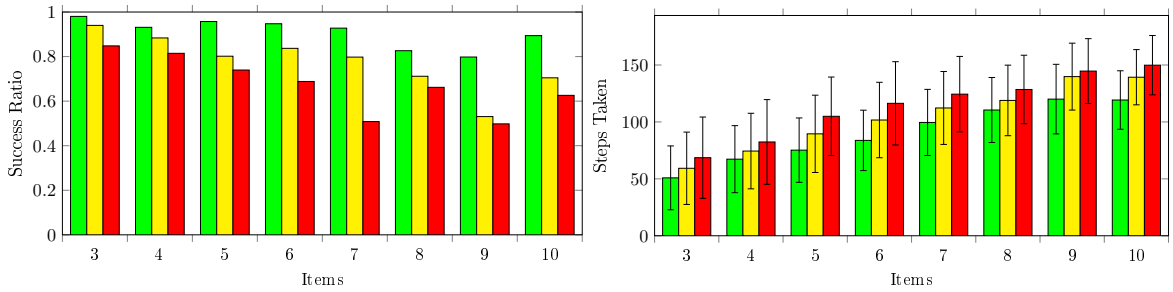
Policies actually improved in some of our tests, despite the increased complexity of the tasks. The optimal policy had not yet been learned in the original learning scenario, and only one of the agents was picking up berries, while the other simply stood on the corner to avoid collisions. With only a 2.5% re-training fraction, this sub-optimal strategy was kept. With 5% and 10% fractions, however, as the amount of items increased, the agents learned a better policy where both carry items. One of the agents goes foraging while the other depositing its item at the base, and they alternate these roles. This is shown in Figure 4.4. In other words, despite not having an optimal policy at the start of training, the team's policy improved when adapting to new harder tasks.

A similar analysis was conducted in the Pursuit game, as shown in Figure 4.5. Despite the increased map size and the larger amount of prey to capture, the IL policy with $J = 4$ predators maintains a 60% success rate with ten prey. On the other hand, the JAL policy with $J = 2$ predators was unable to generalize, even after the prey were handicapped to move at half the predator's speed. With only an additional two prey, the team's performance was worse than IL policies with ten prey.

Out results are an indication that, while JAL policies can generalize well in simpler envi-



(a) MaDDQN with the JAL variant.



(b) MaDDQN with the IL variant.

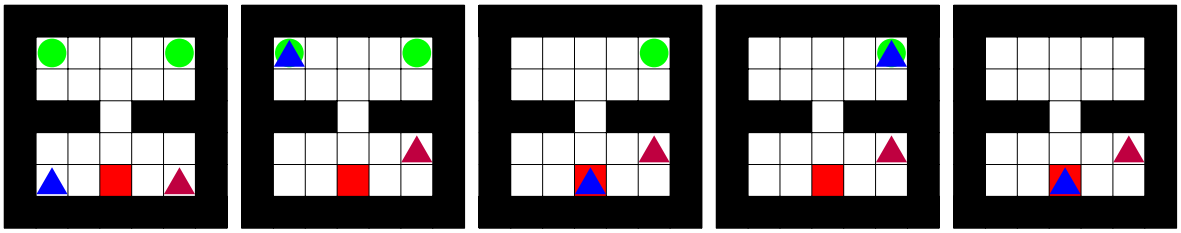
Figure 4.3: Evolution of MaDDQN policies in the Foraging task with 7-wide maps and $J = 2$ agents. The ratio of successful attempts over 1000 test simulations, and the average and standard deviation of the steps taken in them, when adapting to new Foraging scenarios with progressively more items. Policies were re-trained with a 10% (green), 5% (yellow), and 2.5% (red) fraction of the original training steps.

ronments, JAL is not an adequate solution in more complex environments, or environments with more agents. The IL approach can match the performance of JAL policies in simpler environments, and is able to achieve successful results in complex environments with larger teams, as well as when generalizing to unseen harder environments under the same conditions.

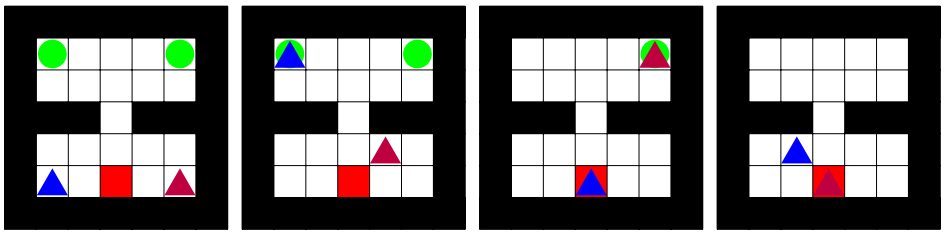
4.3.3 Generalization - Larger Teams

Unlike the JAL counterpart, the IL approach also allows a varying number of agents in the environment. This section thus evaluates the generalization of IL policies as the amount of agents performing a task is increased. The 7-wide Foraging scenario was originally trained with $J = 2$ agents and six items, while the 15-wide Pursuit task was trained with $J = 4$ predators and ten prey. As before, policies were re-trained with a fraction of the original training steps, and our results are shown in Figure 4.6.

Because the Foraging map has a narrow gap that only allows a single agent to move through it at a time, increasing the amount of agents just disturbs the overall team-performance. In other words, this environment represents a task that is not easily parallelized, and the team cannot take advantage of its distributed nature. In the Pursuit game, however, increasing the team's size speeds up the task completion, since more predators can capture the prey faster, but also decrease the task's success rate, since more collisions are now found between agents. Contrary to the Foraging environment, a larger team is able to parallelize its efforts



(a) A sub-optimal where one of the agents collects all items and his teammate simply avoids perturbing it.



(b) A better policy where agents switch roles and both alternate in collecting items.

Figure 4.4: An example of two policies in the Foraging task. After achieving a sub-optimal policy, agents trained on harder tasks improved and converged to a better policy than the original one.

---

and increase its speed when completing a distributed task.

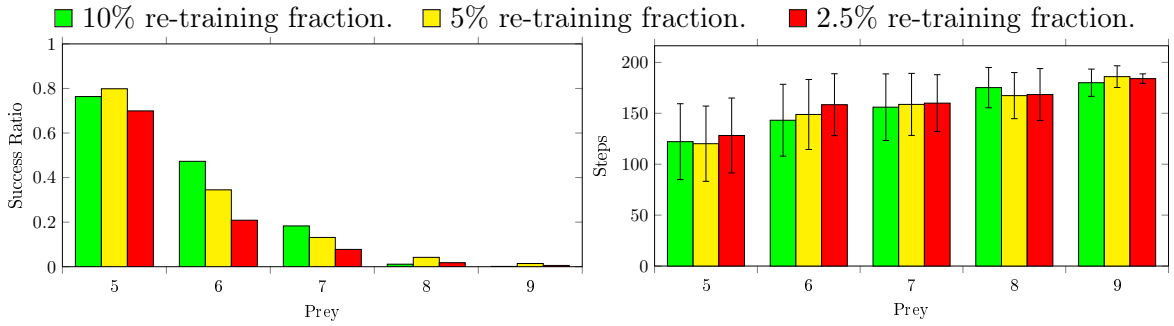
Our results indicate that the IL variant of MaDDQN can adapt to and incorporate larger teams and, when possible, parallelize the completion of a task. The JAL variant does not support varying amounts of agents, since a new network architecture would be required to incorporate a larger joint-action. However, independent agents allows a team to achieve implicit coordination even as the team's size changes.

4.4 Conclusion

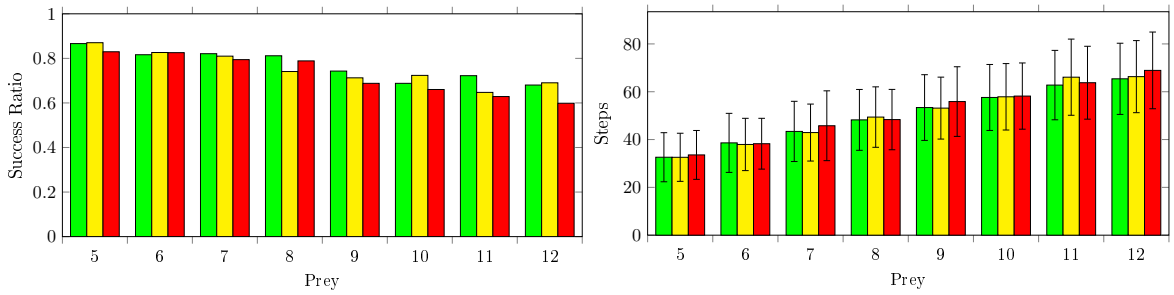
The Double Deep Q-Networks algorithm was formally extended to the multi-agent paradigm with Multi-agent Double Deep Q-Networks, and this chapter described two variants based on the Independent Learners and Joint-Action Learners techniques. Their viability for learning complex policies in multi-agent scenarios was analyzed, when compared with tabular Q-learning, and MaDDQN is shown to be able to handle high-dimensional state-spaces that tabular algorithms cannot. The IL variant can also scale to larger teams, unlike JAL, whose joint-action becomes exceedingly complex.

MaDDQN is also demonstrated to generalize to new unseen tasks, with a small fraction of the original training's steps. The IL variant again outperforms JAL, being able to generalize on more complex tasks, with a higher success rate. JAL policies were only able to generalize to new tasks in the simpler maps. IL policies can also handle a varying amount of agents and parallelize the efforts of the team on distributed tasks.

We conclude that the independent approach with implicit coordination is not only more generalizable, but also a more realistic solution than a centralized approach. Transfer learning techniques allow for gradually harder tasks to be learned, and IL MaDDQN policies can be trained in simple tasks with few agents, and then easily re-trained in harder environments



(a) MaDDQN with the JAL variant, $J = 2$ predators.



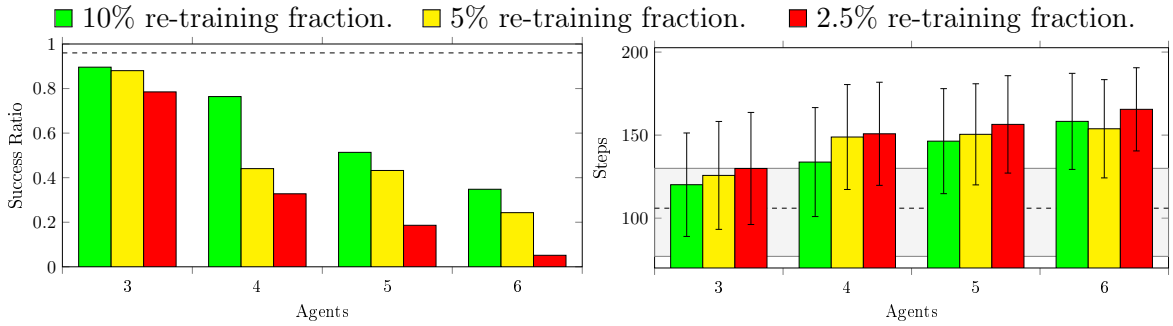
(b) MaDDQN with the IL variant, $J = 4$ predators.

Figure 4.5: Evolution of MaDDQN policies in the Pursuit game with 15-wide maps. The ratio of successful attempts over 1000 test simulations, and the average and standard deviation of the steps taken in them, when adapting to new Pursuit scenarios with progressively more items. Policies were re-trained with a 10% (green), 5% (yellow), and 2.5% (red) fraction of the original training steps.

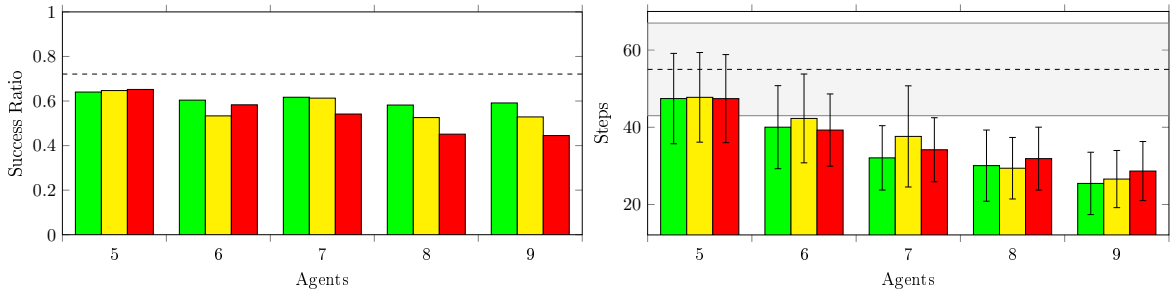
---

while taking advantage of larger team sizes. We do not explore whether learning a simple task and adapting to harder ones is more efficient than simply training *tabula rasa* on harder tasks. However, when hard tasks are prohibitively complex or when policies for simpler tasks are already available, transfer learning remain a viable solution.

MaDDQN ignores the non-stationary behavior of the environment, and falls under the *Ignore* category [31] of RL algorithms. It is also based on ϵ-greedy Q-learning, which does not support stochastic policies. In competitive environments, equilibrium policies are often stochastic, where each action is sampled with some probability, and MaDDQN could not achieve an equilibrium strategy in a simple game like rock-paper-scissors. Finally, the use of a replay memory has the undesirable effect of agents optimizing their policies with outdated samples. In a single-agent environment, the stationary environment of an agent has no concept of outdated samples, but in the multi-agent setting, an agent may adapt to older policies of other agents, and this may cause divergence between learned policies. Algorithms like A3C use multiple workers instead of a replay memory to break sample correlation, and do not suffer from this problem.



(a) MaDDQN with the IL variant in the Foraging task with 7-wide maps and $J = 2$ agents.



(b) MaDDQN with the IL variant in the Pursuit game with 15-wide maps and $J = 4$ agents.

Figure 4.6: Evolution of MaDDQN policies. The ratio of successful attempts over 1000 test simulations, and the average and standard deviation of the steps taken in them, when adapting to new scenarios with progressively larger teams. Policies were re-trained with a 10% (green), 5% (yellow), and 2.5% (red) fraction of the original training steps. The dashed line represents the original training's baseline.

Chapter 5

Mixed-Policy Asynchronous Q-Learning

Algorithms belonging to the *Forget* category adapt to non-stationary behavior of other agents. Throughout their execution, agents learn and adjust their policies to coordinate with other agents. Agents also continuously forget previous information regarding the evolution of other agents' policies. In other words, *Forget* algorithms do not model allies or opponents. Algorithms often strive to achieve NE strategies, where no agent can do better by changing its own strategy. When *Forget* algorithms can do so in self-play, they have the *rational* and *convergent* properties.

Forget-category algorithms include WoLF-PHC [51], GIGA-WoLF [52], WPL [54], and EMA-QL [53], all of which exhibit such properties, while each agent only has information about its own actions and rewards. They keep track of Q-values and maintain a probability distribution over possible actions. WoLF-PHC keeps track of two different policies, and has two different policy update rates to be set *a priori*. GIGA-WoLF also maintains two different policies, but only requires a single policy update rate to be defined. To respect the WoLF principle, GIGA-WoLF updates both policies at different rates, with a constant ratio. WPL has a variable learning rate, and allows agents to move towards the equilibrium strategy faster than moving away from it. EMA-QL again requires two different policy update rates to be defined, but it is shown to outperform WPL.

These mixed-policy algorithms store their values in table representations, and thus cannot be used in high-dimensional, noisy, or continuous environments. This chapter extends these algorithms, all of which require no knowledge about other players' actions or rewards and have been shown to reach equilibrium strategies in self-play, to the deep learning paradigm. Deep-learning implementations based on Asynchronous Deep Q-Learning not only match the performance of the original algorithms in single-state games, but can also find equilibrium strategies in complex environments with continuous or noisy state-spaces. These findings were published in the EPIA17 conference [56]. The source-code for all algorithms and tests was published at `https://github.com/david-simoes-93/Mixed-Policy-Asynchronous-Deep-Q-Learning`.

5.1 Problem Statement

While several algorithms have been proven to converge to Nash equilibria [51, 52], many have unrealistic assumptions, such as knowing the underlying game structure or the optimal

Nash Equilibrium [204, 47], or the actions performed by other agents and their received rewards [48, 49]. These assumptions are unrealistic in most scenarios, either due to their complexity (where the game model is unknown or too complex) or due to conflicting goals (non-cooperative agents will not provide reward information to others, for example). This makes it so that even simple 2-player games are challenging.

However, algorithms have been proposed that achieve Nash equilibriums with only information about the agents' own actions and rewards. These include WoLF-PHC [51], GIGA-WoLF [52], WPL [54], and EMA-QL [53]. WoLF-PHC introduced the *Win or Learn Fast* principle, where different learning rates are used when the agent is winning or losing, a principle also used by GIGA-WoLF. However, both algorithms have shown problems in more complex games, such as Shapley's Game [143]. WPL and EMA-QL have been shown to achieve convergence in such games, but with some setbacks. WPL has no formal analysis and proof of convergence, and EMA-QL features some difficulties learning simpler games with many actions and asymmetric probabilities. Not only that, but since all of these algorithms derive from table-based Q-learning, they also cannot handle high-complexity environments. These algorithms are formally described in Section 2.5.1.

Because of their tabular nature, these algorithms do not handle high-dimensional state-spaces, which become intractable without approximating the Q-value function. They also do not support continuous state-space environments without specific techniques, and they do not generalize to new unseen states. For example, in competitive Pokémon battling, there are hundreds of possible Pokémon and thousands of attack and typing combinations in a partially-observable environment. Despite the tabular algorithm's *rationality* and *convergence* properties in single-state games, none of them are adequate for complex competitive multi-agent environments.

5.2 Proposal

The mixed policy algorithms described in the previous section are now extended to the deep learning paradigm through the Asynchronous 1-step Q-learning (A1Q) framework, as described in Algorithm 7. This adaptation will allow algorithms to approximate the Q-function and their policy function in complex, continuous, or noisy state-space environments. We refer to the deep learning extension of WoLF-PHC as PHCθ, GIGA-WoLF's as GIGAθ, WPL's as WPLθ, and EMA-QL's as EMAθ.

Both A1Q and DQN were adequate candidates for a deep Q-learning basis for the mixed policy algorithms, which are based on tabular Q-learning. However, DQN contains an *experience replay* $\mathcal{D}$ to break its sample dependencies, while A1Q uses asynchronous updates. This implies that DQN incurs a delay, directly proportional to the size of its *experience replay*, for agents to adapt to other agents' strategies, which can lead to diverging policies [153]. A1Q is more complex, but due to its frequent network synchronizations, does not suffer from this problem.

PHCθ, GIGAθ, WPLθ, and EMAθ have identical Q-value updates, and their new policy update equations can simply use the Q-values $Q(s_t, a_t; \theta)$ output by the Q-network with weights θ for state s_t and action a_t at time-step t, instead of their tabular counterparts $Q(s_t, a_t)$. Each algorithm computes its new policy $\pi_{t+1}(s_t)$ accordingly, which is used as the optimization target y_π for a policy network with weights θ_π. While Q-networks are updated with the mean squared error function, policy networks are optimized with the cross entropy

$H(y_\pi, \pi(s_t, \theta_\pi))$ between the current output policy π and the optimization target y_π. Both PHCθ and GIGAθ additionally compute another policy $\hat{\pi}_{t+1}(s_t)$, which is updated slower, and is represented by an average policy network with weights $\theta_{\hat{\pi}}$.

For computational efficiency, agents can use intra-agent parameter sharing and optimize a single network with multiple output layers (the Q-value estimation and the policy), as shown in Figure 5.1. This allows policies and Q-values to be computed in a single feed-forward pass, and the networks to be optimized in a single backward pass. Since network updates are often multiplied by a learning rate η, in order to use intra-agent parameter sharing, the policy update rates δ of each algorithm are defined with respect to η. As an example, consider a tabular mixed policy algorithm where Q-values would be updated with a learning rate x, and the policy with an update rate y. The deep learning implementation could then have a learning rate $\eta = x$, and a policy update rate $\delta = \frac{y}{\eta}$, where target policies are calculated using δ, and network optimizations multiplicatively use η to optimize their weights.

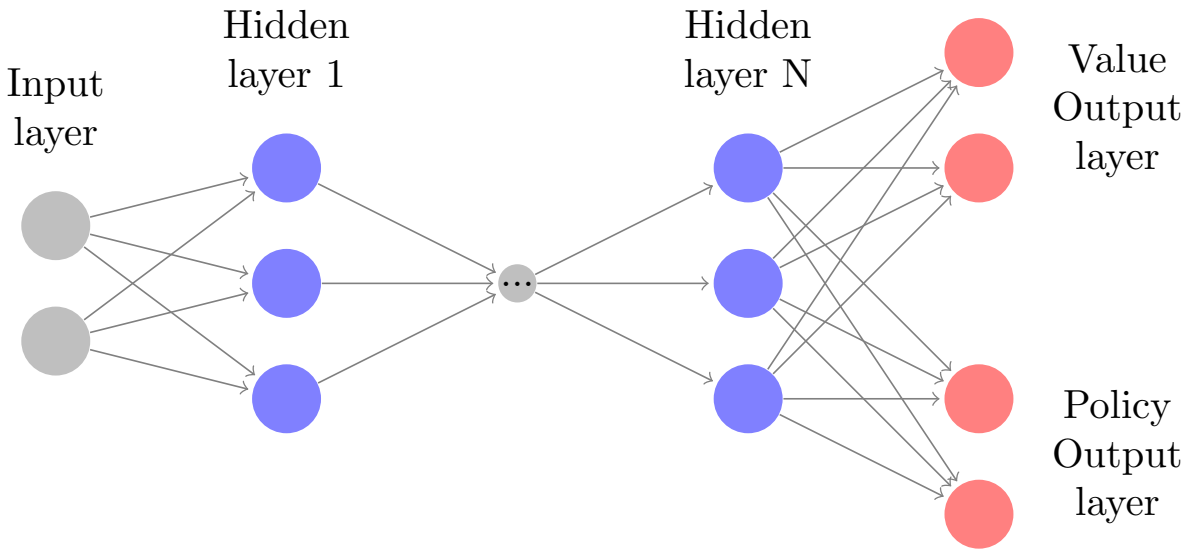


Figure 5.1: An example of a deep neural network using intra-agent parameter sharing, with a fully-connected architecture, N hidden layers, and two output layers. Each node is connected to all the nodes in the next layer and both outputs are computed with a single forwards-pass. To optimize the network, the error of both output layers is summed and propagated in a single backwards-pass.

5.2.1 Update Rules

This section formally describes the deep asynchronous algorithms PHCθ, GIGAθ, WPLθ, and EMAθ, as well as their new update rules.

PHCθ

For PHCθ, a game counter C_t is used instead of a state counter $C_t(s_t)$, incremented at each episode. Because table-based representations are no longer part of the algorithm, state-wide counters are infeasible. Like WoLF-PHC, PHCθ starts by computing the target $y_{\hat{\pi}}$ for the

average policy.

$$y_{\hat{\pi}} = \hat{\pi}_t(s_t;\theta^-) + \frac{\pi_t(s_t;\theta^-) - \hat{\pi}_t(s_t;\theta^-)}{C_{t+1}} \tag{5.1}$$

The winning or losing learning rates are chosen based on the current value estimation, and whether the current policy $\pi_t(s_t;\theta^-)$ outperforms the average policy $\hat{\pi}_t(s_t;\theta^-)$. The target y_π for the current policy is then given by an added increment $\Delta_t(s_t, a_t)$.

$$\delta_t(s_t) = \begin{cases} \delta_w \text{ if } \sum_{a' \in A} \pi_t(s_t, a';\theta^-) Q_t(s_t, a';\theta^-) > \sum_{a' \in A} \hat{\pi}_t(s_t, a';\theta^-) Q_t(s_t, a';\theta^-) \\ \delta_l \text{ otherwise} \end{cases} \tag{5.2}$$

$$\forall a \in A \ \Delta_t(s_t, a_t) = \begin{cases} -\frac{\delta_t(s_t)}{|A|-1} \text{ if } a \neq \text{argmax}_{a' \in A} Q_t(s_t, a';\theta^-) \\ \delta_t(s_t) \text{ otherwise} \end{cases} \tag{5.3}$$

$$y_\pi = P\left(\pi_t(s_t;\theta^-) + \Delta_t(s_t)\right) \tag{5.4}$$

Using these equations, PHCθ is described in Algorithm 11. Two policy networks with weights θ_π and $\theta_{\hat{\pi}}$ are used, and two policy update rates δ_w and δ_l to adhere to the WoLF principle. Both the policy networks and the standard Q-network have on-line and target versions, where the target networks are updated at a slower pace. This prevents network targets from diverging and increases the stability of converging policies.

GIGAθ

At each time-step, GIGAθ estimates a policy $\pi^-_{t+1}(s_t)$ and the target $y_{\hat{\pi}}$ for the average policy.

$$\pi^-_{t+1}(s_t) = P\left(\pi_t(s_t;\theta^-) + \delta Q_t(s_t;\theta^-)\right) \tag{5.5}$$

$$y_{\hat{\pi}} = P\left(\hat{\pi}_t(s_t;\theta^-) + \frac{\delta Q_t(s_t;\theta^-)}{3}\right) \tag{5.6}$$

The algorithm then computes a learning rate δ_t based on whether the average policy $\hat{\pi}_t$ outperformed π_t, and computes the policy target y_π.

$$\delta_t = \min\left(1, \frac{||y_{\hat{\pi}} - \hat{\pi}_t(s_t)||}{||y_{\hat{\pi}} - \pi^-_{t+1}(s_t;\theta^-)||}\right) \tag{5.7}$$

$$y_\pi = (1-\delta_t)\pi^-_{t+1}(s_t;\theta^-) + (\delta_t) y_{\hat{\pi}} \tag{5.8}$$

Using these equations, GIGAθ is described in Algorithm 12. Like PHCθ, two policy networks with weights θ_π and $\theta_{\hat{\pi}}$ are used, but only a single policy update rate δ is required. The main difference between PHCθ and GIGAθ is their policy update rules.

Input: Global shared learning rate η, discount factor γ, target network update period τ, on-line Q-network weights θ, target Q-network weights θ^-, on-line policy network weights θ_π, target policy network weights θ_π^-, on-line average policy network weights $\theta_{\hat{\pi}}$, target average policy network weights $\theta_{\hat{\pi}}^-$, exploration rate ϵ, policy update rates δ_w and δ_l, and maximum iterations $T_{\max}$. Locally, on-line value network weights ϑ, on-line policy network weights ϑ_π, on-line average policy network weights $\vartheta_{\hat{\pi}}$, and time-step counter t.

$t \leftarrow 0$
for iteration $T \leftarrow 0, T_{max}$ **do**
  Reset gradients $d\theta \leftarrow 0$, $d\theta_\pi \leftarrow 0$, and $d\theta_{\hat{\pi}} \leftarrow 0$
  Synchronize $\vartheta \leftarrow \theta$, $\vartheta_\pi \leftarrow \theta_\pi$, and $\vartheta_{\hat{\pi}} \leftarrow \theta_{\hat{\pi}}$
  $t_{start} \leftarrow t$
  Sample state s_t
  repeat
    Select random action a_t with probability ϵ, otherwise sample action a_t according to policy $\pi(a_t|s_t, \vartheta_\pi)$
    Execute a_t
    Sample state s_{t+1} and reward r_t
    Compute target $y \leftarrow \begin{cases} r \text{ for terminal state} \\ r + \gamma \max_a Q(s_{t+1}, a, \theta^-) \text{ otherwise} \end{cases}$ with target Q-network
    Compute targets y_π and $y_{\hat{\pi}}$ with WoLF-PHC's update equations
    Compute loss $L \leftarrow (y - Q(s_t, a_t, \vartheta))^2$ of local on-line Q-network
    Compute loss $L_\pi \leftarrow H(y_\pi, \pi(s_t, \vartheta_\pi))$ of local on-line policy network
    Compute loss $L_{\hat{\pi}} \leftarrow H(y_{\hat{\pi}}, \pi(s_t, \vartheta_{\hat{\pi}}))$ of local on-line average policy network
    Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L}{\partial \vartheta}$
    Accumulate gradients $d\theta_\pi \leftarrow d\theta_\pi + \eta \frac{\partial L_\pi}{\partial \vartheta_\pi}$
    Accumulate gradients $d\theta_{\hat{\pi}} \leftarrow d\theta_{\hat{\pi}} + \eta \frac{\partial L_{\hat{\pi}}}{\partial \vartheta_{\hat{\pi}}}$
    $t \leftarrow t + 1$
  until terminal s_{t+1}
  Update global on-line networks weights $\theta \leftarrow \theta + d\theta$, $\theta_\pi \leftarrow \theta_\pi + d\theta_\pi$, and $\theta_{\hat{\pi}} \leftarrow \theta_{\hat{\pi}} + d\theta_{\hat{\pi}}$
  Update target networks weights $\theta^- \leftarrow \theta$, $\theta_\pi^- \leftarrow \theta_\pi$, and $\theta_{\hat{\pi}}^- \leftarrow \theta_{\hat{\pi}}$, every τ time-steps
end for

Output: A converged Q-network with weights θ to approximate the value function as $Q(s, a, \theta)$, and a converged policy network with weights θ_π to approximate the value function as $\pi(s, a, \theta_\pi)$.

Algorithm 11: Pseudo-code for a worker thread running PHCθ using ϵ-greedy exploration.

WPLθ

At each time-step, WPLθ calculates an increment vector $\Delta(s_t)$ based on the value function $V_t(s_t)$ and the policy learning rate δ.

$$V_t(s_t) = \sum_{a \in A} \pi_t(s_t, a; \theta^-) Q_t(s_t, a; \theta^-) \tag{5.9}$$

$$\forall a \in A \ \Delta_t(s_t, a) = \delta \frac{\partial V_t}{\partial \pi_t(s_t, a; \theta^-)} \begin{cases} \pi_t(s_t, a; \theta^-) \text{ if } \frac{\partial V_t}{\partial \pi_t(s_t, a; \theta^-)} < 0 \\ 1 - \pi_t(s_t, a; \theta^-) \text{ otherwise} \end{cases} \tag{5.10}$$

This vector is then used to compute the target y_π for the agent's policy, where each action must have a non-zero probability α.

$$y_\pi = P\left(\pi_t(s_t; \theta_\pi^-) + \Delta_t(s_t), \alpha \right) \tag{5.11}$$

Using these equations, WPLθ is described in Algorithm 13. Unlike the previous algorithms, a single policy network θ_π is used in conjunction with a Q-network. Of all four described algorithms in this chapter, WPLθ is the one that requires the least hyper-parameters.

Input: Globally, shared learning rate η, discount factor γ, target network update period τ, on-line Q-network weights θ, target Q-network weights θ^-, on-line policy network weights θ_π, target policy network weights θ_π^-, on-line average policy network weights $\theta_{\hat{\pi}}$, target average policy network weights $\theta_{\hat{\pi}}^-$, exploration rate ϵ, a policy update rate δ, and maximum iterations $T_{\max}$. Locally, on-line value network weights ϑ, on-line policy network weights ϑ_π, on-line average policy network weights $\vartheta_{\hat{\pi}}$, and time-step counter t.

$t \leftarrow 0$
for iteration $T \leftarrow 0, T_{max}$ **do**
  Reset gradients $d\theta \leftarrow 0$, $d\theta_\pi \leftarrow 0$, and $d\theta_{\hat{\pi}} \leftarrow 0$
  Synchronize $\vartheta \leftarrow \theta$, $\vartheta_\pi \leftarrow \theta_\pi$, and $\vartheta_{\hat{\pi}} \leftarrow \theta_{\hat{\pi}}$
  $t_{start} \leftarrow t$
  Sample state s_t
  repeat
    Select random action a_t with probability ϵ, otherwise sample action a_t according to policy $\pi(a_t|s_t, \vartheta_\pi)$
    Execute a_t
    Sample state s_{t+1} and reward r_t
    Compute target $y \leftarrow \begin{cases} r \text{ for terminal state} \\ r + \gamma \max_a Q(s_{t+1}, a, \theta^-) \text{ otherwise} \end{cases}$ with target Q-network
    Compute targets y_π and $y_{\hat{\pi}}$ with GIGA-WoLF's update equations
    Compute loss $L \leftarrow (y - Q(s_t, a_t, \vartheta))^2$ of local on-line Q-network
    Compute loss $L_\pi \leftarrow H(y_\pi, \pi(s_t, \vartheta_\pi))$ of local on-line policy network
    Compute loss $L_{\hat{\pi}} \leftarrow H(y_{\hat{\pi}}, \pi(s_t, \vartheta_{\hat{\pi}}))$ of local on-line average policy network
    Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L}{\partial \vartheta}$
    Accumulate gradients $d\theta_\pi \leftarrow d\theta_\pi + \eta \frac{\partial L_\pi}{\partial \vartheta_\pi}$
    Accumulate gradients $d\theta_{\hat{\pi}} \leftarrow d\theta_{\hat{\pi}} + \eta \frac{\partial L_{\hat{\pi}}}{\partial \vartheta_{\hat{\pi}}}$
    $t \leftarrow t + 1$
  until terminal s_{t+1}
  Update global on-line networks weights $\theta \leftarrow \theta + d\theta$, $\theta_\pi \leftarrow \theta_\pi + d\theta_\pi$, and $\theta_{\hat{\pi}} \leftarrow \theta_{\hat{\pi}} + d\theta_{\hat{\pi}}$
  Update target networks weights $\theta^- \leftarrow \theta$, $\theta_\pi^- \leftarrow \theta_\pi$, and $\theta_{\hat{\pi}}^- \leftarrow \theta_{\hat{\pi}}$, every τ time-steps
end for

Output: A converged Q-network with weights θ to approximate the value function as $Q(s, a, \theta)$, and a converged policy network with weights θ_π to approximate the value function as $\pi(s, a, \theta_\pi)$.

Algorithm 12: Pseudo-code for a worker thread running GIGAθ using ϵ-greedy exploration.

EMAθ

At each time-step, EMAθ calculates an increment vector $\vec{\Delta}(s)$. Unlike the previous algorithms, a single policy network θ_π is used in conjunction with a Q-network.

$$\delta_t(s_t, a_t) = \begin{cases} \delta_w \text{ if } a_t = \text{argmax}_{a'} Q_t(s_t, a'; \theta^-) \\ \delta_l \text{ otherwise} \end{cases} \tag{5.12}$$

$$\vec{\Delta}_1(s_t) = (u_0, u_1, \ldots, u_{|A|}) \text{ where } u_a = \begin{cases} 1 \text{ if } a = \text{argmax}_{a'} Q_t(s_t, a'; \theta^-) \\ 0 \text{ otherwise} \end{cases} \tag{5.13}$$

$$\vec{\Delta}_2(s_t) = (u_0, u_1, \ldots, u_{|A|}) \text{ where } u_a = \begin{cases} 0 \text{ if } a = \text{argmax}_{a'} Q_t(s_t, a'; \theta^-) \\ \frac{1}{|A|-1} \text{ otherwise} \end{cases} \tag{5.14}$$

$$\vec{\Delta}(s_t) = \begin{cases} \vec{\Delta}_1(s_t) \text{ if } a_t = \text{argmax}_{a'} Q_t(s, a'; \theta^-) \\ \vec{\Delta}_2(s_t) \text{ otherwise} \end{cases} \tag{5.15}$$

Similarly to WPLθ, this vector is then used to compute the target y_π for the agent's policy.

$$y_\pi = (1 - \delta_t)\pi_t(s_t; \theta^-) + \delta_t \vec{\Delta}(s_t) \tag{5.16}$$

Input: Globally, shared learning rate η, discount factor γ, target network update period τ, on-line Q-network weights θ, target Q-network weights θ^-, on-line policy network weights θ_π, target policy network weights θ_π^-, exploration rate ϵ, policy update rate δ, and maximum iterations $T_{\max}$. Locally, on-line value network weights ϑ, on-line policy network weights ϑ_π, and time-step counter t.

$t \leftarrow 0$
for iteration $T \leftarrow 0, T_{max}$ **do**
  Reset gradients $d\theta \leftarrow 0$, and $d\theta_\pi \leftarrow 0$
  Synchronize $\vartheta \leftarrow \theta$, and $\vartheta_\pi \leftarrow \theta_\pi$
  $t_{start} \leftarrow t$
  Sample state s_t
  repeat
    Select random action a_t with probability ϵ, otherwise sample action a_t according to policy $\pi(a_t|s_t, \vartheta_\pi)$
    Execute a_t
    Sample state s_{t+1} and reward r_t
    Compute target $y \leftarrow \begin{cases} r \text{ for terminal state} \\ r + \gamma \max_a Q(s_{t+1}, a, \theta^-) \text{ otherwise} \end{cases}$ with target Q-network
    Compute targets y_π with WPL's update equations
    Compute loss $L \leftarrow (y - Q(s_t, a_t, \vartheta))^2$ of local on-line Q-network
    Compute loss $L_\pi \leftarrow H(y_\pi, \pi(s_t, \vartheta_\pi))$ of local on-line policy network
    Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L}{\partial \vartheta}$
    Accumulate gradients $d\theta_\pi \leftarrow d\theta_\pi + \eta \frac{\partial L_\pi}{\partial \vartheta_\pi}$
    $t \leftarrow t + 1$
  until terminal s_{t+1}
  Update global on-line networks weights $\theta \leftarrow \theta + d\theta$ and $\theta_\pi \leftarrow \theta_\pi + d\theta_\pi$
  Update target networks weights $\theta^- \leftarrow \theta$ and $\theta_\pi^- \leftarrow \theta_\pi$ every τ time-steps
end for

Output: A converged Q-network with weights θ to approximate the value function as $Q(s, a, \theta)$, and a converged policy network with weights θ_π to approximate the value function as $\pi(s, a, \theta_\pi)$.

Algorithm 13: Pseudo-code for a worker thread running WPLθ using ϵ-greedy exploration.

Using these equations, EMAθ is described in Algorithm 14. A single policy network with weights θ_π is updated with two distinct policy update rates δ_w and δ_l.

5.3 Evaluation

Several games are chosen to demonstrate whether the deep learning implementations of these algorithms can still converge to NE strategies. The chosen games include Matching Pennies, Tricky Game, Biased Game, Rock-Paper-Scissors (RPS), and Null-Rock-Paper-Scissors (NRPS), described in Section 3.3. These games represent a set of diverse two-player single-state games, with different amounts of actions, both symmetric and asymmetric NE, and different average returns.

The equilibrium strategies for Matching Pennies and Tricky Game is to play each action with a probability of $\frac{1}{2}$. For the Biased Game, it is to play one action with probability $\frac{3}{4}$ and the other with probability $\frac{1}{4}$. For Rock-Paper-Scissors, the Nash equilibrium is to play each action with a probability of $\frac{1}{3}$ and for Null-Rock-Paper-Scissors, the equilibrium is to not play the first action and the remaining actions with probability $\frac{1}{3}$.

5.3.1 Tabular Rationality and Convergence

An initial comparison is performed between the original tabular versions of WoLF-PHC, GIGA-WoLF, WPL, and EMA-QL. The same hyper-parameters are kept for all algorithms, with a learning rate $\eta = 0.01$, a policy update rate $\delta = \frac{\eta}{100 + i/2000}$, the winning policy learning rate $\delta_w = \delta$, and the losing policy learning rate $\delta_l = 2\delta_w$. This comparison is a fair evaluation of

Input: Globally, shared learning rate η, discount factor γ, target network update period τ, on-line Q-network weights θ, target Q-network weights θ^-, on-line policy network weights θ_π, target policy network weights θ_π^-, exploration rate ϵ, policy update rates δ_w and δ_l, and maximum iterations $T_{\max}$. Locally, on-line value network weights ϑ, on-line policy network weights ϑ_π, and time-step counter t.

$t \leftarrow 0$
for iteration $T \leftarrow 0, T_{max}$ **do**
  Reset gradients $d\theta \leftarrow 0$, and $d\theta_\pi \leftarrow 0$
  Synchronize $\vartheta \leftarrow \theta$, and $\vartheta_\pi \leftarrow \theta_\pi$
  $t_{start} \leftarrow t$
  Sample state s_t
  repeat
    Select random action a_t with probability ϵ, otherwise sample action a_t according to policy $\pi(a_t|s_t, \vartheta_\pi)$
    Execute a_t
    Sample state s_{t+1} and reward r_t
    Compute target $y \leftarrow \begin{cases} r \text{ for terminal state} \\ r + \gamma \max_a Q(s_{t+1}, a, \theta^-) \text{ otherwise} \end{cases}$ with target Q-network
    Compute target y_π with EMA-QL's update equations
    Compute loss $L \leftarrow (y - Q(s_t, a_t, \vartheta))^2$ of local on-line Q-network
    Compute loss $L_\pi \leftarrow H(y_\pi, \pi(s_t, \vartheta_\pi))$ of local on-line policy network
    Accumulate gradients $d\theta \leftarrow d\theta + \eta \frac{\partial L}{\partial \vartheta}$
    Accumulate gradients $d\theta_\pi \leftarrow d\theta_\pi + \eta \frac{\partial L_\pi}{\partial \vartheta_\pi}$
    $t \leftarrow t + 1$
  until terminal s_{t+1}
  Update global on-line networks weights $\theta \leftarrow \theta + d\theta$ and $\theta_\pi \leftarrow \theta_\pi + d\theta_\pi$
  Update target network weights $\theta^- \leftarrow \theta$ and $\theta_\pi^- \leftarrow \theta_\pi$ every τ time-steps
end for

Output: A converged Q-network with weights θ to approximate the value function as $Q(s, a, \theta)$, and a converged policy network with weights θ_π to approximate the value function as $\pi(s, a, \theta_\pi)$.

Algorithm 14: Pseudo-code for a worker thread running EMAθ using ϵ-greedy exploration.

---

the performance of all algorithms in a varied set of single state games, since hyper-parameters are not tuned to any specific algorithm.

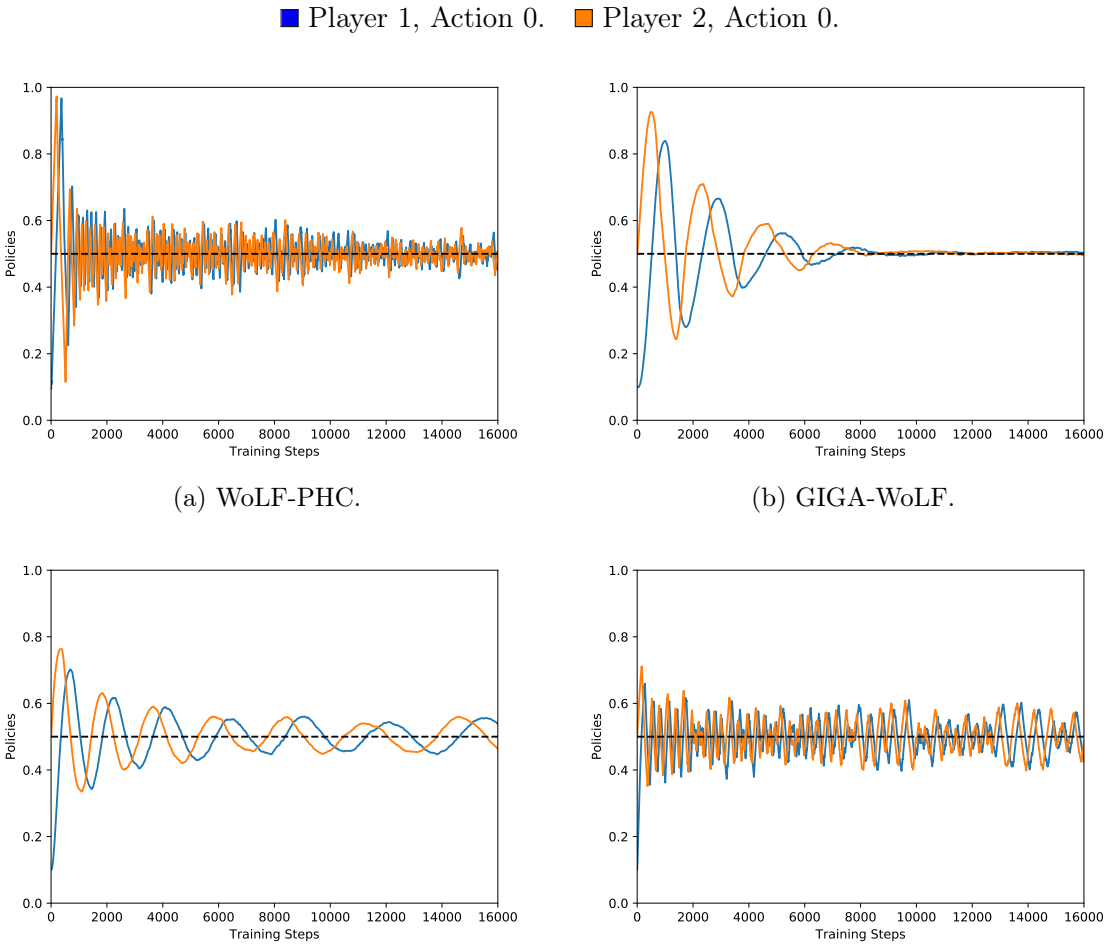
Figures 5.2, 5.3, 5.4, 5.5, and 5.6 show the policy evolutions for multiple games. WoLF-PHC shows a large variance in games with more than two actions, like RPS and NRPS. GIGA-WoLF converges in all scenarios and appears to be the algorithm with less overall variance. WPL also converges in all scenarios, but oscillates more than GIGA-WoLF in symmetric action games like Matching Pennies or NRPS. EMA-QL is unable to converge to the proper NE in a game with multiple asymmetric actions like NRPS.

Our analysis shows that both GIGA-WoLF and WPL are robust to different kinds of game theoretic environments and achieve convergence to NE policies in self-play. The same analysis is now conducted on the deep asynchronous versions of all the previous algorithms.

5.3.2 Deep Asynchronous Rationality and Convergence

The asynchronous mixed policy tests used 3 concurrent workers, a neural network with two hidden layers of 150 nodes with ELU activations [91], a learning rate $\eta = 10^{-4}$, a policy learning rate $\delta_\pi = \frac{\eta}{200}$, a winning policy learning rate $\delta_w = \delta_\pi$, and a losing policy learning rate $\delta_l = 2\delta_w$. Weights were initialized with the Glorot initializer [92] and optimized with the Adam optimizer [96].

Figures 5.7, 5.8, 5.9, 5.10, and 5.11 show the policy evolutions for multiple games. PHCθ now shows a large variance in most games, and its policies seem to diverge over time in RPS and NRPS, and never actually converge in the Tricky game. GIGAθ and WPLθ exhibit similar rational and convergent behavior. EMAθ is more unstable, remains unable to converge



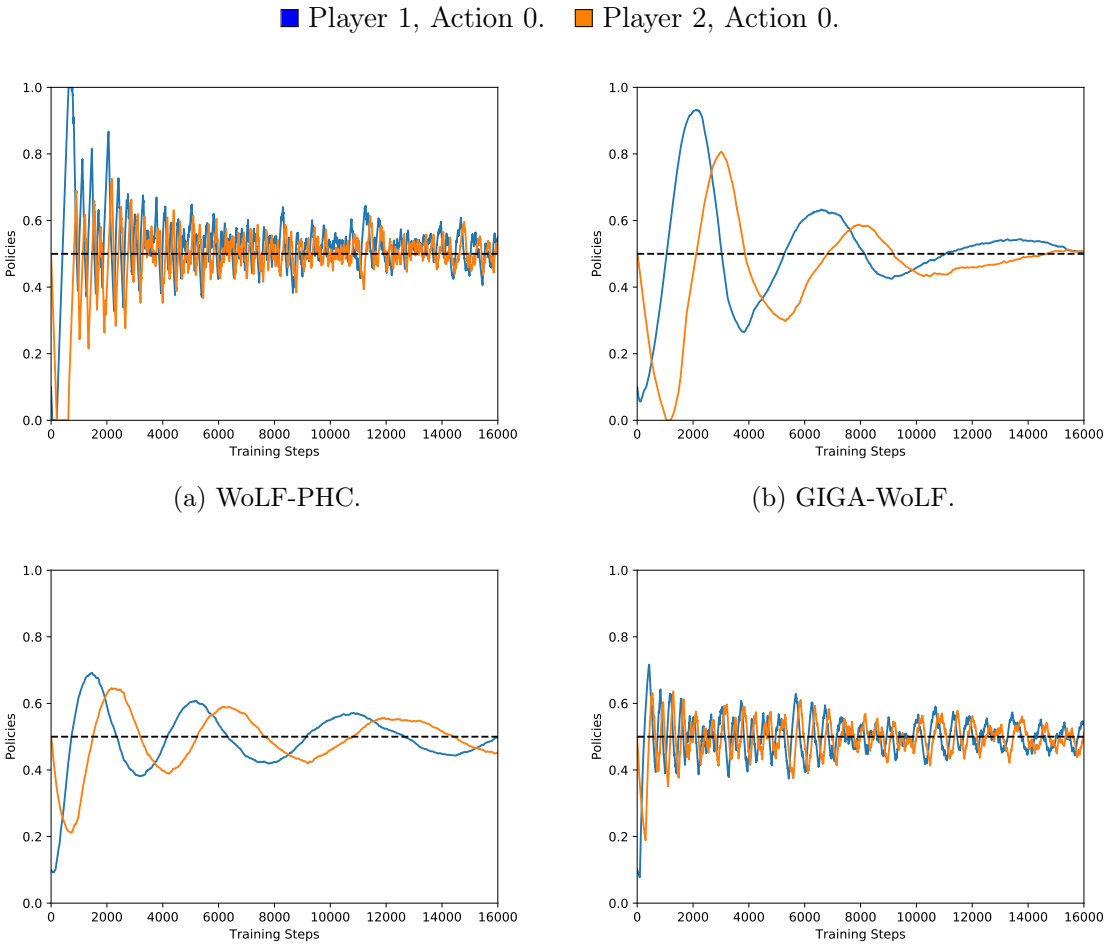
(a) WoLF-PHC.

(b) GIGA-WoLF.

(c) WPL.

(d) EMA-QL.

Figure 5.2: The evolution of the policies of 2 agents in self-play on the Matching Pennies game. Players use the tabular Wolf-PHC, GIGA-WoLF, WPL, and EMA-QL algorithms, and plots show the probability of choosing the first action over epochs of 25 time-steps.



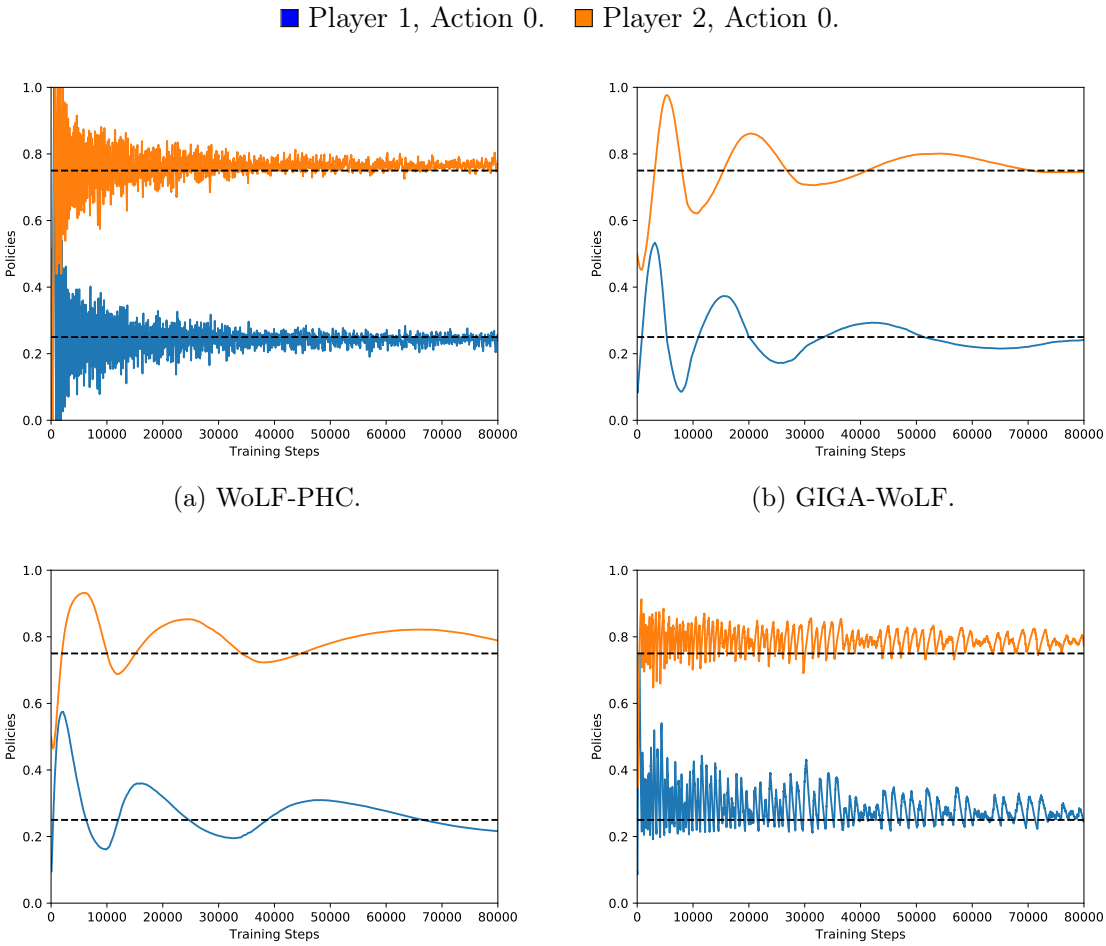
(a) WoLF-PHC.

(b) GIGA-WoLF.

(c) WPL.

(d) EMA-QL.

Figure 5.3: The evolution of the policies of 2 agents in self-play on the Tricky game. Players use the tabular Wolf-PHC, GIGA-WoLF, WPL, and EMA-QL algorithms, and plots show the probability of choosing the first action over epochs of 25 time-steps.



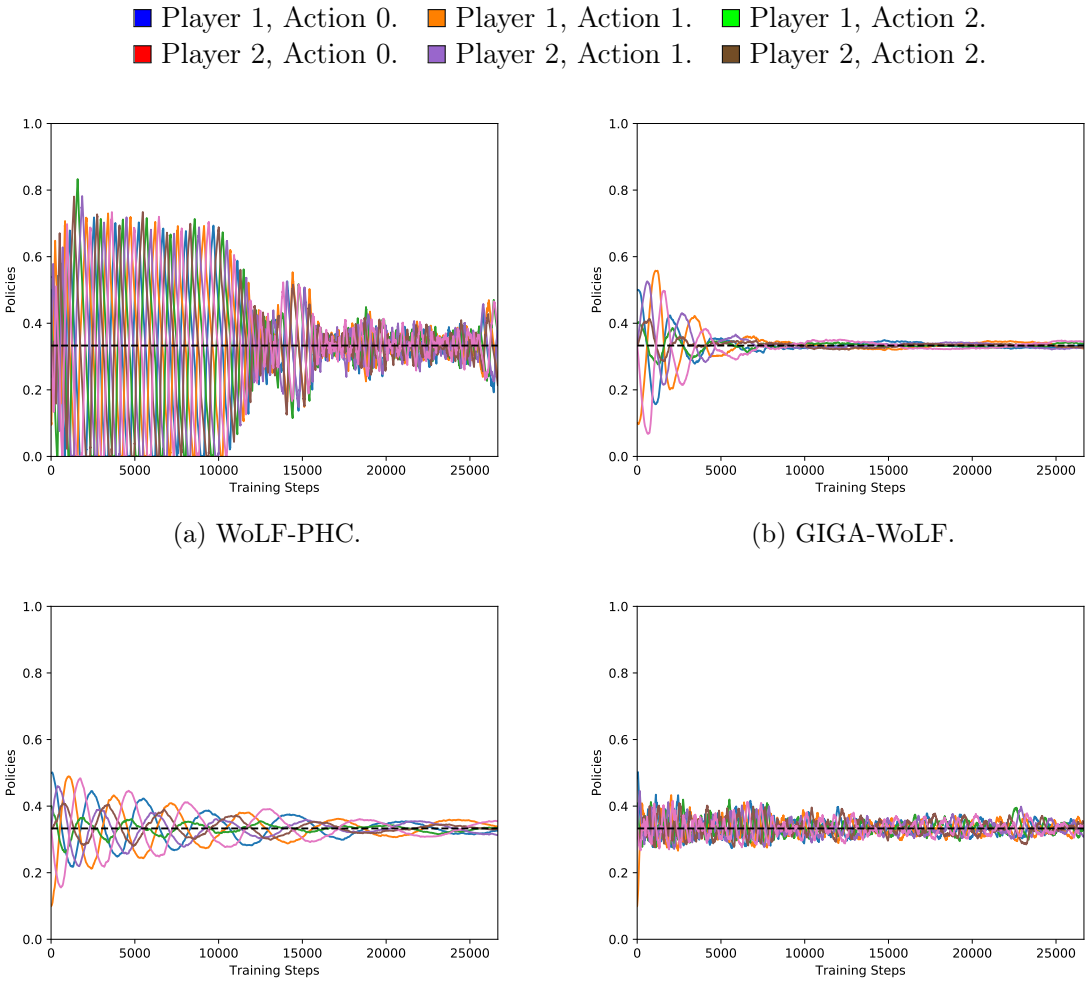
(a) WoLF-PHC.

(b) GIGA-WoLF.

(c) WPL.

(d) EMA-QL.

Figure 5.4: The evolution of the policies of 2 agents in self-play on the Biased game. Players use the tabular Wolf-PHC, GIGA-WoLF, WPL, and EMA-QL algorithms, and plots show the probability of choosing the first action over epochs of 25 time-steps.



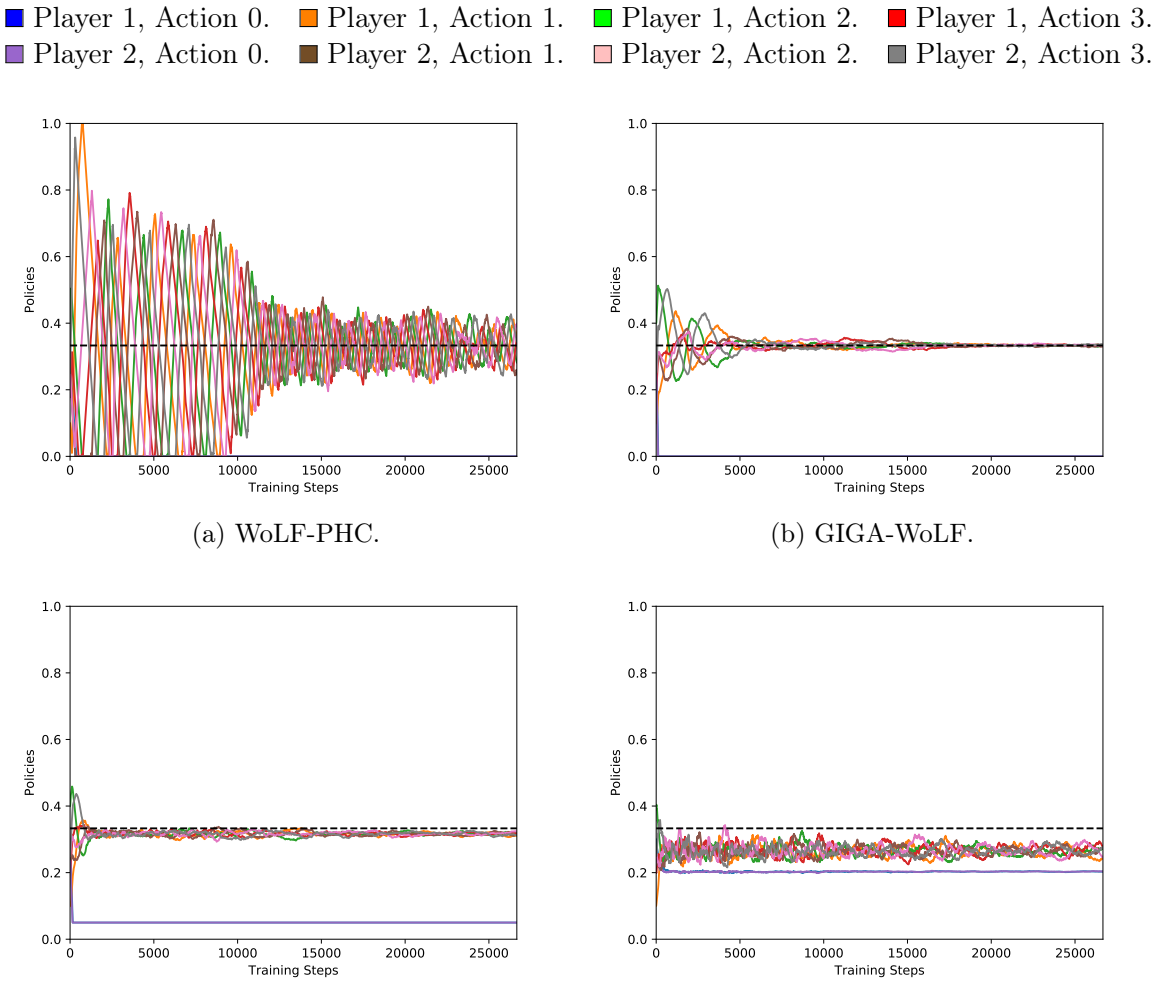
(a) WoLF-PHC.

(b) GIGA-WoLF.

(c) WPL.

(d) EMA-QL.

Figure 5.5: The evolution of the policies of 2 agents in self-play on the RPS game. Players use the tabular Wolf-PHC, GIGA-WoLF, WPL, and EMA-QL algorithms, and plots show the probability of choosing actions over epochs of 25 time-steps.



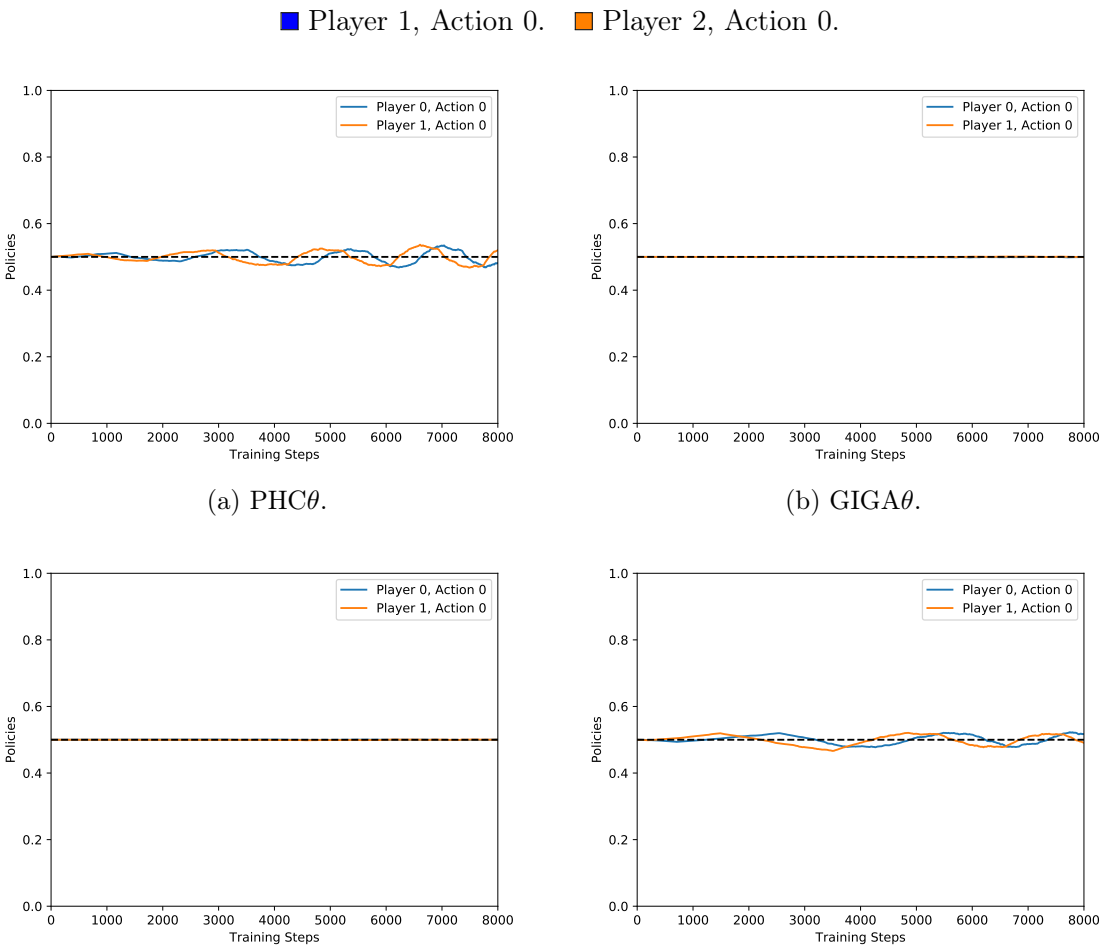
(a) WoLF-PHC.

(b) GIGA-WoLF.

(c) WPL.

(d) EMA-QL.

Figure 5.6: The evolution of the policies of 2 agents in self-play on the NRPS game. Players use the tabular Wolf-PHC, GIGA-WoLF, WPL, and EMA-QL algorithms, and plots show the probability of choosing actions over epochs of 25 time-steps.



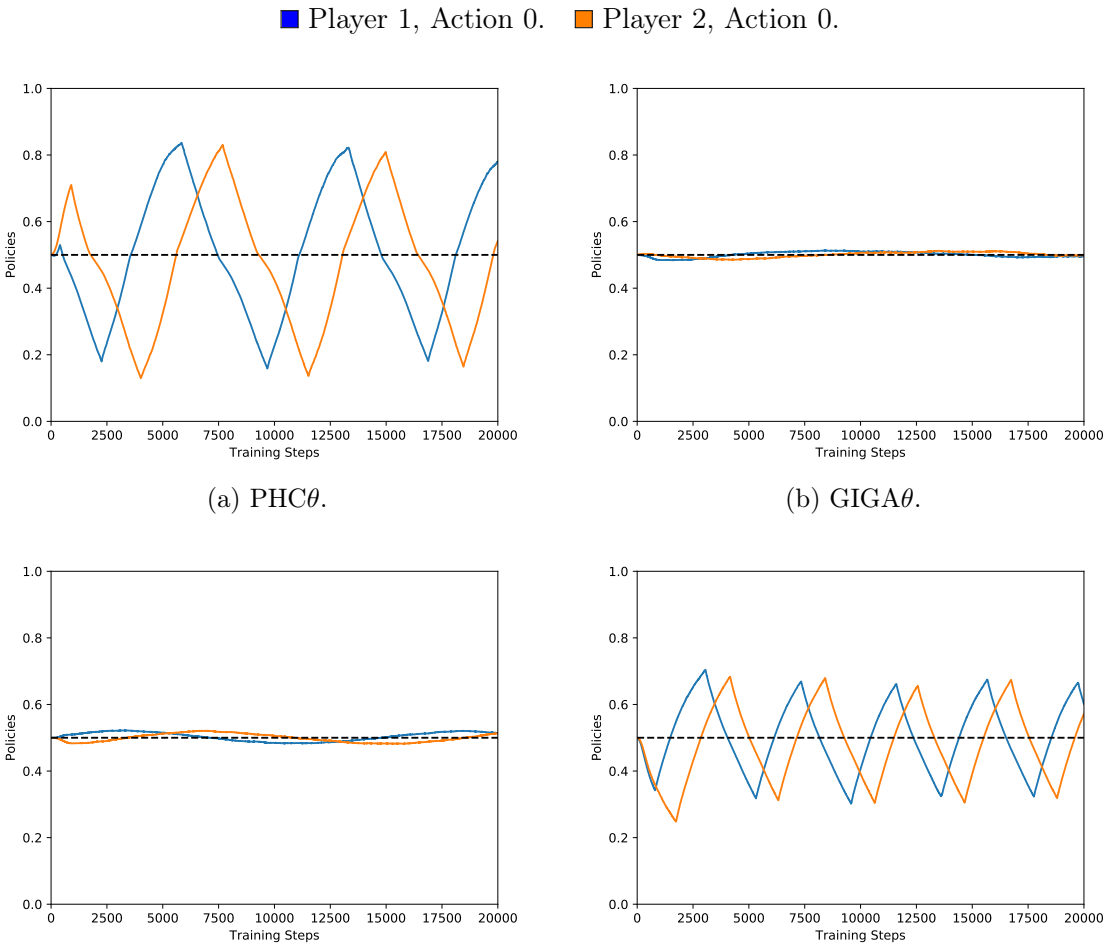
(a) PHCθ.

(b) GIGAθ.

(c) WPLθ.

(d) EMAθ.

Figure 5.7: The evolution of the policies of 2 agents in self-play on the Matching Pennies game. Players use the PHCθ, GIGAθ, WPLθ, and EMAθ algorithms, and plots show the probability of choosing the first action over epochs of 25 time-steps.



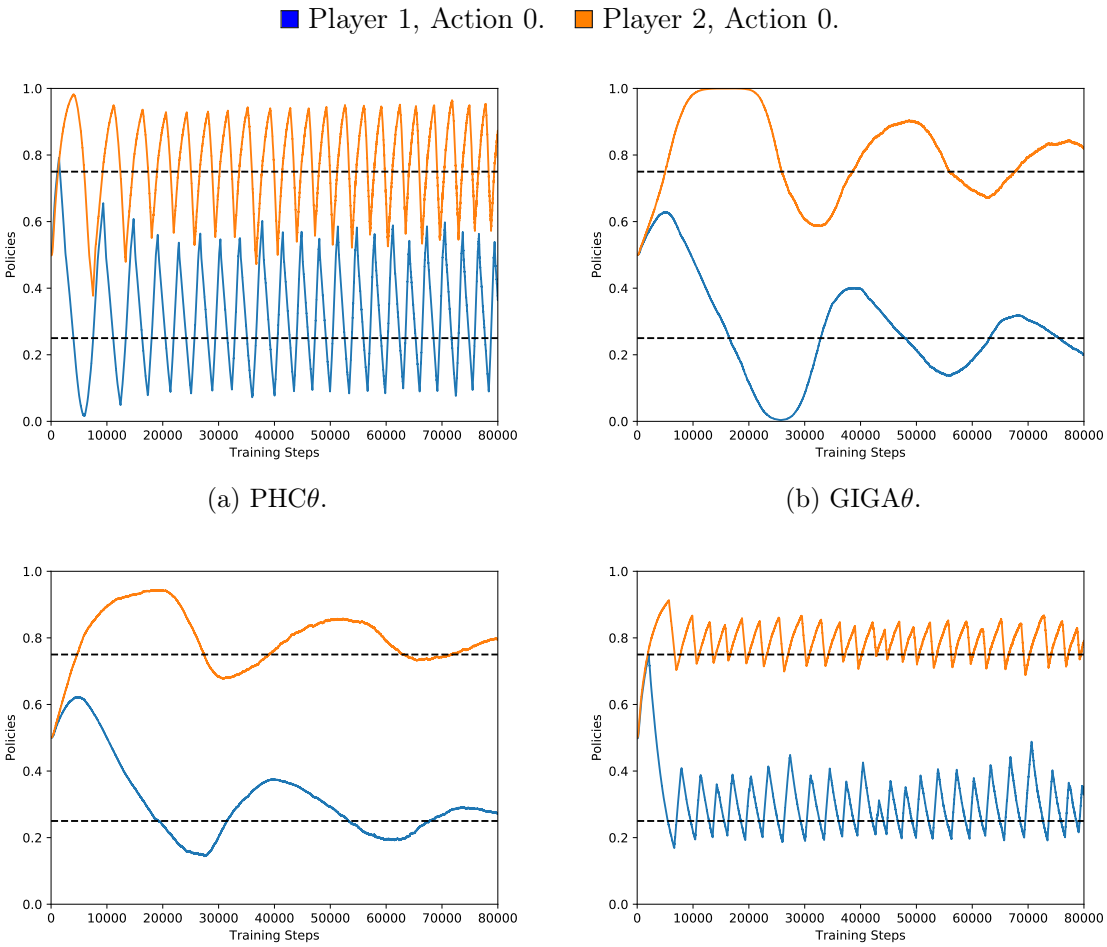
(a) PHCθ.

(b) GIGAθ.

(c) WPLθ.

(d) EMAθ.

Figure 5.8: The evolution of the policies of 2 agents in self-play on the Tricky game. Players use the PHCθ, GIGAθ, WPLθ, and EMAθ algorithms, and plots show the probability of choosing the first action over epochs of 25 time-steps.



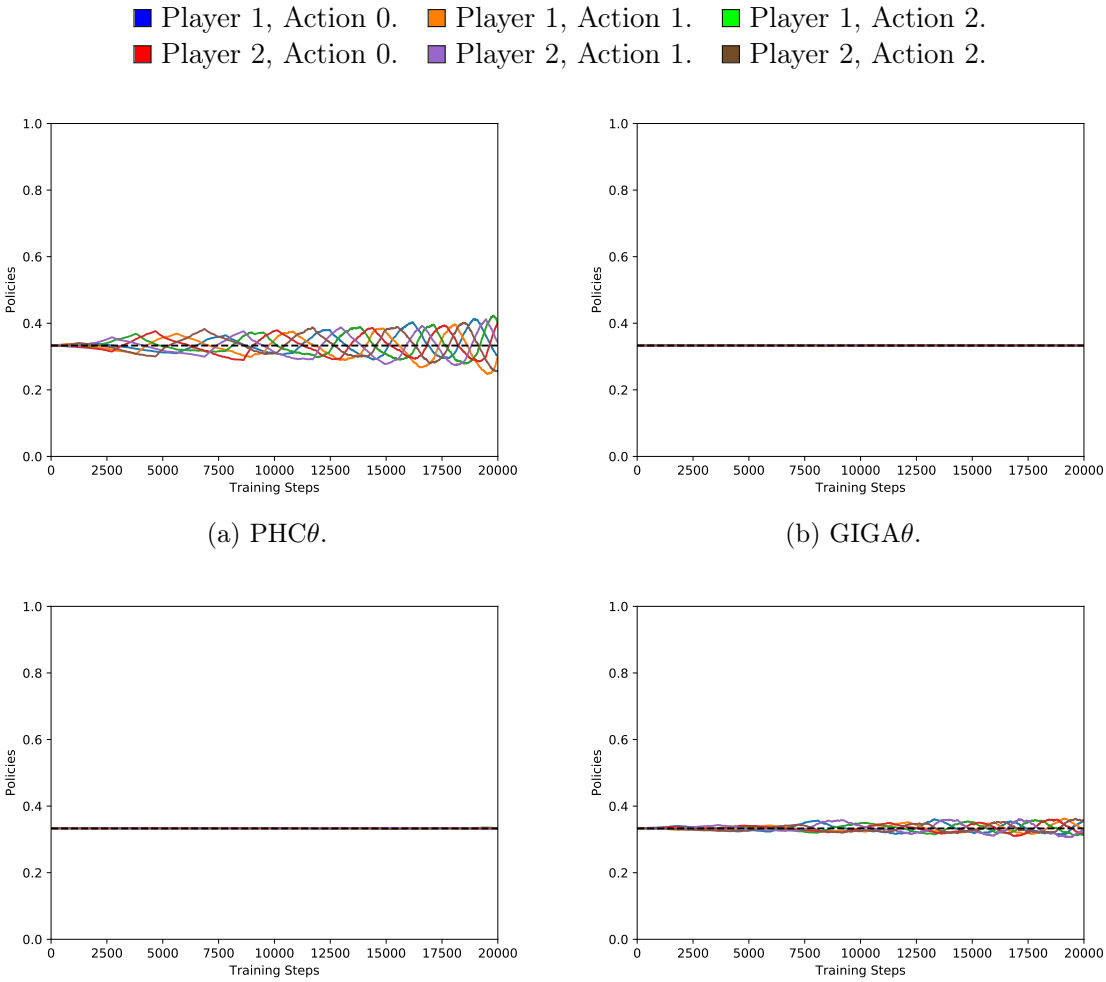
(a) PHCθ.

(b) GIGAθ.

(c) WPLθ.

(d) EMAθ.

Figure 5.9: The evolution of the policies of 2 agents in self-play on the Biased game. Players use the PHCθ, GIGAθ, WPLθ, and EMAθ algorithms, and plots show the probability of choosing the first action over epochs of 25 time-steps.



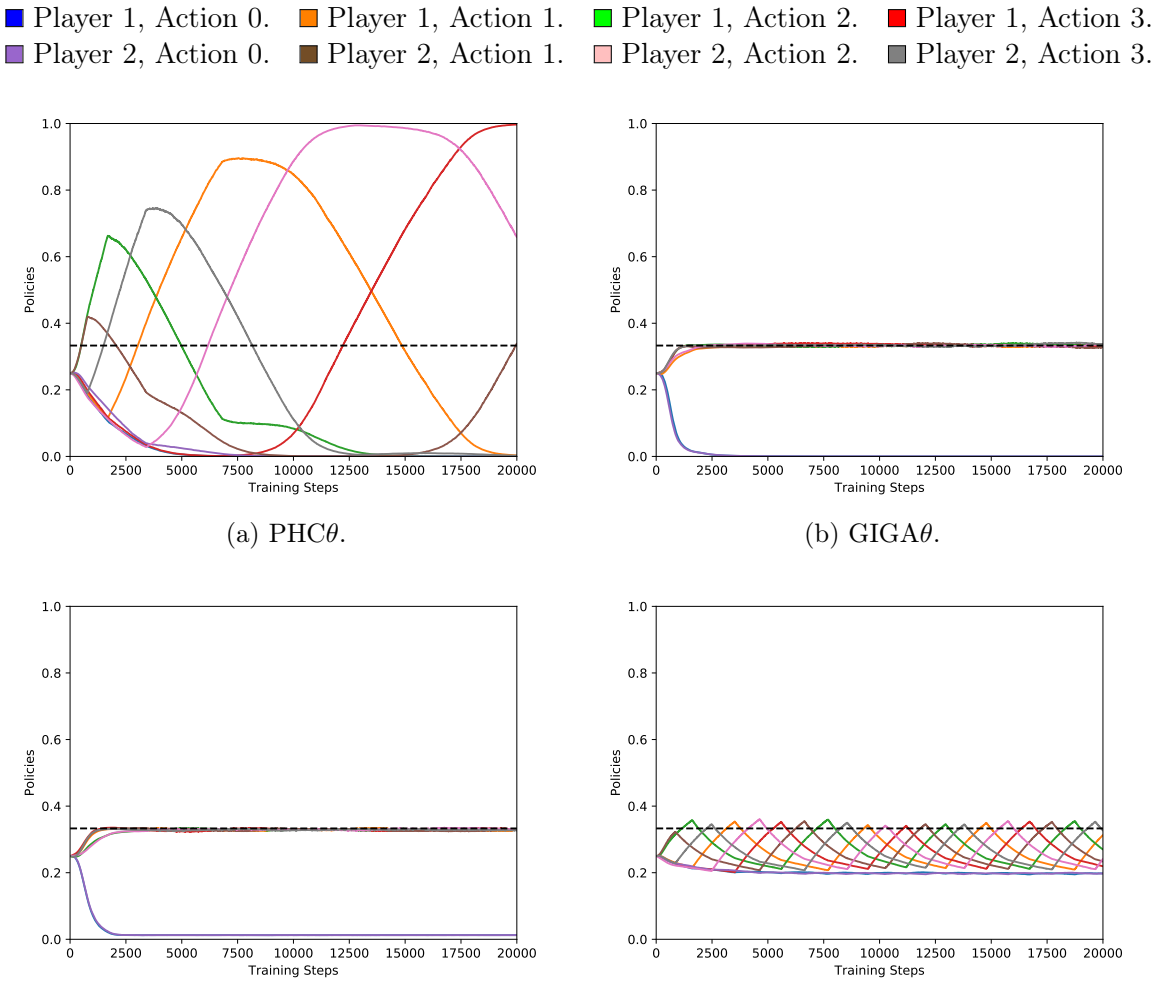
(a) PHCθ.

(b) GIGAθ.

(c) WPLθ.

(d) EMAθ.

Figure 5.10: The evolution of the policies of 2 agents in self-play on the RPS game. Players use the PHCθ, GIGAθ, WPLθ, and EMAθ algorithms, and plots show the probability of choosing each action over epochs of 25 time-steps.



(a) PHCθ.

(b) GIGAθ.

(c) WPLθ.

(d) EMAθ.

Figure 5.11: The evolution of the policies of 2 agents in self-play on the NRPS game. Players use the PHCθ, GIGAθ, WPLθ, and EMAθ algorithms, and plots show the probability of choosing each action over epochs of 25 time-steps.

in NRPS and is now also heavily oscillating in the Tricky game.

Tests were repeated with a less complex neural network architecture, using only a single hidden layer of nine nodes, and all other hyper-parameters unchanged. While most algorithms maintained their behaviors with this simpler architecture, the GIGAθ algorithm diverged in the NRPS game, as shown in Figure 5.12. A single state game does not require a complex non-linear function approximator for the value or policy functions, but GIGAθ does require a more complex network architecture than WPLθ in order to converge in this scenario.

Our analysis shows that WPLθ remains robust to different kinds of game theoretic environments and network architectures, and achieves convergence to NE policies in self-play. GIGAθ also shows rationality and convergence properties with a sufficiently complex network architecture. The use of learning batches can contribute to the increased instability of algorithms, since updates are no longer conducted every time-step, but instead at every mini-batch of time-steps, leading to a delayed response to opponent strategies. The non-linear approximation that the neural network itself performs on the value and policy functions can also contribute to this problem, by providing an inaccurate estimation of these functions.

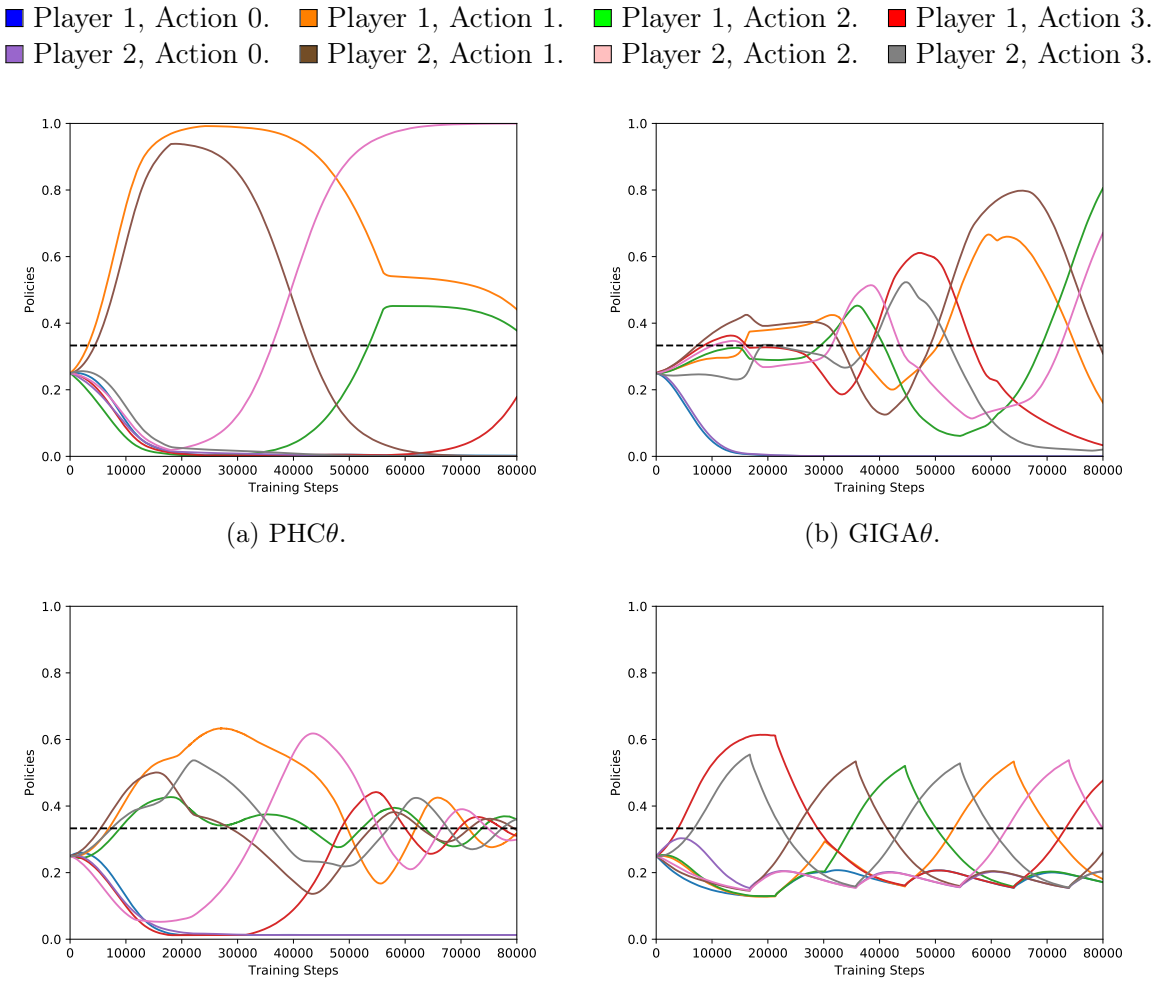
5.3.3 Multi-State Environments

Deep asynchronous algorithms have been shown to handle noisy, partially-observable or continuous state-space environments [131, 141]. Tests are now conducted on a partially-observable variant of MazeRPS, described in Section 3.4.1, where agent observations are noisy. Two agents are spawned and observe noisy continuous values for all cells except their opponent's. Agents must converge and play NRPS to end the game. This variant of MazeRPS features multiple properties that make it an adequate test bed. It features an infinite set of observations, which is unsupported by the tabular versions of the described algorithms. All game states have the same amount of actions, four for moving, and four for NRPS, which simplifies network architectures. NRPS has a positive average reward, this making the game's final state a desirable one. Finally, MazeRPS features two distinct phases, one with an optimal deterministic strategy where agents reach each other, and one where agents play the NRPS game with stochastic policies. Deep asynchronous mixed-policy algorithms should be able to achieve optimal policies in both phases of the game.

Similar hyper-parameters were used, but with two hidden layers of 150 nodes, a future reward discount factor $\gamma = 0.9$, and the ϵ-annealing exploration technique, following the scheme used originally in Mnih et al. [139], with each worker annealing the exploration rate from 1 to one of $0.5, 0.1, 0.01$ over 80% of episodes. Additional tests were also conducted with a higher learning rate $\eta = 10^{-4}$. The Asynchronous Advantage Actor-Critic (A3C) algorithm [139] was also tested, using a batch size $t_{\max} = 1$. It is an actor-critic algorithm that has been shown to outperform A1Q in single-agent environments. However, despite maintaining a probability distribution over possible actions for each state, and thus being able to converge to a mixed policy, it was not designed for NE convergence in multi-agent learning.

The mixed policies for the final game state in both tests are shown in Figures 5.13 and 5.14. PHCθ, EMAθ, and A3C fail to converge to the NE strategies in both tests. The higher learning rate also causes GIGAθ to catastrophically diverge. WPLθ successfully completes the task in both cases, and converges to the NE strategy.

Once more, WPLθ is shown to be robust to hyper-parameters changes, and is able to converge to the NE strategy with different learning rate magnitudes. GIGAθ is only able to converge with a smaller learning rate, but WPLθ still shows a smaller variance. An analysis of



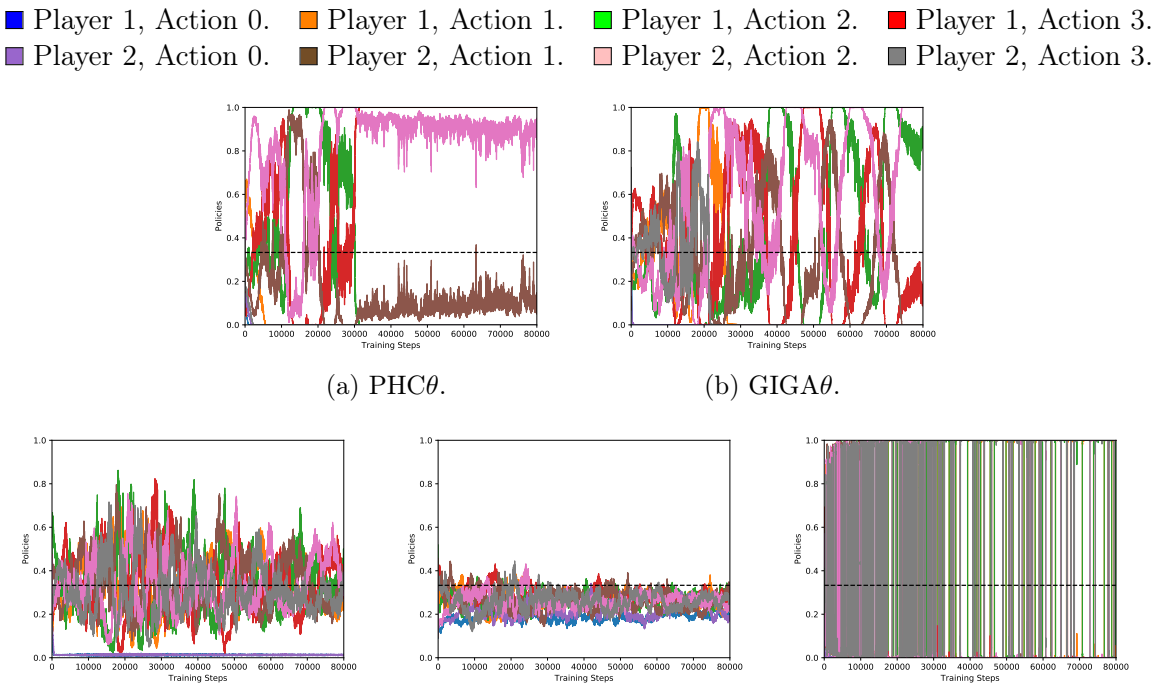
(a) PHCθ.

(b) GIGAθ.

(c) WPLθ.

(d) EMAθ.

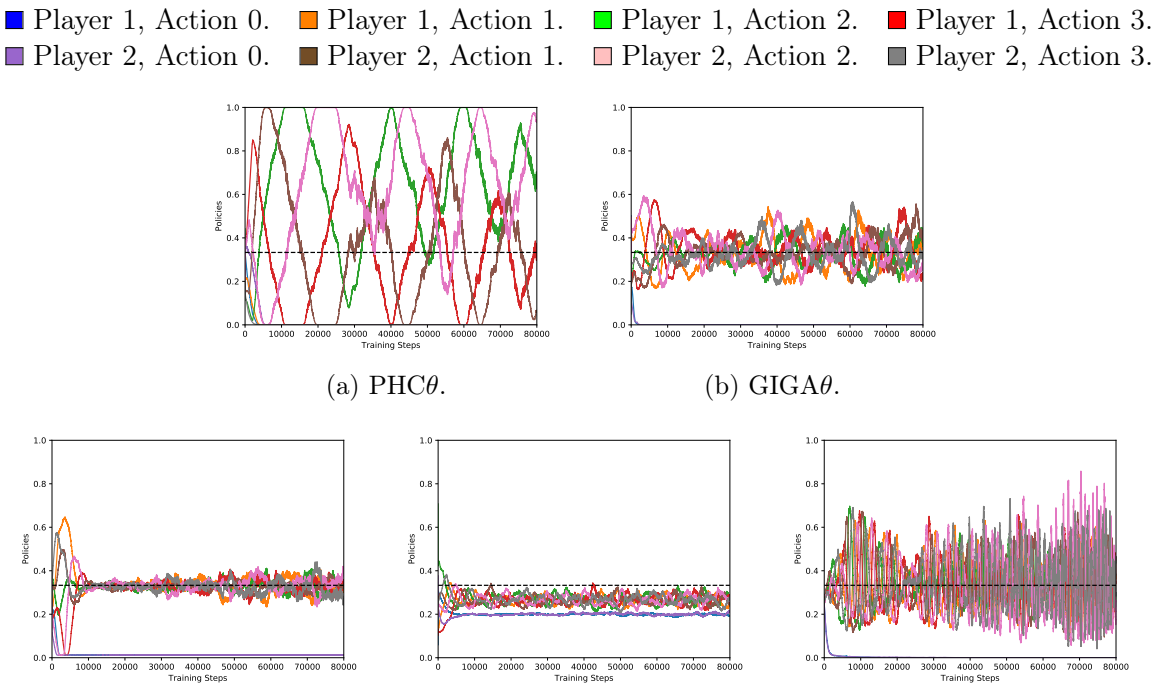
Figure 5.12: The evolution of the policies of 2 agents in self-play on the NRPS game. Players use the PHCθ, GIGAθ, WPLθ, and EMAθ algorithms, with a single hidden-layer of 9 nodes, and plots show the probability of choosing each action over epochs of 25 time-steps.



(a) PHCθ. (b) GIGAθ.

(c) WPLθ. (d) EMAθ. (e) A3C.

Figure 5.13: The evolution of the policies of 2 agents in self-play on the noisy MazeRPS game, using a learning rate $\eta = 10^{-4}$. Players use the PHCθ, GIGAθ, WPLθ, EMAθ, and A3C algorithms, and plots show the probability of choosing each action over elapsed episodes.



(a) PHCθ. (b) GIGAθ.

(c) WPLθ. (d) EMAθ. (e) A3C.

Figure 5.14: The evolution of the policies of 2 agents in self-play on the noisy MazeRPS game, using a learning rate $\eta = 10^{-5}$. Players use the PHCθ, GIGAθ, WPLθ, EMAθ, and A3C algorithms, and plots show the probability of choosing each action over elapsed episodes.

the average game length for each algorithm is shown in Figure 5.15. The average game length is determined by deterministic policies in the first phase of the environment, where agents must reach each other. No algorithm was able to converge to the performance of hard-coded policies, with EMAθ showing the worst results. A3C with a learning rate $\eta = 10^{-4}$ matched that performance within the same amount of episodes.

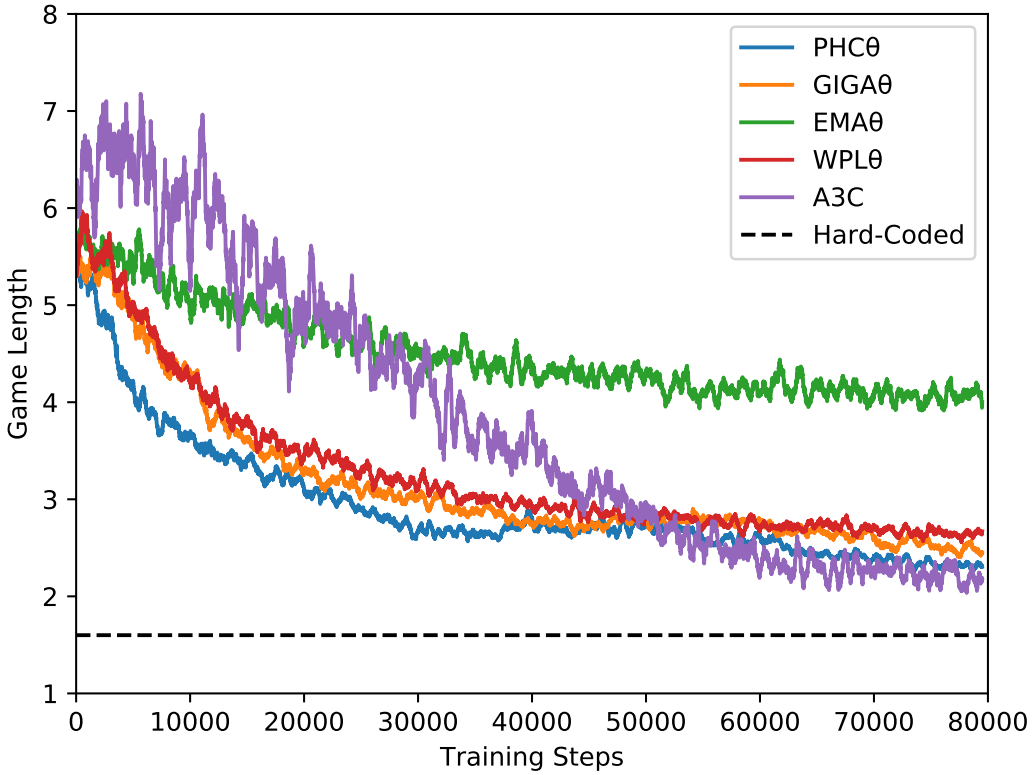
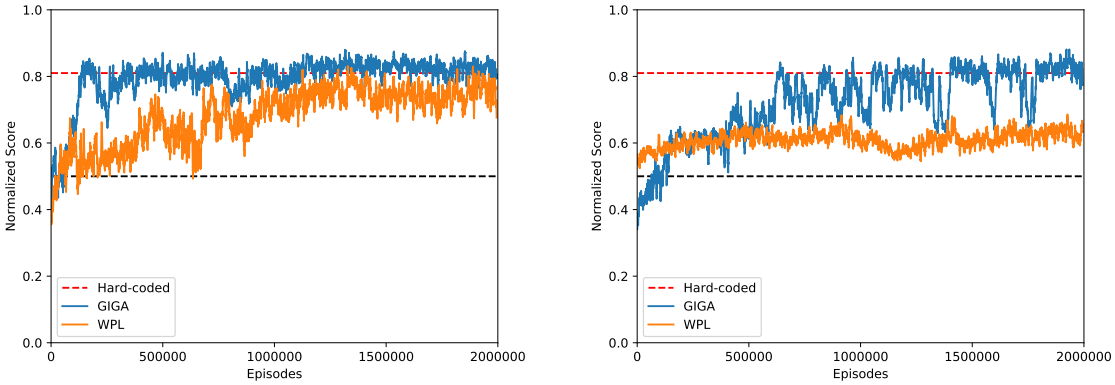


Figure 5.15: The evolution of the policies of 2 agents in self-play on the noisy MazeRPS game, using a learning rate $\eta = 10^{-5}$. Players use the PHCθ, GIGAθ, WPLθ, EMAθ, and A3C algorithms, and plots show the average game length over elapsed episodes.

Despite being biased against deterministic strategies and its lack of formal proof of convergence, the GIGAθ and WPLθ extensions can converge to both pure and mixed NE strategies in MazeRPS. WPLθ is shown to be more robust, requires less hyper-parameters (only a single policy update rate δ), and is computationally cheaper (a single policy network is required) than GIGAθ. However, its bias against deterministic strategies may cause it to converge asymptotically in environments where the NE for the majority of states are pure. This is the case of SPE, described in Section 3.5.4, where agents try to win a Pokémon battle. While some states can exhibit stochastic NE, the majority have deterministic equilibrium decisions (such as using the highest damaging move when a switch is no longer possible).

The GIGAθ and WPLθ algorithms are now evaluated in the SPE environment. Tests were conducted in a complete version of SPE and in a simpler version (with only seven different

Pokémon types), and used 12 concurrent workers, a neural network with three hidden layers of 150 nodes with ELU activations [91], and an ϵ-annealing technique over 65% of episodes [139], with other hyper-parameters unchanged. Both algorithms trained in self-play for two million episodes, and were tested in a fixed scenario, where a trained agent in a disadvantageous position played against a random enemy. The learning results are shown in Figure 5.16, and compared against the best hard-coded baseline we could manually create. Both GIGAθ and WPLθ achieve good results in the 7-type SPE, but GIGAθ converges faster and is able to match the performance of the hard-coded baseline. In the complete version, WPLθ is no longer able to learn adequate policies. On the other hand, GIGAθ has a slower evolution with higher variance, likely due to the increased complexity of the environment, but still matches the performance of the hard-coded solution.



(a) Only 7 Pokémon types.

(b) All 18 Pokémon types.

Figure 5.16: Results of GIGAθ and WPLθ for the SPE, in the unfair test scenario. The plots represent the normalized score of each algorithm, trained in self-play but tested against a random agent, over training episodes.

The fixed test scenario puts the trainer at an initial disadvantage, initially favoring the opponent with a Fire/Normal roster, while the trainer has a Grass/Water roster. Because Fire is effective against Grass, and Water against Fire, the trainer's first move should be to switch Pokémon, favoring long-term rewards. After the switch, the Water Pokémon has both Water and Fighting attacks (effective against Fire and Normal Pokémon, respectively), and should pick those moves with high priority. This was the strategy used by the hard-coded policy, and the best we found during experiments with the environment. An example is shown in Figure 5.17, demonstrating how a GIGAθ trainer behaves in the unfair scenario. The trainer prioritizes strategic choices and exploits type advantages, by following the policy described previously. Without any previous domain knowledge, it learned how Fire is vulnerable to Water, Water to Grass, and Normal to Fighting, and makes strategic decisions while using effective moves when possible.

5.4 Conclusion

This chapter described the extension of WoLF-PHC, GIGA-WoLF, WPL, and EMA-QL to the deep learning paradigm. These mixed-policy tabular algorithms require only local agent

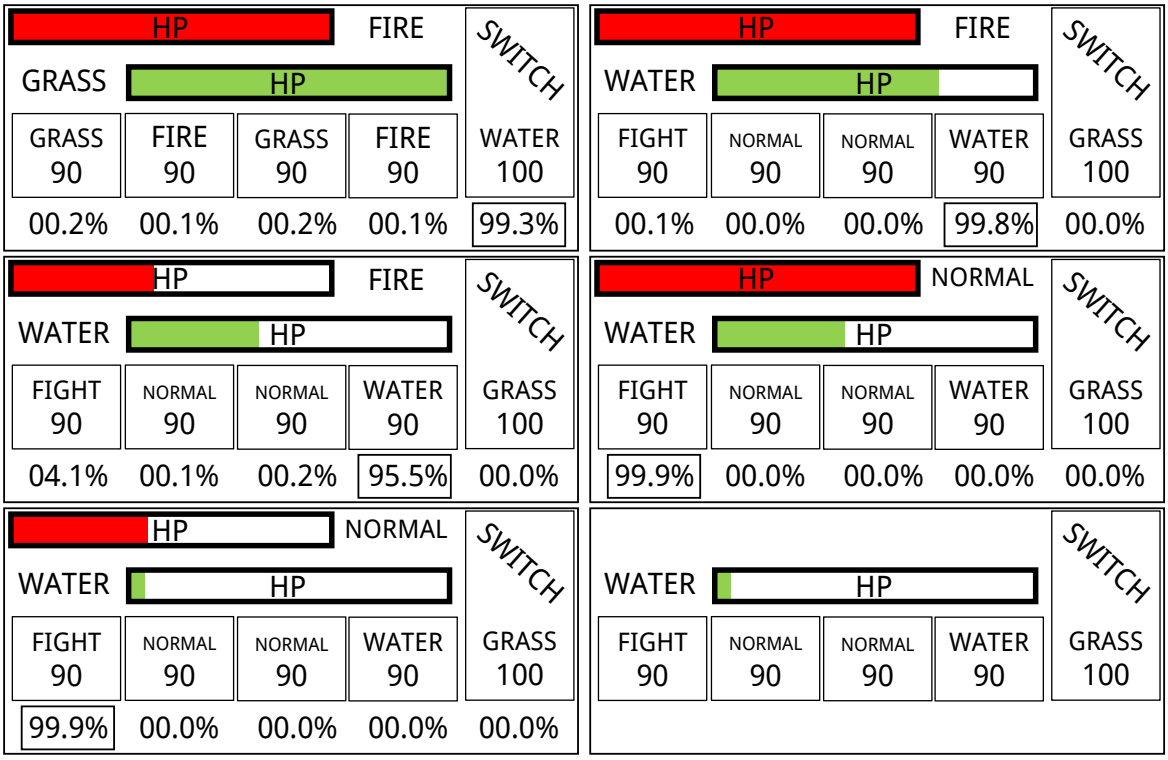


Figure 5.17: A GIGAθ agent's policy in the fixed test scenario. The policy of the agent for each state is shown below, with the chosen action highlighted. The opponent's Pokémon use attacks of their own type every turn, but the initial switch gives the local agent an advantage, and it wins the battle with no casualties.

information to converge to NE policies, and a fair analysis of their original implementations is conducted. Results show the GIGA-WoLF and WPL outperform others in a set of single-state games, with GIGA-WoLF achieving the smallest variance.

Their asynchronous deep learning extensions no longer maintain the same behaviors. PHCθ and EMAθ both diverge in multiple games. GIGAθ requires a higher fine-tuning of hyper-parameters, while WPLθ is shown to be more robust. When tested on multi-state partially-observable games, GIGAθ required smaller learning rates and its variance was larger than WPLθ's in MazeRPS, but it was able to outperform WPLθ in SPE. A3C, a single-agent algorithm capable of mixed policies, oscillated around the NE strategy without converging. GIGAθ and WPLθ outperformed other candidates overall, with WPLθ seemingly more robust to hyper-parameters, and GIGAθ achieving better results in environments where the majority of NE are deterministic.

WPLθ under-performs in such environments because WPL's policy update equations are biased against deterministic strategies. The policy update rate approaches zero in such scenarios, and therefore both WPL and WPLθ's convergence speed is greatly hindered in games where the majority of states' optimal strategy is deterministic, which is highly undesirable.

Chapter 6

Adjusted Bounded Weighted Policy Learner

Weighted Policy Learner (WPL) [54] is an algorithm that keeps track of Q-values and maintains a probability distribution over possible actions. It has a variable learning rate, and allows agents to move towards the equilibrium strategy faster than moving away from it. However, its policy update rate is based on the agents' policies, which approximate zero when converging to a deterministic strategy. While the algorithm still converges to pure NE policies in practical cases, it converges asymptotically. This causes its learning speed in games with multiple states whose optimal strategies are pure to decrease substantially, a highly undesirable effect.

This chapter describes an extension to the WPL algorithm, with a new update rule that will allow the algorithm to converge faster to pure policies, where some actions are dominated by others, while still maintaining its original behavior when converging to stochastic policies. Two different hand-tuned approaches are analyzed and compared, and an automatically adjusted approach is then derived from those. The proposal is compared with the original algorithm in a wide set of game-theoretic environments, and against other state-of-the-art algorithms. These findings were published in the RoboCup18 symposium [58]. The algorithm's source-code and tests were published at `https://github.com/david-simoes-93/ABWPL`.

6.1 Problem Statement

Despite not having a formal proof of convergence due to the non-linear nature of WPL's dynamics, Abdallah et al. [54] perform a numerical analysis of WPL in a 2-player 2-action single-state game, where its equations can be simplified by defining a player's policy $\pi(s, a)$ by a single probability. Player 1 uses the strategy $\pi_p = (p, 1-p)$, while Player 2 uses the strategy $\pi_q = (q, 1-q)$, and the equilibrium strategy of the game is defined by $\pi^* = (p^*, q^*)$. The value functions $V_r(p, q)$ and $V_c(p, q)$, which WPL approximates with Q-values, are then

defined by the rewards of the game as

$$\begin{aligned} V_r(p,q) &= r_{11}pq + r_{12}p(1-q) + r_{21}(1-p)q + r_{22}(1-p)(1-q) = \\ &= ((r_{11} - r_{12} - r_{21} + r_{22})q + r_{12} - r_{22})p, \\ V_c(p,q) &= c_{11}pq + c_{12}p(1-q) + c_{21}(1-p)q + c_{22}(1-p)(1-q) = \\ &= ((c_{11} - c_{12} - c_{21} + c_{22})p + c_{12} - c_{22}). \end{aligned} \tag{6.1}$$

The definitions of $V_r(p,q)$ and $V_c(p,q)$ can be simplified by using the equalities

$$\begin{aligned} u_{r1} &= r_{11} - r_{12} - r_{21} + r_{22}, \\ u_{r2} &= r_{12} - r_{22}, \\ u_{c1} &= c_{11} - c_{12} - c_{21} + c_{22}, \\ u_{c2} &= c_{21} - c_{22}, \end{aligned} \tag{6.2}$$

thus obtaining

$$\begin{aligned} V_r(p,q) &= (u_{r1}q + u_{r2})p, \\ V_c(p,q) &= (u_{c1}p + u_{c2})q. \end{aligned} \tag{6.3}$$

For clarity, we refer readers to Section 2.5.1, which describes the policy update steps of WPL. We can now calculate $\frac{\partial V_t(s)}{\partial \pi_t(s,a)}$, seen in equation 2.9, as $\frac{\partial V_r(p,q)}{\partial p}$ and $\frac{\partial V_c(p,q)}{\partial q}$, by

$$\begin{aligned} \frac{\partial V_r(p,q)}{\partial p} &= u_{r1}q + u_{r2}, \\ \frac{\partial V_c(p,q)}{\partial q} &= u_{c1}p + u_{c2}. \end{aligned} \tag{6.4}$$

The probabilities p and q of agents (and thus, their strategies) are, from equations 2.9, 6.3, and 6.4, now given by

$$\begin{aligned} p_{t+1} &= p_t + \eta(u_{r1}q_t + u_{r2}) \begin{cases} p_t \text{ if } u_{r1}q_t + u_{r2} < 0 \\ 1 - p_t \text{ otherwise} \end{cases}, \\ q_{t+1} &= q_t + \eta(u_{c1}p_t + u_{c2}) \begin{cases} q_t \text{ if } u_{c1}p_t + u_{c2} < 0 \\ 1 - q_t \text{ otherwise} \end{cases}. \end{aligned} \tag{6.5}$$

The rate $\frac{dp}{dt}$ and $\frac{dq}{dt}$ at which policies p and q evolve, respectively, can be found by rewriting their equations with respect to t. While $p_t = p(t)$ and $q_t = q(t)$, their value on the next time-step is given by $p_{t+1} = p(t+\eta)$ and $q_{t+1} = q(t+\eta)$. With this, equation 6.5 can be rewritten

to represent a derivative in which $\eta \to 0$, thus allowing the calculation of the derivatives of p and q with respect to t.

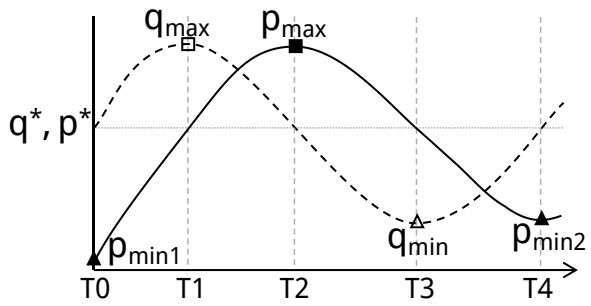
$$
\begin{aligned}
p(t+\eta) &= p(t) + \eta(u_{r1}q_t + u_{r2}) \begin{cases} p_t \text{ if } u_{r1}q_t + u_{r2} < 0 \\ 1 - p_t \text{ otherwise} \end{cases} \quad \Leftrightarrow \\
\frac{p(t+\eta) - p(t)}{\eta} &= (u_{r1}q_t + u_{r2}) \begin{cases} p_t \text{ if } u_{r1}q_t + u_{r2} < 0 \\ 1 - p_t \text{ otherwise} \end{cases} \quad \Leftrightarrow \\
\Leftrightarrow \frac{dp}{dt} &= (u_{r1}q_t + u_{r2}) \begin{cases} p_t \text{ if } u_{r1}q_t + u_{r2} < 0 \\ 1 - p_t \text{ otherwise} \end{cases} , \\
q(t+\eta) &= q(t) + \eta(u_{c1}p_t + u_{c2}) \begin{cases} q_t \text{ if } u_{c1}p_t + u_{c2} < 0 \\ 1 - q_t \text{ otherwise} \end{cases} \quad \Leftrightarrow \\
\frac{q(t+\eta) - q(t)}{\eta} &= (u_{c1}p_t + u_{c2}) \begin{cases} q_t \text{ if } u_{c1}p_t + u_{c2} < 0 \\ 1 - q_t \text{ otherwise} \end{cases} \quad \Leftrightarrow \\
\Leftrightarrow \frac{dq}{dt} &= (u_{c1}p_t + u_{c2}) \begin{cases} q_t \text{ if } u_{c1}p_t + u_{c2} < 0 \\ 1 - q_t \text{ otherwise} \end{cases} .
\end{aligned}
\tag{6.6}
$$

Given that $\frac{\frac{dp}{dt}}{\frac{dq}{dt}} = \frac{dp}{dq}$ is a differentiable separable equation, it can be solved by separation and integration of both sides of the equation.

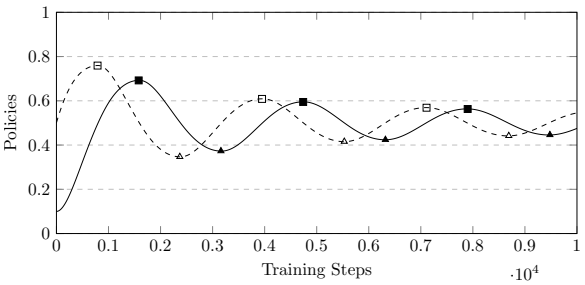
$$
\begin{aligned}
&\frac{(u_{r1}q_t + u_{r2}) \begin{cases} p_t \text{ if } u_{r1}q_t + u_{r2} < 0 \\ 1 - p_t \text{ otherwise} \end{cases}}{(u_{c1}p_t + u_{c2}) \begin{cases} q_t \text{ if } u_{c1}p_t + u_{c2} < 0 \\ 1 - q_t \text{ otherwise} \end{cases}} = \frac{dp}{dq} \\
&\int \frac{(u_{c1}p_t + u_{c2})}{\begin{cases} p_t \text{ if } u_{r1}q_t + u_{r2} < 0 \\ 1 - p_t \text{ otherwise} \end{cases}} dp = \int \frac{(u_{r1}q_t + u_{r2})}{\begin{cases} q_t \text{ if } u_{c1}p_t + u_{c2} < 0 \\ 1 - q_t \text{ otherwise} \end{cases}} dq + k.
\end{aligned}
\tag{6.7}
$$

Abdallah et al. [54] attempt to solve these equations in the case shown in Figure 6.1(a) for a Matching Pennies game. Here, Player 2 starts on the equilibrium strategy and, from T0 to T4, the strategy of Player 1 gets closer to his equilibrium strategy while Player 2 ends on his own equilibrium strategy. If $p_{min1} < p_{min2}$, or in other words, if the row player gets closer to his equilibrium strategy from $T0 \to T4$, then by induction, the players converge to an equilibrium strategy in the following iterations. However, this cannot be solved in closed form, since there are five unknowns (p_{min1}, p_{min2}, p_{max}, q_{min}, and q_{max}), and only four equations.

Instead, the authors numerically show that policies converge in the example in Figure 6.1(b), where $p_{min1} = 0.1$. The primitives for the intervals $T_0 \to T_1$, $T_1 \to T_2$, $T_2 \to T_3$, and $T_3 \to T_4$, can be calculated, since $u_{r1}q + u_{r2} < 0 \Leftrightarrow q < q^*$ and $u_{c1}p + u_{c2} < 0 \Leftrightarrow p > p^*$ for this game. The plots use the actual algorithm and the marks are predicted by the theoretical model, with an adjusted scale to match the practical values. In practice, WPL uses the Q-function $Q_t(s, a)$ to approximate the value function $V_t(s)$.



(a) The general scenario used in WPL's analysis, divided into four time intervals. The probability p starts at some minimum value p_{min1} and oscillates from p_{max} to p_{min2}. The probability q starts at the equilibrium q^* and oscillates from q_{max} to q_{min}, before returning to q^*.



(b) The evolution of the policies of the row player (solid) and the column player (dashed) in a Matching Pennies game, using the original WPL algorithm.

Figure 6.1: The general scenario used in WPL's analysis, and a theoretical and practical example in a Matching Pennies game. The squared marks represent the p_{max} and q_{max} values, while triangular marks represent p_{min} and q_{min} values. Simultaneously, solid marks represent p_{max} and p_{min} values and clear marks represent q_{max} and q_{min} values.

Because agents move towards the equilibrium strategy faster than moving away from it (which ensures WPL's *convergence* property), problems are found with pure equilibrium strategies, where the algorithm only converges in the limit. If the equilibrium strategy is pure, WPL will converge to this strategy asymptotically. If an environment contains hundreds of states with deterministic optimal strategies, all of which WPL will be converging to asymptotically, WPL's performance is severely hindered. This happens since the policy update rate approaches zero in these cases, due to the use of $\pi_t(s, a)$ to adjust the rate. In practice, the algorithm does converge in simple games due to randomness, numerical limits, and a non-zero exploration chance used, but the process fails to converge in complex environments.

This has a highly undesirable effect for either pure-strategy single state games, or for any complex environment where sufficient states contain pure strategies which are, in fact, optimal.

6.2 Proposal

This chapter describes a modification to WPL's update rule, such that the update factor no longer approaches 0 when converging to pure strategies, thus removing its asymptotic convergence properties. By binding the $\pi_t(a)$ factor used to calculate the $\Delta(s_t)$ increment vector (as shown in equation 2.9), such that the original interval $[0, 1]$ no longer approaches 0, WPL will no longer converge asymptotically.

Intuitively, we can do so by adjusting the $\pi_t(a)$ factor away from 0, and we consider two options. The first, which we call *Bounded WPL*, keeps the interval's mean at 0.5, and both lower and higher bounds are changed. The second, *High WPL*, is based on changing the interval's lower bound and mean, and maintaining its upper bound at 1. The intervals are adjusted in such a way that the original convergence properties of WPL are maintained.

Both variants can be numerically compared with a similar analysis to WPL's. We consider the example where the adjusted interval has half its original size. For *Bounded WPL*, $\pi_t(s,a) \in [0.25, 0.75]$, while for *High WPL*, $\pi_t(s,a) \in [0.5, 1.0]$.

6.2.1 Bounded WPL

From equation 2.9, the new policy update rules for Bounded WPL is

$$\forall a \in A \ \Delta_t(s,a) = \eta_t^{\pi} \frac{\partial V_t(s)}{\partial \pi_t(s,a)} \times \begin{cases} \frac{\pi_t(s,a)}{2} + 0.25 \text{ if } \frac{\partial V_t(s)}{\partial \pi_t(s,a)} < 0 \\ 0.75 - \frac{\pi_t(s,a)}{2} \text{ otherwise} \end{cases} . \tag{6.8}$$

These adjustments do not invalidate the convergence properties of the algorithm, since we have kept the fundamental property of WPL where the probability of choosing an action increases or decreases by a rate that decreases as the probability approaches the boundary of the probability simplex. In other words, agents move towards their Nash Equilibrium strategy (away from the simplex boundary) faster than they move away from it.

The differential equation of Bounded WPL can then be written as

$$\int \frac{(u_{c1}p_t + u_{c2})}{\begin{cases} \frac{p_t}{2} + 0.25 \text{ if } u_{r1}q_t + u_{r2} < 0 \\ 0.75 - \frac{p_t}{2} \text{ otherwise} \end{cases}} dp = \int \frac{(u_{r1}q_t + u_{r2})}{\begin{cases} \frac{q_t}{2} + 0.25 \text{ if } u_{c1}p_t + u_{c2} < 0 \\ 0.75 - \frac{q_t}{2} \text{ otherwise} \end{cases}} dq + k. \tag{6.9}$$

This equation can be solved for the same case shown in Figure 6.1, by calculating the definite integral for each interval, from T_0 to T_4.

$$\begin{aligned} \int_{p_{min1}}^{p^*} \frac{(u_{c1}p_t + u_{c2})}{0.75 - \frac{p_t}{2}} dp &= \int_{q^*}^{q_{max}} \frac{(u_{r1}q_t + u_{r2})}{0.75 - \frac{q_t}{2}} dq \\ -(2u_{c2} + 3u_{c1}) \ln \frac{|2p^* - 3|}{|2p_{min1} - 3|} + 2u_{c1}(p_{min1} - p^*) &= \\ = -(2u_{r2} + 3u_{r1}) \ln \frac{|2q_{max} - 3|}{|2q^* - 3|} &+ 2u_{r1}(q^* - q_{max}) \end{aligned} \tag{6.10}$$

$$\begin{aligned} \int_{p^*}^{p_{max}} \frac{(u_{c1}p_t + u_{c2})}{0.75 - \frac{p_t}{2}} dp &= \int_{q_{max}}^{q^*} \frac{(u_{r1}q_t + u_{r2})}{\frac{q_t}{2} + 0.25} dq \\ -(2u_{c2} + 3u_{c1}) \ln \frac{|2p_{max} - 3|}{|2p^* - 3|} + 2u_{c1}(p^* - p_{max}) &\\ = (2u_{r2} - u_{r1}) \ln \frac{2q^* + 1}{2q_{max} + 1} &+ 2u_{r1}(q^* - q_{max}) \end{aligned} \tag{6.11}$$

$$\begin{aligned} \int_{p_{max}}^{p^*} \frac{(u_{c1}p_t + u_{c2})}{\frac{p_t}{2} + 0.25} dp &= \int_{q^*}^{q_{min}} \frac{(u_{r1}q_t + u_{r2})}{\frac{q_t}{2} + 0.25} dq \\ (2u_{c2} - u_{c1}) \ln \frac{2p^* + 1}{2p_{max} + 1} + 2u_{c1}(p^* - p_{max}) &\\ = (2u_{r2} - u_{r1}) \ln \frac{2q_{min} + 1}{2q^* + 1} &+ 2u_{r1}(q_{min} - q^*) \end{aligned} \tag{6.12}$$

$$\int_{p^*}^{p_{min2}} \frac{(u_{c1}p_t + u_{c2})}{\frac{p_t}{2} + 0.25} dp = \int_{q_{min}}^{q^*} \frac{(u_{r1}q_t + u_{r2})}{0.75 - \frac{q_t}{2}} dq$$

$$(2u_{c2} - u_{c1}) \ln \frac{2p_{min2} + 1}{2p^* + 1} + 2u_{c1}(p_{min2} - p^*) \tag{6.13}$$

$$= -(2u_{r2} + 3u_{r1}) \ln \frac{|2q^* - 3|}{|2q_{min} - 3|} + 2u_{r1}(q_{min} - q^*)$$

6.2.2 High WPL

Analogously, for High WPL, the new policy update rule becomes

$$\forall a \in A \ \Delta_t(s,a) = \eta_t^{\pi} \frac{\partial V_t(s)}{\partial \pi_t(s,a)} \times \begin{cases} \frac{\pi_t(s,a)}{2} + 0.5 \text{ if } \frac{\partial V_t(s)}{\partial \pi_t(s,a)} < 0 \\ 1 - \frac{\pi_t(s,a)}{2} \text{ otherwise} \end{cases} . \tag{6.14}$$

The differential equation of High WPL can then be written as

$$\int \frac{(u_{c1}p_t + u_{c2})}{\begin{cases} \frac{p_t}{2} + 0.5 \text{ if } u_{r1}q_t + u_{r2} < 0 \\ 1 - \frac{p_t}{2} \text{ otherwise} \end{cases}} dp = \int \frac{(u_{r1}q_t + u_{r2})}{\begin{cases} \frac{q_t}{2} + 0.5 \text{ if } u_{c1}p_t + u_{c2} < 0 \\ 1 - \frac{q_t}{2} \text{ otherwise} \end{cases}} dq + k. \tag{6.15}$$

And lastly this equation can be solved for the same case, from T_0 to T_4.

$$\int_{p_{min1}}^{p^*} \frac{(u_{c1}p_t + u_{c2})}{1 - \frac{p_t}{2}} dp = \int_{q^*}^{q_{max}} \frac{(u_{r1}q_t + u_{r2})}{1 - \frac{q_t}{2}} dq$$

$$2(u_{c1}(p_{min1} - p^*) + (u_{c2} + 2u_{c1}) \ln \frac{|p_{min1} - 2|}{|p^* - 2|}) = \tag{6.16}$$

$$= 2(u_{r1}(q^* - q_{max}) + (u_{r2} + 2u_{r1}) \ln \frac{|q^* - 2|}{|q_{max} - 2|})$$

$$\int_{p^*}^{p_{max}} \frac{(u_{c1}p_t + u_{c2})}{1 - \frac{p_t}{2}} dp = \int_{q_{max}}^{q^*} \frac{(u_{r1}q_t + u_{r2})}{\frac{q_t}{2} + 0.5} dq$$

$$2(u_{c1}(p^* - p_{max}) + (u_{c2} + 2u_{c1}) \ln \frac{|p^* - 2|}{|p_{max} - 2|}) \tag{6.17}$$

$$= 2((u_{r2} - u_{r1}) \ln \frac{q^* + 1}{q_{max} + 1} + u_{r1}(q^* - q_{max}))$$

$$\int_{p_{max}}^{p^*} \frac{(u_{c1}p_t + u_{c2})}{\frac{p_t}{2} + 0.5} dp = \int_{q^*}^{q_{min}} \frac{(u_{r1}q_t + u_{r2})}{\frac{q_t}{2} + 0.5} dq$$

$$2((u_{c2} - u_{c1}) \ln \frac{p^* + 1}{p_{max} + 1} + u_{c1}(p^* - p_{max})) \tag{6.18}$$

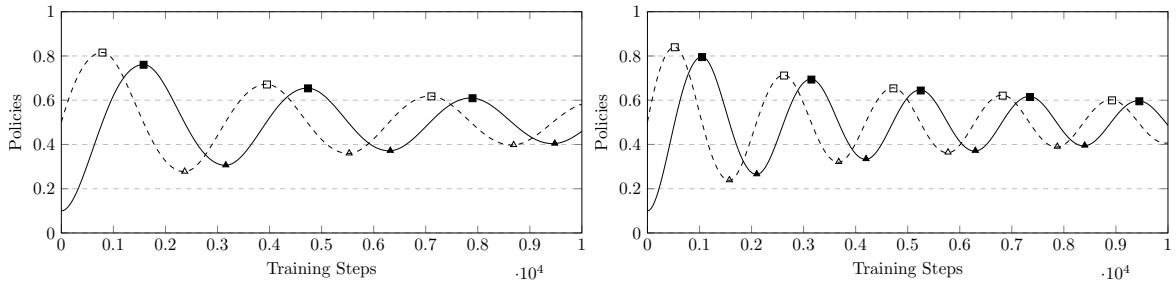
$$= 2((u_{r2} - u_{r1}) \ln \frac{q_{min} + 1}{q^* + 1} + u_{r1}(q_{min} - q^*))$$

$$\int_{p^*}^{p_{min2}} \frac{(u_{c1}p_t + u_{c2})}{\frac{p_t}{2} + 0.5} dp = \int_{q_{min}}^{q^*} \frac{(u_{r1}q_t + u_{r2})}{1 - \frac{q_t}{2}} dq$$

$$2((u_{c2} - u_{c1}) \ln \frac{p_{min2} + 1}{p^* + 1} + u_{c1}(p_{min2} - p^*)) \tag{6.19}$$

$$= 2(u_{r1}(q_{min} - q^*) + (u_{r2} + 2u_{r1}) \ln \frac{|q_{min} - 2|}{|q^* - 2|})$$

The results of the theoretical model and an empirical demonstration can be seen in Figure 6.2. Both variants maintain the *convergence* property. Bounded WPL has a slightly slower convergence speed than the original algorithm (which is to be expected, as the convergence speed has now been bounded to a smaller interval), but it maintains a very similar pattern to the original algorithm. However, the High WPL variant overcompensates and thus causes the policies to oscillate a lot more than the original algorithm, since we increased and off-set the average policy update rate. This is a highly undesirable effect, as it may lead to policy divergence in the learning stage.



(a) Bounded WPL. (b) High WPL.

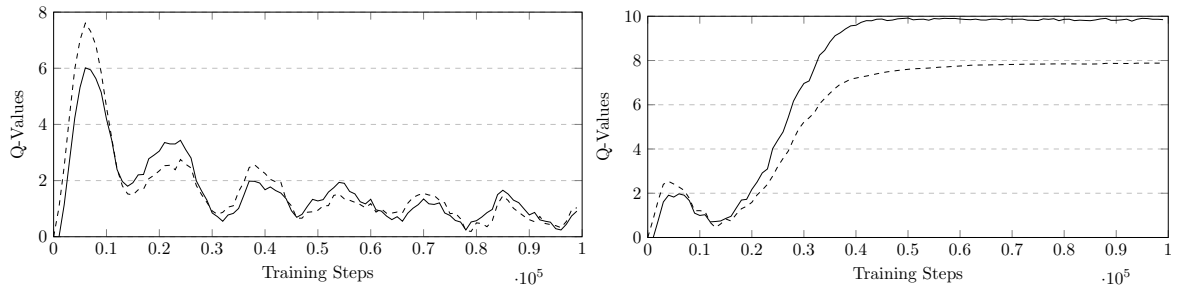
Figure 6.2: The evolution of the policies of the row player (solid) and the column player (dashed) in a Matching Pennies game, using each proposed variant. The squared marks represent the p_{max} and q_{max} values, while triangular marks represent p_{min} and q_{min} values. Simultaneously, solid marks represent p_{max} and p_{min} values and clear marks represent q_{max} and q_{min} values.

While both update rules eliminate the asymptotic convergence property of WPL to pure equilibria, this analysis shows that the Bounded variant outperforms the High variant when converging to stochastic equilibria. However, the smaller interval also affects the convergence to stochastic equilibria by making slower updates. Ideally, a mechanism to automatically adjust this interval could provide the best of both worlds. The interval would maintain its original size in stochastic NE games, and decrease in pure NE games.

6.2.3 Adjusted Bounded WPL

In order to automatically adjust the interval, such that scenarios with pure equilibria converge faster, and stochastic policy scenarios are not disturbed, an update rule based on the actions' Q-values is proposed. Because a pure equilibrium means that one action out-values all others, then Q-values converge such that the dominant action always has a higher Q-value than the remaining actions. In games with mixed policies, the Q-values of actions that belong

to the equilibrium oscillate around the same value, based on small variations of the agents' policies. This behavior can be observed in Figure 6.3.



(a) Matching Pennies. (b) Coop Matching Pennies.

Figure 6.3: The evolution of Q-values for two actions of a single player in two different games, using the original WPL algorithm. Matching Pennies has a stochastic NE, while Coop Matching Pennies has a deterministic NE.

The Adjusted Bounded WPL (ABWPL) proposal measures the frequency at which Q-values oscillate, and adjusts the policy update based on that. It averages the amount of steps taken for the maximum Q-value's action to change, and starts binding the $\pi_t(s, a)$ factor when an action has remained dominant for more than that average amount. The intuition behind this is that the Q-values usually oscillate with a decreasing period p_s as policies adjust (due to learning rates and action randomicity), and while they are oscillating, the algorithm is converging to a stochastic policy. ABWPL does not interfere with the update rule as long as Q-values are oscillating within the period p_s, since the dominant action is expected to change at its end.

If an action remains dominant for longer than p_s, the $\pi_t(s, a)$ factor is narrowed until it is a $[0.5 - n, 0.5 + n]$ factor for all actions, where n is an arbitrarily small non-zero positive value. At that point, ABWPL adjusts probabilities at nearly the same speed for both pure and stochastic policies, no longer asymptotically converging to pure equilibrium solutions, and keeping its convergence properties. Whenever an action is no longer dominant, the $\pi_t(s, a)$ factor is reset to its original $[0, 1]$, as the solution is once more expected to be a stochastic policy.

Formally, for a state s, given the dominant action with highest Q-value $a^d_{s,t}$, at time-step t, with t^d_s time-steps elapsed since the last reset (where that state's dominant action $a^d_{s,t}$ changed), and an expected total p_s time-steps for the dominant action to be replaced, ABWPL calculates a new bounded $\pi^b_t(s, a)$ factor to be within a $[f_{s,t}, 1 - f_{s,t}]$ interval by

$$f_{s,t} = \begin{cases} f_{s,t-1} + \frac{0.5}{p_s} \text{ if } a^d_{s,t} = a^d_{s,t-1} \text{ and } t^d_s > p_s \\ 0 \text{ otherwise} \end{cases}, \tag{6.20}$$

$$\pi^b_t(s, a) = \pi_t(s, a)(1 - 2f_{s,t}) + f_{s,t}, \tag{6.21}$$

where the constraint $f_t = [0, 0.5[$ is enforced outside the equation. The intuition here is to calculate the factor $f_{s,t}$ to decrease the policy update interval based on how long the oscillation period p_s is for state s. The factor $f_{s,t}$ reaches its maximum after the same action

has remained dominant for p_s time-steps after the oscillation period has elapsed. We then replace equation 2.9 in WPL with

$$\forall a \in A \ \Delta_t(s,a) = \eta_t^{\pi} \frac{\partial V_t(s)}{\partial \pi_t(s,a)} \begin{cases} \pi_t^b(s,a) \text{ if } \frac{\partial V_t(s)}{\partial \pi_t(s,a)} < 0 \\ 1 - \pi_t^b(s,a) \text{ otherwise} \end{cases} . \tag{6.22}$$

To calculate p_s, in order to avoid noise and keep a stable and gradual evolution, we found that a moving average filter with 2 windows and ignoring intervals where $t_s^d < \frac{p_s}{2}$ represented a robust approximation. Noise happens when actions have very similar Q-values, and so oscillate very quickly. This would cause p_s to decrease to a very small value, when in fact the actions were only oscillating due to randomness in the policies. When the time taken for a dominant action to change is too small (in our case, smaller than half of the current p), we assume it as noise. To make p_s change gradually, we average the previous and the new value, an approach followed in other algorithms (like CMA-ES) to bind the update step. However, we don't assume this approximation to be the only solution, and many other methods (possibly problem dependent) are expected to work. The algorithm is robust to different initial values for p_s. We used the minimum (and most aggressive) value $p_s = 1$ in our tests, and larger values simply cause the constraint $f_{s,t}$ to change slower, leading to a more conservative initial adjustment of the update rate.

If $f_{s,t} = 0$, the algorithm is the original WPL, and this situation occurs when there is no single dominant action. In other words, when the policy should converge to a stochastic NE, ABWPL maintains WPL's behavior. On the other hand, with a pure NE, the $\pi_t^b(s,a)$ factor ensures that the policy updates do not decrease as the policy approximates the limit.

This proposal has increased the state-wise memory consumption of the original algorithm, due to keeping track of several new values per state. However, we believe that the benefits of Adjusted Bounded WPL compensate for its drawbacks, as can be seen in the following section.

6.3 Evaluation

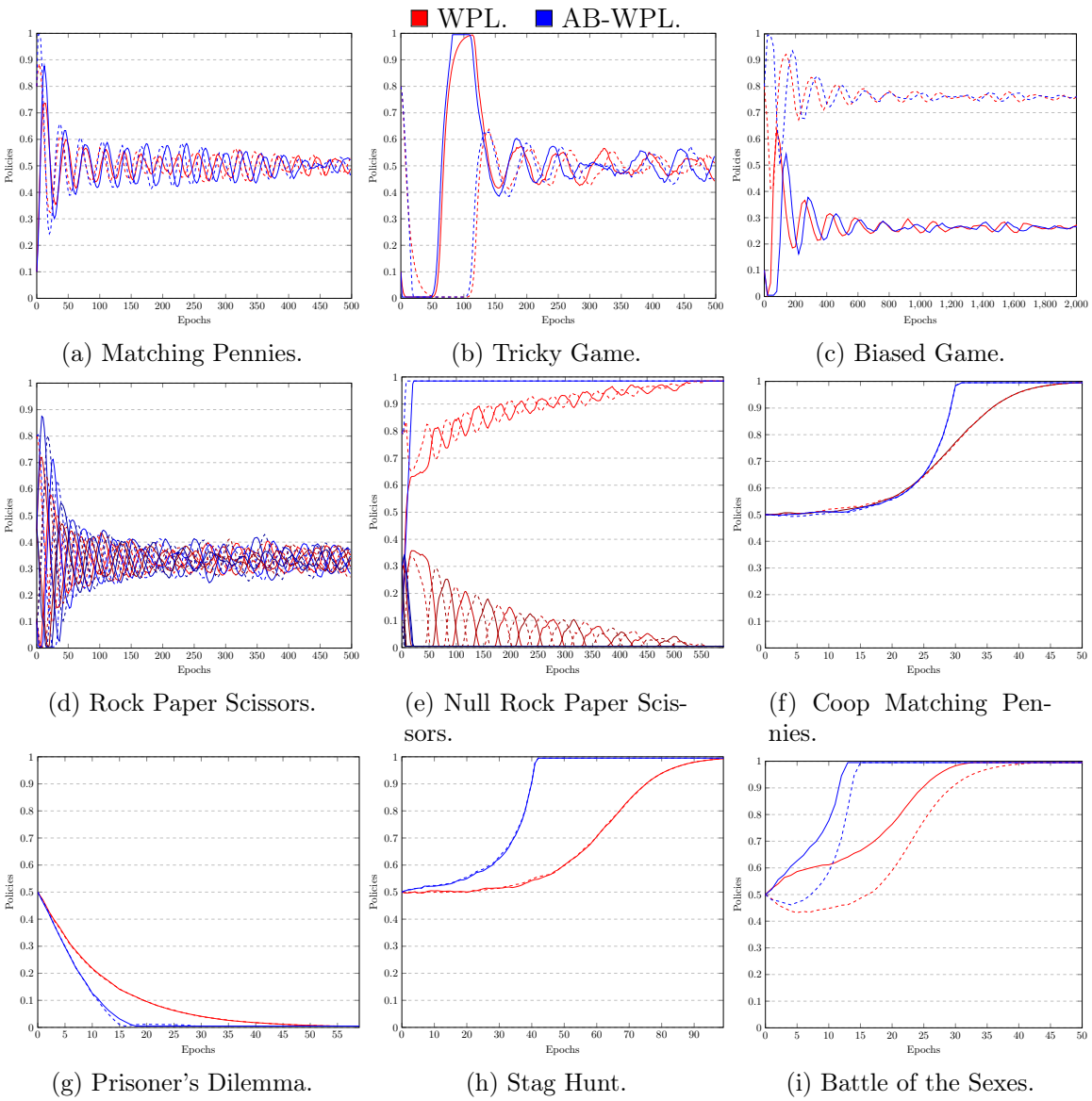
ABWPL is now compared against WPL in a set of game-theoretic scenarios and multi-state games, to demonstrate how the new update rule behaves in both pure and stochastic NE environments. These include the games described in Section 3.3, as well as the grid Soccer Kick and Soccer Keep-Away environments in Section 3.4.1. ABWPL is then compared against other mixed-policy algorithms, to assess its relative performance and robustness, in some of the game theoretic environments described previously.

Unless stated otherwise, plots are shown across epochs of 1000 iterations (x-axis), with an exploration rate $\epsilon = 0.05$, a learning rate $\eta = 0.01$, a policy learning rate $\eta^{\pi} = \eta/100$ and a discount factor $\gamma = 0.9$.

6.3.1 Comparing ABWPL and WPL

This section shows how Adjusted Bounded WPL behaves in comparison with the original WPL in both stochastic and pure NE games. The results for single-state game theoretic environments are shown in Figure 6.4. ABWPL matches the performance of the original WPL in all stochastic NE games, and outperforms it in all pure NE games. This is the expected behavior of the proposed policy update rule, where we speed up the convergence

when an action dominates others, but do not disrupt the learning process when a stochastic equilibrium causes actions to continuously oscillate.



(a) Matching Pennies. (b) Tricky Game. (c) Biased Game.

(d) Rock Paper Scissors. (e) Null Rock Paper Scissors. (f) Coop Matching Pennies.

(g) Prisoner's Dilemma. (h) Stag Hunt. (i) Battle of the Sexes.

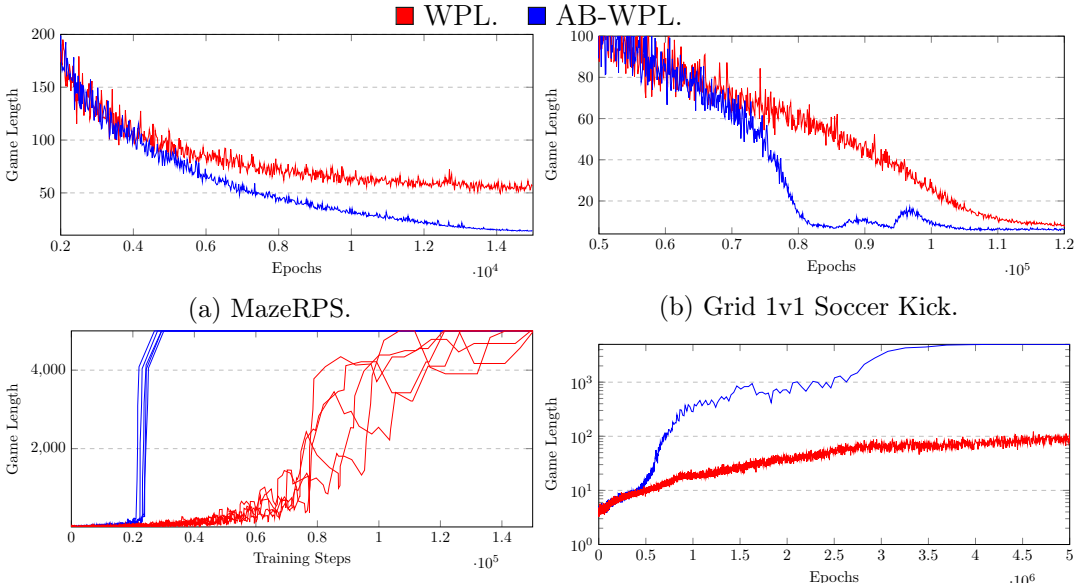
Figure 6.4: The evolution of the probability of playing the first action by 2 players in several 2-player games. For games with more than 2 actions, the probabilities of all actions are shown. The row player (solid) starts with an initial probability $p_0 = 0.1$ or $p_0 = 0.5$, and the column player (dashed) with an initial probability $q_0 = 0.8$ or $q_0 = 0.5$, depending on the game. The graphs represent the original WPL algorithm (red) and Adjusted Bounded Bounded WPL (blue).

ABWPL is now compared against WPL in MazeRPS, a grid 1v1 Soccer Kick, and grid 3v2 Keep-Away Soccer. A complete match of MazeRPS consists on two players having to cross a labyrinth, and playing a single round of Null Rock Paper Scissors. In 1v1 Soccer Kick, an attacker carries the ball and must feint the defender in order to score. The defender's goal is

to reach the attacker and predict the feint. In 3v2 keep-away soccer environment, 3 defenders with the ball need to cooperate to keep 2 attackers from reaching it. The defenders cannot move in one variant of the game.

The goal for agents in the MazeRPS and Soccer Kick environments is to end the game as quickly as possible, to maximize their expected rewards. In MazeRPS, this happens due to NRPS having a positive average reward, and in Soccer Kick, players get small penalties at each time-step. At the end of the learning phase, both WPL and ABWPL achieved the NE strategy and successfully complete the game. However, as we can see in Figure 6.5(a)(b), ABWPL outperforms the original WPL algorithm in terms of game-length, since the majority of states have deterministic optimal strategies. WPL prematurely converges in such states, and takes much longer to complete its matches.

Defenders in the Keep-Away Soccer environments have the opposite goal, to prevent the game from ending for as long as possible. Once an attacker reaches them, the ball is captured, the game ends, and defenders get a heavy penalty. Once again, it can be easily seen in Figure 6.5(c)(d) how ABWPL outperforms WPL, achieving a winning strategy in a fraction of the training steps. ABWPL defenders quickly reach a strategy where the attackers cannot capture the ball and the game ends only by time-step limit (5000 steps).



(a) MazeRPS.

(b) Grid 1v1 Soccer Kick.

(c) 3v2 Keep-Away (no defender movement).

(d) 3v2 Keep-Away.

Figure 6.5: The time-steps (y-axis) taken by two agents to play a complete game across training steps and epochs (x-axis). The plots represent the original WPL (red), and Adjusted Bounded WPL (blue).

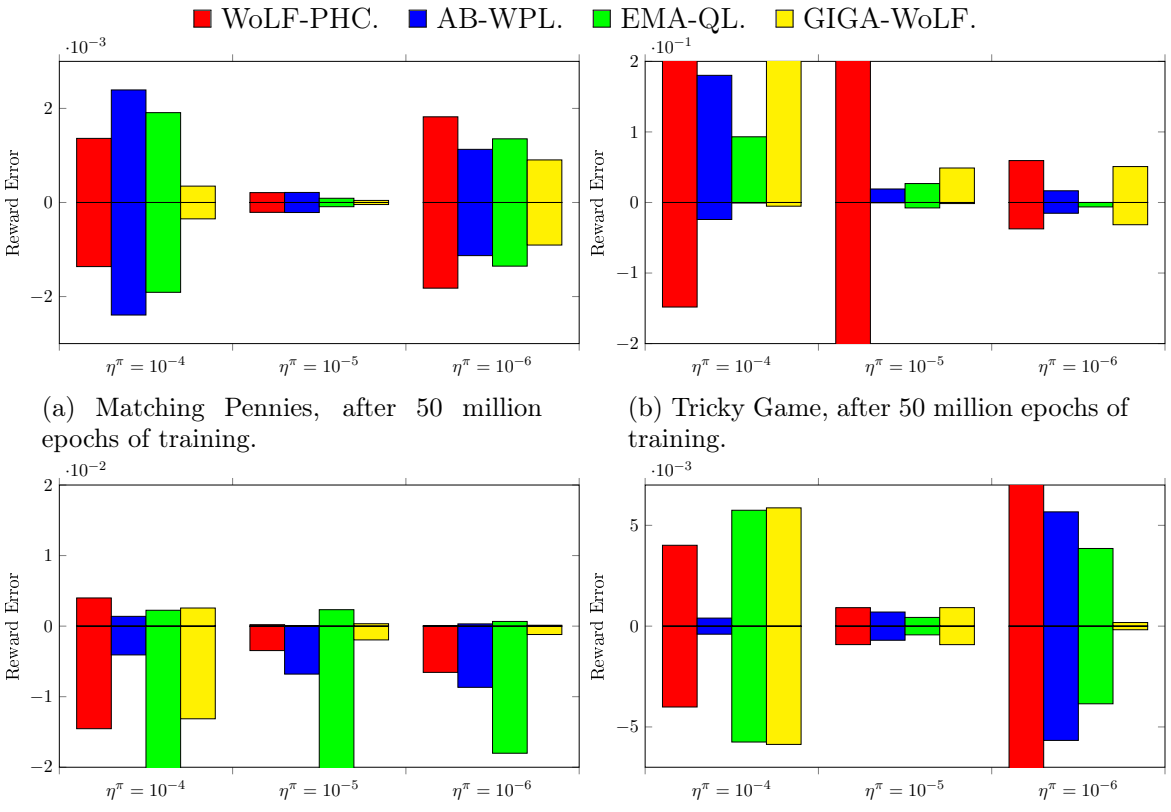
6.3.2 Comparing Mixed-Policy Algorithms

The performance of ABWPL is also compared with WoLF-PHC, EMA-QL, and GIGA-WoLF, other state-of-the-art stochastic search algorithms. Because all algorithms are based on Q-learning and share similar architectures, all four are compared using the same set of

hyper-parameters. The parameters were chosen such that all algorithms could converge to the NE solutions in the evaluated games.

Since the algorithms keep their own action distribution, the minimum probability of each action is set to be equal to the exploration rate ϵ divided by the number of available actions. The learning rate η affects all algorithms' Q-values in the same way, and the discount factor γ represents how important future rewards are. For algorithms that require two policy learning rates (for both winning and losing situations), the losing rate is $\eta_l^\pi = \eta^\pi$, and the winning rate becomes $\eta_w^\pi = \eta_l^\pi/2$. As such, the only hyper-parameter that affects each algorithm differently is the policy learning rate η^π. Therefore, all four algorithms are evaluated on several magnitudes of the policy learning rate η^π. Algorithms are given a sufficient number of epochs in the training phase to achieve convergence, and evaluated on their final quarter of epochs. The average error of each player's policy is measured against their NE strategies.

ABWPL matches the performance of other state-of-the-art algorithms for a set of mixed policy games, as can be seen in Figure 6.6. On all games except Matching Pennies, there is a learning rate for which ABWPL outperforms all other algorithms.



(a) Matching Pennies, after 50 million epochs of training.

(b) Tricky Game, after 50 million epochs of training.

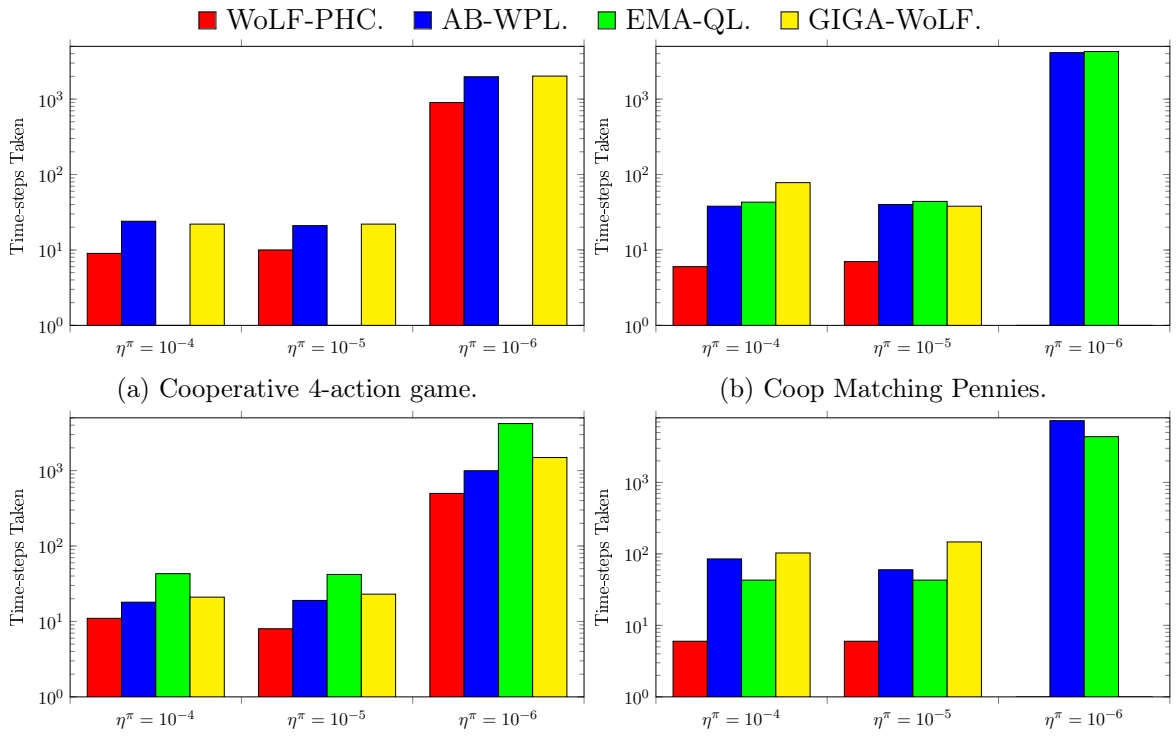
(c) Biased Game, after 200 million epochs of training.

(d) RPS, after 50 million epochs of training.

Figure 6.6: Average reward error (y-axis) of WoLF-PHC (red), ABWPL (blue), EMA-QL (green), and GIGA-WoLF (yellow) against the expected returns of the NE strategy. The error of player 1 is shown above the 0-line, and of player 2 below, and plots are shown over different policy learning rates (x-axis).

Figure 6.7 shows the time taken for the same algorithms to converge in pure policy games.

ABWPL is outperformed by WoLF-PHC in most scenarios, but with very small learning rates, WoLF-PHC did not converge to a policy in a Cooperative Matching Pennies game, since agents could not decide which equilibrium strategy to converge to. However, ABWPL can match the remaining algorithms' performance, and is the only out of all four that converged to a correct strategy in all tested magnitudes of the policy learning rate η^π.



(a) Cooperative 4-action game.

(b) Coop Matching Pennies.

(c) Prisoner's Dilemma.

(d) Stag Hunt.

Figure 6.7: Average time-steps taken (y-axis, logarithmic scale) to converge to a pure strategy, for the policies of WoLF-PHC (red), AB-WPL (blue), EMA-QL (green), GIGA-WoLF (yellow), over different policy learning rates (x-axis).

6.4 Conclusion

WPL has been shown to achieve convergence in complex 2-player games and in games with up to 100 players, despite having no formal analysis and proof of convergence. However, it is biased against deterministic strategies, and the policy update rate tends to zero in pure NE games.

ABWPL is a WPL extension with a new update rule that allows the algorithm to converge to both deterministic policies (where some actions dominate others) and stochastic policies, by regulating the policy update rate based on the expected rewards for each action. Despite the increased memory consumption, great improvements in the convergence speed are shown. ABWPL is robust to hyper-parameter changes, maintains all the convergence properties and speed of WPL in mixed policy games, is faster in pure policy games, and can match the performance of other state-of-the-art mixed-policy algorithms.

Techniques like A3C rely on approximating the value function through an artificial neural network. This non-linear approximator allows agents to handle both discrete and continuous high-dimensional state-spaces, as well as to adapt to new unseen scenarios. In comparison, some of ABWPL's shortcomings, being a table-based algorithm, are its inability to adapt to previously unseen states, to handle high-dimensional or continuous state-spaces.

Chapter 7

Asynchronous Advantage Actor Centralized-Critic with Communication

Partially-observable multi-agent environments feature a number of challenges for reward-based learning algorithms. Each agent must handle all the complexities and issues that exist in a single-agent environment, including the underlying model's complexity, the partial-observability of the environment, or high-dimensional action-spaces. Agents must then also handle non-stationary environments, possible information exchange between them, the credit assignment problem, and the convergence to rational cooperative policies.

Both Independent Q-Learners and Joint-Action Learners are applicable to actor-critic deep-learning methods, where each agent has its own actor and critic, from its own state-action history. Independent actor-critic is straightforward, but the lack of information sharing makes it difficult for agents to learn coordinated strategies that depend on information known by others, as well as for an individual agent to estimate the contribution of its actions to the team's reward. It is also insufficient in partially-observable environments where agents must share local information to successfully complete a task. Joint-Action actor-critic suffers from the original problems of lack of scalability, and centralized execution.

With partially-observable environments, agents obtain observations o about the environment's underlying state s. These observations are often incomplete or noisy, and may not allow agents to achieve globally optimal policies. To compensate for partial observations, agents may have to rely on other agents' knowledge. This raises issues related with sensor fusion (how to optimally combine the agent's perceptions) and decision making under partial observability (which can be an intractable problem [205]). Regardless of whether agents observe the environment's state or a partial observation, they can also perceive a global observation, or local perspectives of their observations, with spatial (at different locations), temporal (arriving at different times) and semantic (with different interpretations) differences.

Recent research has both focused on implicitly shared information while agents converge to policies, or on explicit communication while policies are executed. Such communication protocols can be learned *tabula rasa* [44, 45], hard-coded by researchers [42, 43], or derived from symbol alphabets [109, 110]. Communication is a general and flexible approach, that allow agents to share both low- and high-level information [37], despite being constrained by communication restraints of the environment (e.g., distance). The transmission of local

information compensates for the partial observability of the environment, and helps reduce the complexity of a challenge.

This chapter proposes Asynchronous Advantage Actor-Centralized-Critic with Communication (A3C3), a multi-agent deep actor-critic algorithm. Agents are executed independently, but a centralized critic is used in the learning phase for agents to learn implicit coordination, while also speeding up the learning process and increasing its robustness. The critic's complexity increases with the amount of agents in the environment, so an additional permutation invariance technique is described to increase the critic's scalability. Agents also learn their own communication protocols, through which they share relevant information for other team members, even with noisy communication channels. A3C3 is compared with other state-of-the-art algorithms, as well as single-agent algorithms implemented with the Independent Q-Learners approach. Tests are conducted to demonstrate the effects of learning communication, and an analysis of the learned protocols is presented, where direct correlations between local observations (e.g., each agent's location) and the sent messages can be seen. These protocols allow agents to handle partially-observable environments. The effects of noise in transmitted messages are also analyzed. A permutation invariant technique for scalability is described in a swarm environment, and the architecture invariance of A3C3 is also shown. These findings were partially published in the WorldCIST19 [61], IJCNN19 [62], and ROBOT19 [63] conferences. They have also been submitted in the Neurocomputation journal and as an extended publication in the ICAE and JAISCR journals. The algorithm's source-code and tests were published at `https://github.com/david-simoes-93/A3C3`.

7.1 Problem Statement

This chapter focuses on multi-agent cooperative environments with J agents modeled as Dec-POMDP, with both homogeneous joint-rewards (where agents receive points as a team), and cooperative heterogeneous rewards (where agents receive individual rewards but benefit from cooperating with one another). Each agent j has local partial observations o_t^j of the environment at each discrete time-step t. An observation o_t^j is a (usually incomplete) representation of the environment's state s_t, and can be noisy, as well as discrete or continuous. Orthogonally, observations may also be local or global. Agent j acts independently upon the environment, through a discrete set of actions $\mathcal{A}^j$, and obtains reward r_t^j based on a reward function $\mathcal{R}^j$ determined by the environment.

Agents must be executed in a distributed and independent manner, based solely on their own local observations. However, during the learning phase, algorithms can take advantage of centralized architectures, and incorporate additional information from the environment or from the agents into their policies. This includes concatenating agent observations, or accessing the underlying state s, for a better estimation of the value function. This additional information is no longer available during the execution phase. The amount of agents J can vary throughout each episode, depending on the environment (e.g., a race can initially have three vehicle agents, but only two remain active after one finishes the track).

Agents can also share information between them through message passing. Messages may have limited range, where agents can only communicate with geographically close team members. Messages can also be size-limited, and agents may not be able to transmit large sets of continuous values, but instead only single bits of information. Finally, messages may also have target constraints, and cannot be broadcast to the entire team, but instead directly

sent to a specific target. Not only that, but communication channels are imperfect, and suffer from noise. Messages can be delayed or, in the worst case, lost. They can also suffer external interference (e.g., random noise over the message's values) or suffer from internal jumbling (e.g., an agent receives the average of two messages, instead of receiving each message separately and distinctly).

Deep-learning contributions in the last few years for multi-agent environments have ranged from approaches with implicit coordination to learning communication protocols, but often do not adhere to all these assumptions.

COMA [131] allows agents to run in a distributed manner, but its critic's complexity is proportional to the amount of agents in the environment, it does not support heterogeneous reward functions, and does not use communication between agents. MADDPG [23] is similar to COMA, and introduces a communication mechanism to handle partially-observable environments, by formulating message passing as an additional discrete action-space. This not only assumes a discrete communication alphabet defined *a priori*, but also assumes noiseless communication. VDN [154] also uses a centralized critic, and communicates by sharing network nodes at run-time, thus requiring centralized execution. QMIX [99] combines each individual value function in a more complex manner, but disregards communication between agents.

BiCNet [141] does not support distributed execution and its agents communicate by sharing the RNN's internal states with their hierarchical neighbors, which is not a fully decentralized approach. CommNet [44] agents communicate by sending multiple messages at each time-step, but CommNet outputs a joint-action, which prevents distributed execution. The authors also assume noiseless communication between agents. DIAL [142] agents exchange discrete messages, but authors assume perfect communication, and test their proposal in a limited set of short-horizon environments. The applicability of DIAL to complex environments is not described, but DIAL disables the *experience replay* feature, which was shown to be an essential component of DQN [26] to achieve successful policies in complex environments like the Atari 2600 test-bed [206].

Some proposals [110, 109, 155] use a pre-determined vocabulary to communicate between agents, which must be defined *a priori*, using heuristics or random processes. The algorithms also do not allow the vocabulary to grow or otherwise change without repeating the learning phase. Other proposals [159, 160] deviate from pure deep reward-based learning approaches, and instead mix reward-based with supervised learning techniques. This require researchers to provide a pre-determined solution for agents to imitate, which may introduce human bias in the provided solutions, or may simply be impossible in complex environments.

7.2 Proposal

This chapter describes an adaptation of the A3C algorithm to the non-competitive multi-agent paradigm. The proposal, Asynchronous Advantage Actor Centralized-Critic with Communication (A3C3), requires a centralized learning phase, but supports distributed execution. A centralized critic implicitly shares information during the learning phase (by having access to the observations of all agents), allowing policies to robustly converge and for implicit coordination to be learned. Agents communicate with continuous-valued messages through noisy channels at each time-step, to compensate for partially-observable environments. The communication protocols are learned independently for any population, *tabula rasa*, and improve

team-mates' policies. They do not require any domain knowledge for their definition, thus preventing researchers from having to *a priori* determine what or how to communicate.

Each agent is represented by an actor, a centralized critic, and an additional communication network, as shown in Figure 7.1. The actor network remains decentralized, and outputs agent j's action probability distribution based on its current observation o_t^j. The centralized critic now outputs a value estimation based on a centralized observation O_t^j, which can be an aggregate of all agents' local observations, or the environment's underlying state when possible. The central critic acts as an implicit information sharing mechanism, by having access to additional information, which makes gradient updates much more robust. The communication network outputs a message sc_t^j sent by each agent j, which is then received by its team members at time-step $t+1$. The communication networks are optimized in order to help agents improve their policies, by handling partially-observable domains in a cooperative setting. Both the central critic and the communication network techniques thus improve the coordination level of the team.

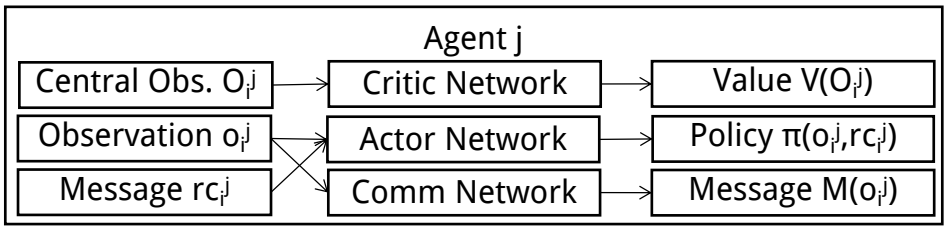


Figure 7.1: The architecture of an agent j at time-step i, using three separate networks: a policy (or actor) network, which outputs an action probability $\pi(o_i^j; rc_i^j)$ (from which a_i^j is sampled) based on a given local observation o_i^j and received messages rc_i^j from other agents; a communication network, which outputs an outgoing message sc_i^j based on a given local observation o_i^j; and a value (or critic) network, which outputs a value estimation based on a given centralized observation O_i^j.

Multiple workers asynchronously update all networks for each agent, by periodically making local copies of the networks, using them to calculate gradients, and applying the gradients on the global networks, as shown in Figure 7.2. A3C3 is a more general version of A3C, since if the number of agents $J = 1$, there is no communication, and the centralized critic uses the agent's local observation, the algorithm becomes A3C. We describe the behavior of a worker thread in Algorithm 15.

A3C3 can be horizontally scaled by increasing the amount of workers, which increases the amount of samples and updates per unit of time, and speeds up the learning process. Computations may also be sped up if networks with the same input use intra-agent parameter sharing for all layers but the last one [207, 57, 59]. This technique consists on having a single network with two output layers, instead of two separate networks. The error of both outputs is summed to optimize the shared network. For example, if all networks take as input the agent's local observation and received messages (effectively making the critic decentralized), each agent may be represented by a single network with three output layers.

If agents are homogeneous, A3C3 can also use inter-agent parameter sharing, to further speed up the learning process. This allows the same actor, critic, or communication network to be learned by all agents simultaneously. Since agents have different perspectives and batches of experience, when using inter-agent parameter sharing, each agent's mini-batch of samples will

Input: Globally, shared learning rate η, discount factor γ, entropy weight β, number of agents J, actor network weights θ_a^j, critic network weights θ_v^j, communication network weights θ_c^j, batch size $t_{\max}$, maximum iterations $T_{\max}$, and default message value rc_{initial}. Locally, actor network weights ϑ_a^j, critic network weights ϑ_v^j, communication network weights ϑ_c^j, and step counter t.

$t \leftarrow 0$
$rc_0^j \leftarrow rc_{\text{initial}}$ for all agents j
for iteration T $= 0, T_{max}$ **do**
  Reset gradients $d\theta_a^j \leftarrow 0$, $d\theta_v^j \leftarrow 0$, and $d\theta_c^j \leftarrow 0$ for all agents j
  Synchronize $\vartheta_a^j \leftarrow \theta_a^j$, $\vartheta_v^j \leftarrow \theta_v^j$, $\vartheta_c^j \leftarrow \theta_c^j$ for all agents j
  $t_{start} \leftarrow t$
  Sample observation o_t^j for all agents j
  Sample or derive centralized observation O_t^j for all agents j
  repeat
    for agent $j = 1, J$ **do**
      Calculate message sc_t^j to send with $sc_t^j \leftarrow M(o_t^j, \vartheta_c^j)$
      Sample action a_t^j according to policy $\pi(a_t^j | o_t^j, rc_t^j, \vartheta_a^j)$
      Map sent communication sc_t^j into received communication rc_{t+1}, and build communication map m_t
    end for
    Take action a_t^j for all agents j
    Sample reward r_t^j and new observation o_{t+1}^j for all agents j
    Sample or derive centralized observation O_{t+1}^j for all agents j
    $t \leftarrow t + 1$
  until terminal o_t^j for all agents j or $t - t_{start} = t_{max}$
  for agent $j = 1, J$ **do**
    $R^j = \begin{cases} 0 \text{ for terminal observation} o_t^j \\ V(O_t^j, \vartheta_v^j) \text{ otherwise} \end{cases}$
    $L_c \leftarrow 0$
    for step $i = t - 1, t_{start}$ **do**
      $R^j \leftarrow r_i^j + \gamma R^j$
      Value loss $L_{v_i}^j \leftarrow (R^j - V(O_i^j, \theta_v^j))^2$
      Actor loss
      $L_{a_i}^j \leftarrow \log \pi(a_i^j | o_i^j, rc_i^j, \vartheta_a^j) GAE(\gamma, 0) = (r_i^j + \gamma V(O_{i+1}, \theta_v^j) - V(O_i, \theta_v^j)) - \beta \times H(\pi(a_i^j | o_i^j, rc_i^j, \vartheta_a^j))$
    end for
    for step $i = t, t_{start} + 1$ **do**
      Received communication loss $L_{rc_i}^j \leftarrow \frac{\partial L_{a_i}^j}{\partial rc_i^j}$
    end for
  end for
  for step $i = t, t_{start} + 1$ **do**
    Map received communication loss $L_{rc_{i+1}}$ into sent communication loss L_{sc_i} using communication map m_i for all agents
  end for
  for agent $j = 1, J$ **do**
    for step $i = t - 1, t_{start}$ **do**
      Accumulate gradients $d\theta_c^j \leftarrow d\theta_c^j + \frac{\partial L_{sc_i}^j}{\partial \vartheta_c^j}$
      Accumulate gradients $d\theta_a^j \leftarrow d\theta_a^j + \frac{\partial L_{a_i}^j}{\partial \vartheta_a^j}$
      Accumulate gradients $d\theta_v^j \leftarrow d\theta_v^j + \frac{\partial L_{v_i}^j}{\partial \vartheta_v^j}$
    end for
  end for
  Update network weights $\theta_a^j \leftarrow \theta_a^j + \eta d\theta_a^j$, $\theta_v^j \leftarrow \theta_v^j + \eta d\theta_v^j$, and $\theta_c^j \leftarrow \theta_c^j + \eta d\theta_c^j$ for all agents j
end for

Output: Converged centralized-critic, actor, and communication networks for each agent.

Algorithm 15: Pseudo-code for a worker thread running A3C3. Workers copy global parameters into the local networks. They repeatedly sample the observations and rewards for all agents and output corresponding actions and messages. Actions are taken on the environment until a mini-batch has been gathered, or until a terminal state is reached. Workers then compute the loss of the local networks, apply those gradients on the global parameters, and repeat this process until convergence has been achieved.

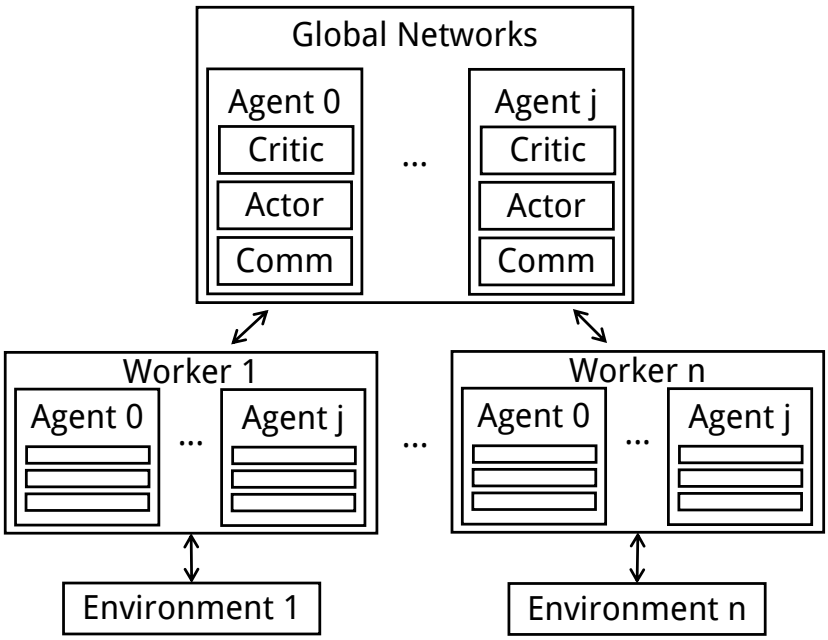


Figure 7.2: A3C3 architecture, using n separate workers. Each worker interacts with its own environment and its separate set of j agents. As samples are collected in mini-batches, workers asynchronously update the global networks, and copy those weights into their local networks.

---

break correlations in the network updates, in the same way multiple workers do so in A3C's single-agent learning. In other words, with homogeneous agents and inter-agent parameter sharing, A3C3 can optimize agent policies with a single worker. Multiple workers speed-up the learning process, but a single worker may be required in contexts with hardware limitations, such as when a GPU is required by the environment but only one is available, or when using older commodity hardware.

In the limit, if agents are homogeneous and the critic is decentralized, A3C3 can optimize a single network (with three output layers) for all agents. However, this computational increase comes at the cost of a more complex network being necessary to approximate multiple functions simultaneously. It also causes agents to have homogeneous policies, since their experience is now optimizing the same network.

7.2.1 Actor Network

The actor networks with weights θ_a^j for each agent j aggregate observations o_t^j and received messages rc_t^j as input, and output policy $\pi(a_t^j|o_t^j, rc_t^j, \theta_a^j)$ through a softmax function. For scalability, agents can pre-process the received messages in a number of ways, instead of aggregating them, reducing the complexity of the network at the cost of losing information from messages. Options include averaging the received messages, choosing (possible randomly) a single message to receive, having a protocol where a single message is sent and received by the team per cycle, among others. An exemplary actor network is shown in Figure 7.3, where an agent aggregates received messages and moves north with 70% chance, or south otherwise.

Workers optimize the actor loss $L_{a_i}^j$ based on Generalized Advantage Estimation (GAE)

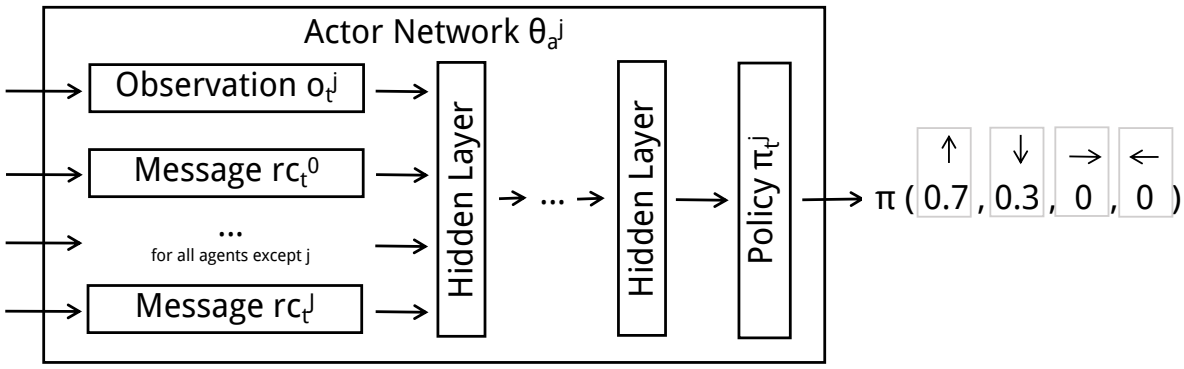


Figure 7.3: An exemplary architecture of agent j's actor network. In this case, the actor aggregates the observation and the broadcast message of all other agents as its input. The network's output layer then outputs a probability distribution for agent j's movement in four possible directions. The output layer is directly based on the environment's action space.

---

[208], the logarithm of the policy, and an entropy factor $H(\pi(a_i^j|o_i^j, rc_i^j, \vartheta_a^j))$. The advantage estimator uses $GAE(\gamma, 0) = (r_i^j + \gamma V(O_{i+1}, \theta_v^j) - V(O_i, \theta_v^j))$, which leads to low variance updates. The advantages are then multiplied by the logarithm of the policy $\log \pi(a_i^j|o_i^j, rc_t^j, \vartheta_a^j)$. An entropy factor β determines the weight of the policy's entropy loss $H(\pi(a_i^j|o_i^j, rc_i^j, \vartheta_a^j))$ and discourages premature convergence [209].

Agents with homogeneous action-spaces and reward functions can use inter-agent parameter sharing in the actor networks.

7.2.2 Centralized Critic Network

The centralized critic networks with weights θ_v^j for each agent j use centralized observations O_t^j as input, and output expected value estimations $V(O_t^j, \theta_v^j)$, as shown in Figure 7.4. This centralized observation is environment dependent, and can be:

- Sampled as a fully-observable environment state, common to all agents. This may not be possible in some environments;
- Derived as a fully-observable environment observation, from the computation of each agent's partial local observation. This requires that the concatenation of all agents' partial observations o^j can create a fully-observable observation;
- Derived as a partial observation of the environment, from the computation of each agent's partial local observation. This is the least restrictive option, as it makes no assumptions on the observability of the environment.

This centralized observation O_t^j can also incorporate the actions of all other agents $k, \forall k \neq j$, the messages received by the current agent j, or both. If actions are incorporated, then the value function is now stationary, regardless of the policies of other agents. This technique is used by COMA to estimate the contribution of each agent for the overall expected reward. If the messages are incorporated, then some redundancy is added into the network, as the messages were calculated based on local observations, which are also included in the centralized

observation. This redundancy causes a simpler value function to be approximated, as the critic now has access to all agents' complete input. However, at the same time, this increases the input size and complexity of the network, which may offset its benefits.

Workers optimize the critic loss $L^j_{v_i}$ based on the squared difference between the actual returns R and the value estimation $V(S^j_i, \theta^j_v)$. The critic estimation bootstraps the next state's expected value at the end of a mini-batch, if a terminal state has not been reached.

Agents with homogeneous reward functions can use inter-agent parameter sharing in the critic networks. A3C3 supports using the same centralized critic for all agents (following the approach of COMA), or the more general case of a centralized critic for each agent (following the approach of MADDPG).

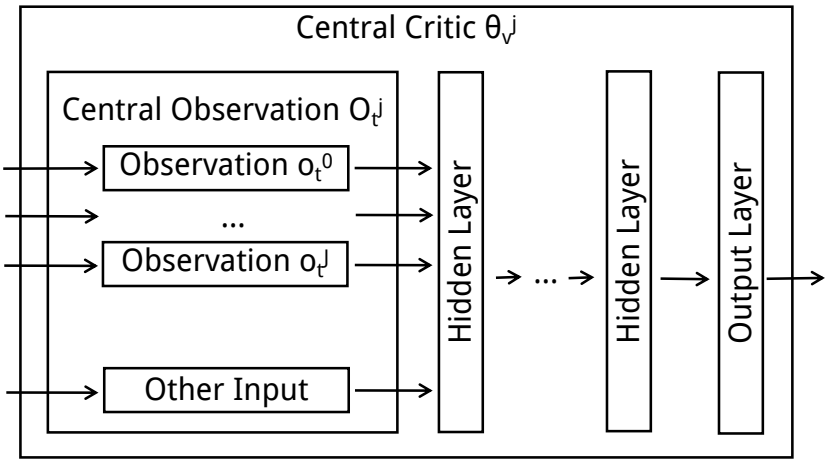


Figure 7.4: An exemplary architecture of agent j's centralized critic. In this case, the centralized observation O^j_t concatenates all agent observations as well as some additional information from the environment. Other input architectures are supported, each with its own advantages and requirements. If a centralized learning phase is not supported and the environment does not provide access to any additional information, the centralized observation O^j_t takes as input solely the agent's local observation. This in effect creates a decentralized critic, which has to approximate a non-stationary value function. On the opposite end of the spectrum, if the environment does provide access to its underlying state, the centralized observation O^j_t does not require possibly redundant or noisy agent observations, it simply becomes the complete environment state s_t.

7.2.3 Communication Network

The communication networks with weights θ^j_c for each agent j use observations o^j_t as input, and output messages $M(o^j_t, \vartheta^j_c)$. The output layer of this network defines the size and type of the generated messages. In other words, n-channel messages are generated by a network with an output layer of n nodes, and its activation function defines the value range of each channel. For example, an 8-node output layer using binary activations computes single-byte messages, as shown in Figure 7.5.

Communication constraints are environment-dependent, and can be classified based on reliability, connectivity, parallelism, or others. Messages can be broadcast to all other agents

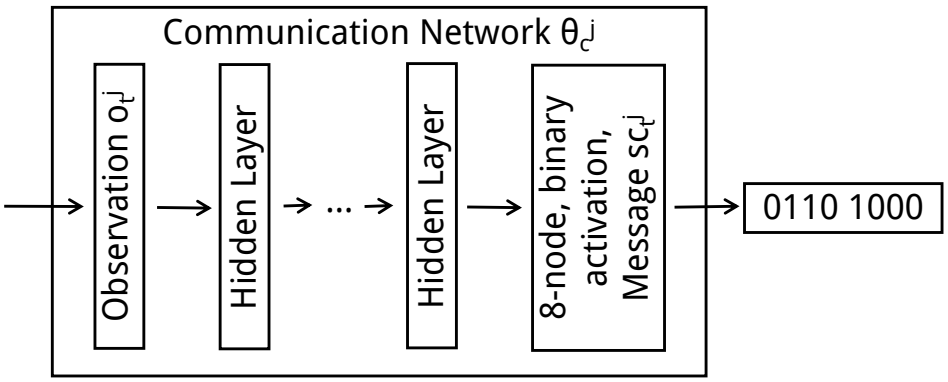


Figure 7.5: An exemplary architecture of agent j's communication network. In this case, the network's output layer uses a binary activation function and has 8 nodes, generating single-byte messages. Other output architectures are supported, including continuous valued messages. For example, a 10-node layer with *tanh* activations outputs a vector with elements $x_i, i = 1, \ldots, 10$, where each element $x_i \rightarrow [-1, 1]$.

---

or sent to specific recipients, depending on the environment. Received messages can be averaged to maintain a simpler network architecture, at the cost of losing some information. Broadcasting messages to all other agents may lead to scalability issues with large teams. If agents expect to send or receive a specific amount of messages based on the team size, then the networks may not handle varying numbers of agents during execution. These communication properties are captured within a communication map, built by each worker thread, that describes which agents sent which messages to which agents in the current batch of samples. An undelivered messaged takes a value rc_{initial} for the network input layer. If agents send messages to themselves, they create a memory channel through which information from previous states can still be accessed.

A sent message sc_t^j is received by other agents in the next time-step as rc_{t+1}^j. Gradients are first computed on the actor network with respect to the received messages, and is modeled as the error of received messages $L_{rc_{i+1}}$. Those are then applied to the sent messages (through the communication map) as message loss L_{sc_i}, which is then minimized by optimizing the communication network. The loss can be summed or averaged, if a sent message is received by multiple agents, which can lead to large network updates, or steady slow ones, respectively. Figure 7.6 shows an example where three agents broadcast messages to others. The actor error for agents 1 and 2 is propagated into the message sent by agent 0 on the previous time-step, and used to optimize the communication network. This makes agents optimize the messages they send such that other agents improve their policies, thus causing agents to send relevant information and enforcing coordination.

Because gradients of a received message rc_{t+1}^j are propagated across the last time-step's communication network of agent j to optimize message sc_t^j, the agent learns to transmit as much relevant information as possible from its observations o_t^j. Information is considered relevant when it improves the policies of team members, since the environments are cooperative.

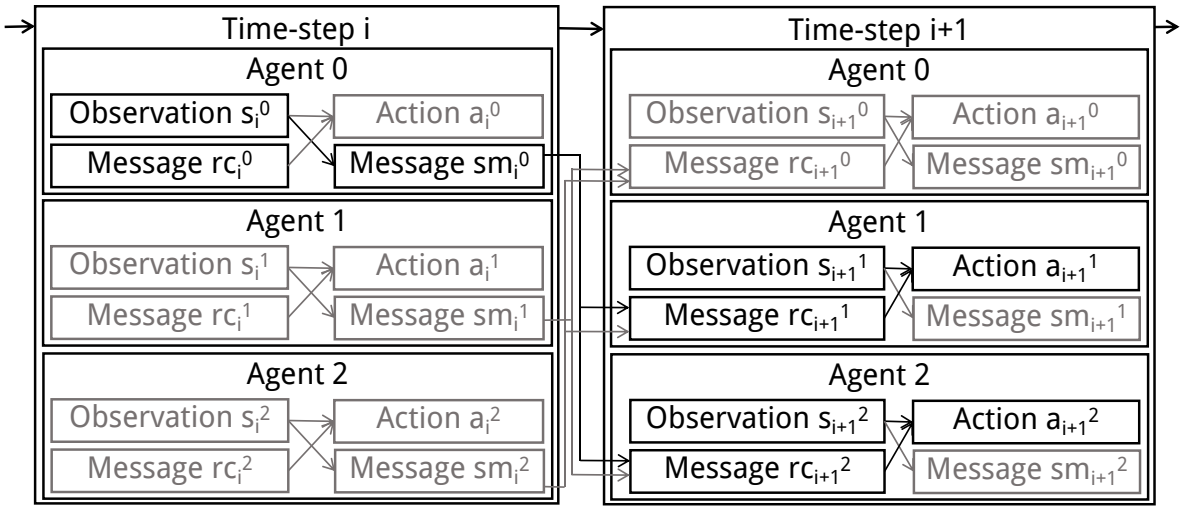


Figure 7.6: Diagram of how broadcast communication with three agents is performed across two time-steps (arrow direction), and of how gradients for the communication network are propagated backwards (emphasized lines). This architecture can be extended to an arbitrary amount of agents. If messages are not broadcast, gradients are pushed only to the corresponding message senders, based on the built communication map.

7.2.4 Permutation Invariant Networks

As the amount of agents in the environment increases, the scalability of A3C3 is hindered by the centralized critic. Since this network commonly aggregates observations from all agents (although it is not necessary to), its complexity is directly proportional to the amount of agents on the team. An additional issue, found with homogeneous teams, is that switching agents' identifiers (and therefore the order with each each agent's observation is fed to the critic) should also not affect the value estimation. However, a neural network approximator is not permutation invariant with respect to each agent's observation. In an environment with at least two homogeneous agents, the order with each each agent's observation is aggregated to the input of the network affects its value estimation. The problem is further accentuated when the amount of agents grows to large values.

In swarm scenarios, where there are usually tens of homogeneous agents, a critic must derive a value function V using a very large input layer, and it must also learn that agents' inputs are permutation invariant. Receiving the positions of agents ranging from index $0, J$, from $J, 0$, or from any other permutation, should output the exact same value expectation. It is possible for algorithms to learn permutation invariant policies in MAS with a small amount of agents simply based on randomness and on the algorithm's ability to learn similar policies for states where agents are in opposite locations [56, 62, 61]. With only two to four agents, A3C3 can learn to output similar value estimations for agents in the same positions but listed in different orders. However, once the amount of agents grows large, this becomes factorially more difficult to learn for a network, since the amount of possible permutations $P = J!$.

The question then becomes *how to allow a network whose input contains a set of agent-dependent information to become permutation invariant to the order of this set.* More specifically, how to allow the centralized critic to ignore permutations in homogeneous agents'

observations. While it is possible to learn some kind of permutation invariance by feeding the network enough data that it learns to be robust to agent permutations, it is a slow method, and does not scale well.

A possible solution is to maintain the same network structure, but pre-process its input. By ordering the set of agents based on their observations, the network will output the same value estimation regardless of the original set's order. When each agent's observation is 1-dimensional, this is a trivial task. However, even in the simple case of a 2-dimensional observation of (x, y) coordinates, defining a proper ordering metric can be complex. With (x, y), agents can be ordered by the average or sum of their coordinates, their L^2 norm, or by prioritizing a coordinate and using the other to resolve ties. Different methods will likely yield different results. If the agents' 2-dimensional observation is not coordinates, but unrelated values, this task becomes even more complex and entirely problem-dependent. While in some cases is will probably be an adequate technique, it is not general enough.

This section describes a technique where the network architecture is itself permutation invariant, which has been independently researched as Deep M-Embeddings (DME) [210]. DMEs consist on an initial network θ_p that extracts F features from each agent's observation, and then applies a permutation invariant filter on these features. From there, the remainder of the network is permutation invariant to the order of agents and estimates the value function as before. The feature extraction network must use the same weights θ_p for all agents j, in order to maintain its permutation invariance properties, and outputs a set of features λ with dimensions $[J, F]$. The permutation invariant filter is essentially an operation that reduces the dimensionality of the feature extraction network's output into a single vector Λ of length F, like a mean function

$$\Lambda_f = \frac{\sum_{j=1}^{J} \lambda_{j,f}}{J}, \text{for all features } f$$

across agent observations. Other possible functions include the maximum function

$$\Lambda_f = \max_j \lambda_{j,f}, \text{for all features } f$$

or the weighted softmax function

$$\Lambda_f = \sum_{j=1}^{J} w_{j,f} \lambda_{j,f}, \text{with } w_{j,f} = \frac{e^{\beta_f \lambda_{j,f}}}{\sum_{j=1}^{J} \beta_f \lambda_{j,f}}, \text{for all features } f,$$

where β is a set of weights, each for its corresponding feature, which is also optimized. The reduced vector Λ can then be used as a regular network layer for the remainder of the centralized value network θ_v^j. Figure 7.7 shows an example of the proposed architecture.

Popular deep learning frameworks [211, 212] do not feature DME layers, and implementing them by hand can lead to unoptimized or incorrect code. Therefore, a simpler solution is to take advantage of pre-existing architectures, specifically convolutional layers, to emulate the behavior of DME. Convolutional layers use a kernel, usually with $[P_x, P_y, C]$ dimensions, with P_x and P_y corresponding to pixels, and C being the amount of color channels in the image. This kernel is repeatedly slided across the input image, with a given stride, representing the amount of pixels the kernel is slided. A stride of 1 means the kernel is moved one pixel at a time. As the kernel is slided, it analyses small parts of the original input image, with dimensions $[P_x, P_y]$, and generates an output of depth f for each one, which corresponds to detected features (like edges, or geometric shapes).

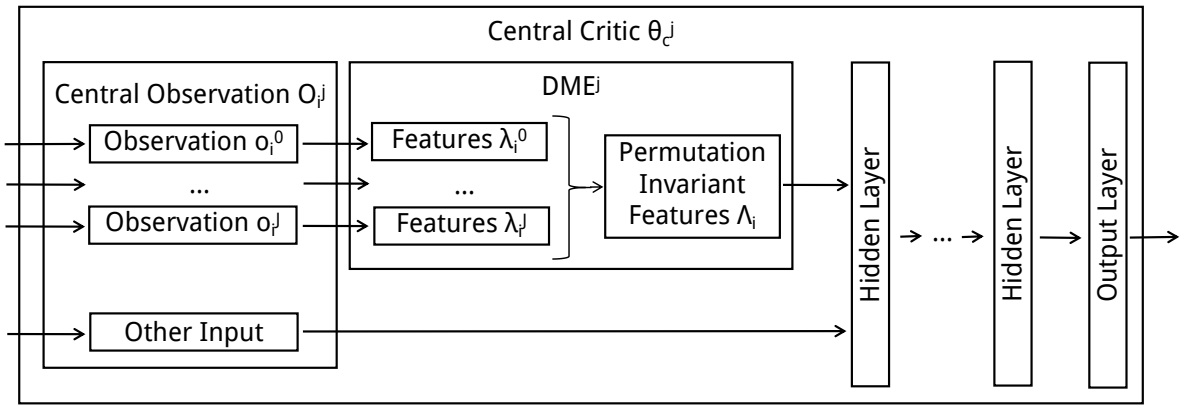


Figure 7.7: The architecture of a permutation invariant Central Critic for agent j. The critic takes as input the set of observations from all agents as well as any other relevant information that is available from the environment. Every agent's observation is processed by a DME layer, which extracts a set of permutation invariant features. These are then concatenated with the information from the environment which did not belong to agent observations (if any) and connected with the remainder of the layers in the critic network.

Convolutional layer implementations can be shaped to act like DME layers. If each agent observation has a length of O, and the set of agent observations being fed to the centralized critic has a shape of $[J, O]$, a convolutional layer with a kernel of $[1, O, 1]$ dimensions, with stride 1 and depth f, will extract features from each agent observation into a vector of size f. The set of these J vectors can then be reduced with permutation invariant functions to create a single f-sized vector, which acts as a standard hidden layer in the network (with the exception of being permutation invariant to the set of agent observations). This behaves like the architecture shown in Figure 7.7, and it remains an end-to-end differentiable architecture, which can still be fully optimized through backpropagation. This technique is adaptable to higher dimensions, like two-dimensional pixel-based observations. Multiple convolutional layers with non-linear activation functions can be chained as desired, to extract non-linear features from each agent observation.

Agents' actor networks, when their inputs are correlated with their teammates and affected by their permutations, can also benefit from this architecture. For example, this can happen if agents receive broadcast messages from all their teammates, or if they observe all teammates' locations.

7.3 Evaluation

The A3C3 algorithm is evaluated in multiple environments and scenarios. Initially, a state of the art comparison is performed, and A3C3 is tested against independent A3C and DDPG agents without communication, an A3C3 ablation without a centralized critic, and communication-less multi-agent algorithms (MADDPG and A3C3 without communication). The benefits of communication are then analyzed, and how the amount of shared information impacts the performance of a team. The learned protocols are analyzed and their meaning is partly-inferred in the following section. Finally, the effects of noise in the communication

medium are tested against baselines with *no noise* or *no communication*. The advantages of permutation invariant network architectures are then evaluated, using swarm environments with large amounts of agents. Augmented critic-inputs are also tested with respect to how they affect the final policies. Finally, an analysis of different network architectures is conducted to demonstrate the robustness of A3C3 with different network shapes and sizes.

Some of the tests with three workers were conducted in a medium-range laptop without a GPU, demonstrating how the proposed algorithm can be deployed in commodity hardware and still achieve complex reward-based learning policies within a reasonable amount of time. The performance of algorithms is based on the average episodic reward obtained by the team, where better policies imply higher rewards. Accordingly, the plots shown represent the average reward obtained by the team. Learned policies have been published at `https://youtu.be/fB71yKcP3iU`.

The network architectures were determined with a grid parameter search, where configurations of one to three hidden layers with 10 to 120 nodes were tested, as well as ReLU, ELU and Sigmoid activation functions, which perform the best across environments and algorithms [213]. We chose one of the simplest architectures that consistently converged to proper policies.

Networks used the Glorot initializer [92] with default parameters to compute their initial weights, the Adam optimizer [96] with default parameters to optimize them, and an entropy weight $\beta = 0.01$ to discourage premature convergence. For each environment, we chose the highest learning rate η that still allowed convergence, and a future reward discount factor γ dependent on the horizon of each scenario and its importance of future rewards. Following reproducibility guidelines [213], our tests have been published along with our source-code at `https://github.com/david-simoes-93/A3C3`, and our parameters are described in Table 7.1.

7.3.1 State of the Art Comparison

This section focuses on evaluation the performance of learned policies in multiple environments. Teams are evaluated based on their average episodic reward, and algorithms are given similar training parameters for a fair evaluation. A3C3 is now compared against its possible ablations, as well A3C, DDPG, MADDPG, COMA, VDN and QMIX, all of which are described in Section 2.5. Tests are run in the POC and MPE suites, both described in Section 3.5.

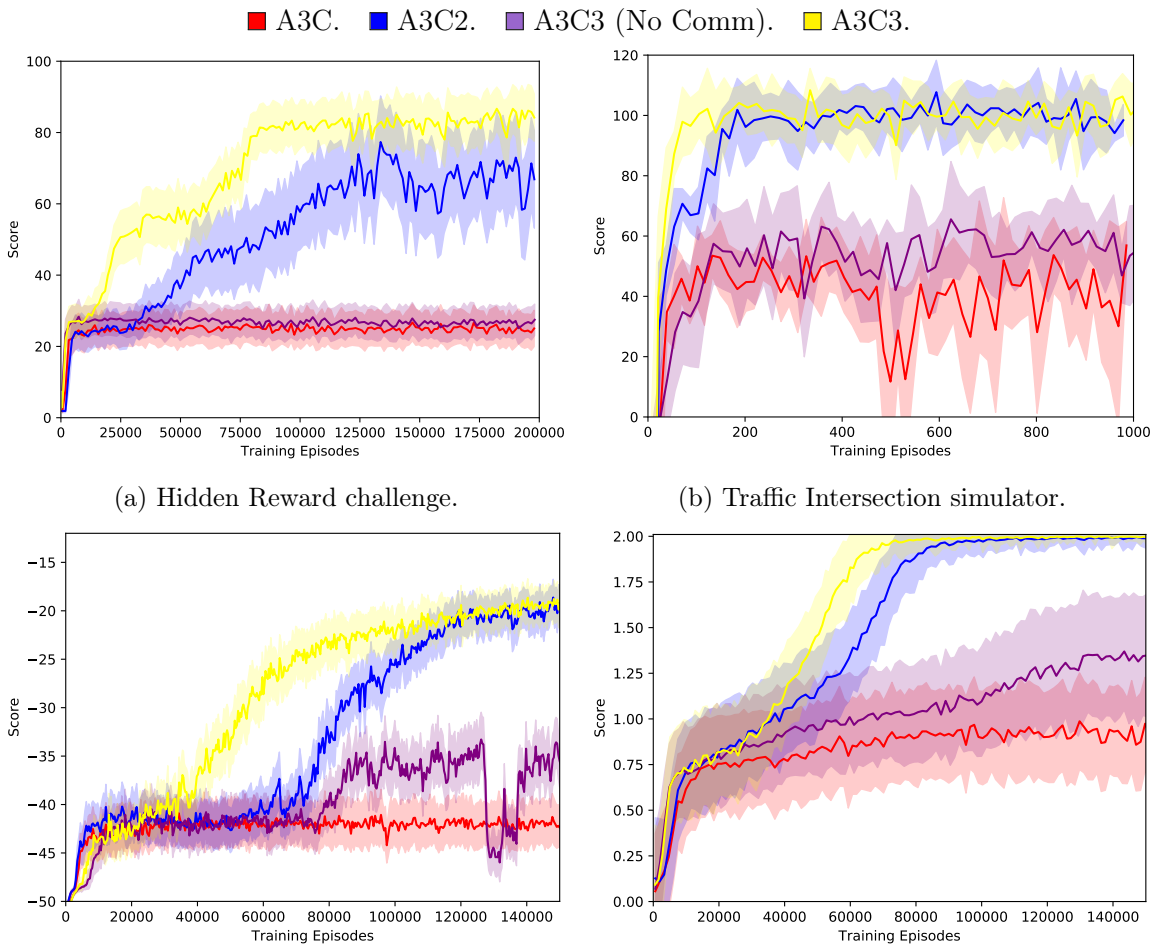
A3C3 can be directly compared with its ablations and with A3C, due to their similarities. The ablations consist on a variant of A3C3 without communication, where implicit coordination is learned but no information is shared, as well as a variant with communication but without a centralized critic, henceforth called A3C2 [62]. In this ablation, each agent's critic takes as input its own local partial observation to approximate its value function. The same hyper-parameters can be used across these tests, and learning curves are directly comparable. In contrast, DDPG and MADDPG cannot use the exact same hyper-parameters, as they are derived from different bases. Instead, the hyper-parameters proposed by Lowe et al. [23] are used, in their described set of environments, and the average performance of the best obtained policies (after these have converged) is used as a baseline against A3C3. Similarly, COMA, VDN, and QMIX also use the hyper-parameters described in the PyMARL suite [214], which is targeted at multi-agent StarCraft II. Environments were adapted to the requirements of each implementation, by providing team-wide rewards in all environments and providing empty-states for vehicles in the Traffic Simulator that are not active.

Environment	J	N	γ	η	CC	x	Episodes
POC Hidden Reward	4	3	0.95	10^{-4}	20	2	2×10^5
POC Traffic Simulator	40	3	0.1	10^{-4}	1	2	1×10^3
POC Pursuit	3	12	0.95	5×10^{-5}	10	6	1.5×10^5
POC Navigation	2	3	0.95	10^{-4}	20	4	1.5×10^5
MPE Coop Navigation	3	3	0.001	10^{-4}	10	8	2.5×10^4
MPE Coop Communication	2	3	0.001	10^{-4}	10	8	2.5×10^4
MPE Coop Reference	2	3	0.001	10^{-4}	10	8	2.5×10^4
MPE Tag	3	12	0.95	10^{-4}	10	8	5×10^5
KiloBots Light	17	12	0.95	10^{-4}	2	4	3×10^5
KiloBots Join	17	12	0.95	10^{-4}	2	4	4.5×10^5
KiloBots Split	17	12	0.95	10^{-4}	2	4	4×10^5
3dSSL Passing	3	6	0.9	10^{-3}	0	4	1.6×10^3
3dSSL Keep-Away	3	6	0.9	10^{-3}	0	4	6×10^3

Table 7.1: The hyper-parameters used for the tests conducted in this section, for all environments. This table lists the amount of agents J, the amount of workers N, the future reward discount γ, the learning rate η, the amount of communication channels (CC), and the layer size modifier x. Critic and actor networks used two fully connected hidden layers of $20x$ and $10x$ nodes activated with a ReLU function. The communication network used a single hidden layer with $10x$ nodes activated with a ReLU function, and an output layer of CC nodes, activated with a hyperbolic tangent function. The non-received message rc_{initial} default value is all zeros.

The results of A3C3 and its ablations in the POC suite, shown in Figures 7.8, demonstrate that A3C3 outperforms all other options. The algorithm achieves stronger policies within less time-steps and with less variance. Without communication, both A3C3 and A3C fail to complete tasks successfully. Without the centralized critic, the learning process takes longer and policies have much higher variance.

A3C3 is also compared with COMA, VDN, QMIX and CommNet in the POC suite, as shown in Table 7.2. COMA, VDN, and QMIX rely on centralized critics for coordination, and feature no communication. To compensate for that lack of information sharing, the networks use a recurrent layer such that agents can remember information from their multiple partial observations of the environment, and their centralized critic, like A3C3, has access to the underlying environment state. However, COMA, VDN and QMIX cannot achieve successful results in any of the environments. They match the performance of independent A3C in the cooperative tasks that can be partially completed without communication. In the Traffic Intersection simulator, the amount of agents impacts the performance of algorithms and they are unable to learn an adequate policy. In the Pursuit game, QMIX and COMA agents cannot overcome the partial-observability of the environment and the prey constantly elude them. Predators are only able to learn not to collide with each other. A3C3 clearly outperforms other state-of-the-art non-communication algorithms in all these environments. CommNet, on the other hand, does feature communication between agents but outputs a joint-action for the entire team. It was tested in the Hidden Reward and Navigation environments, using both feed-forward architectures and LSTM architectures. Despite outperforming other



(a) Hidden Reward challenge.

(b) Traffic Intersection simulator.

(c) Pursuit game.

(d) Navigation task.

Figure 7.8: Results of A3C3 for the POC suite. The plots represent the average reward and standard deviation (over N workers) obtained by A3C, A3C2, and A3C3 (with and without communication), over training episodes.

---

options, aside from A3C2 and A3C3, in the Navigation environment, it was unable to achieve successful policies in either.

A3C3 either outperformed or matched our heuristic baselines. These baselines were achieved with a hard-coded centralized controller that output a joint-action for the entire team, with access to all agent observations. In the Traffic Simulator and Navigation environments, which require little map exploration, A3C3's behavior closely follows the baseline. In the Hidden Reward and Pursuit environments, A3C3 optimized a better map exploration policy than the baseline, and completed the challenges faster, thus achieving a higher score.

Tests are now conducted on A3C, A3C3, DDPG, and MADDPG, on the MPE suite. While this section summarizes each environment, the reader is referred to Lowe et al. [23] for further information. All environments except the Tag challenge provide instant progressive rewards, so a future reward discount factor $\gamma = 0.001$ is used in these for simplicity. MADDPG agents integrate communication as an additional action-space with a predetermined vocabulary in these environments.

	A3C	A3C2	A3C3	Heur.	COMA	PPO	QMIX	VDN	CommNet
POC H.R.	24	67	**84**	71	14	27	19	19	21
POC T.S.	-93	-16	-14	**-13**	-257	-122	-255	-259	-
POC Pursuit	-42	-19	**-19**	-27	-50	-50	-50	-44	-
POC Nav.	0.94	1.97	1.99	**2**	0.68	0.93	0.38	0.92	1.22

Table 7.2: Comparison of multiple algorithms and a heuristic baseline for the tested environments. The average reward (over 100 test runs) obtained by the team after each algorithm trained for the amount of episodes shown in Table 7.1. Each algorithm tries to maximize the obtained reward, and the best results are shown in bold. A3C3 is able to match or surpass all other algorithms in all environments.

	Cooperative Navigation		Cooperative Communication	
	Average Distance	# Collisions	Average Distance	Target Reached
A3C2	0.219	1.223	0.007	99.6%
A3C3	**0.162**	**1.245**	**0.006**	**99.9%**
MADDPG	1.767	0.209	0.133	84.0%
DDPG	1.858	0.375	0.456	32.0%

Table 7.3: Results of algorithms for Cooperative Navigation and Cooperative Communication environments.

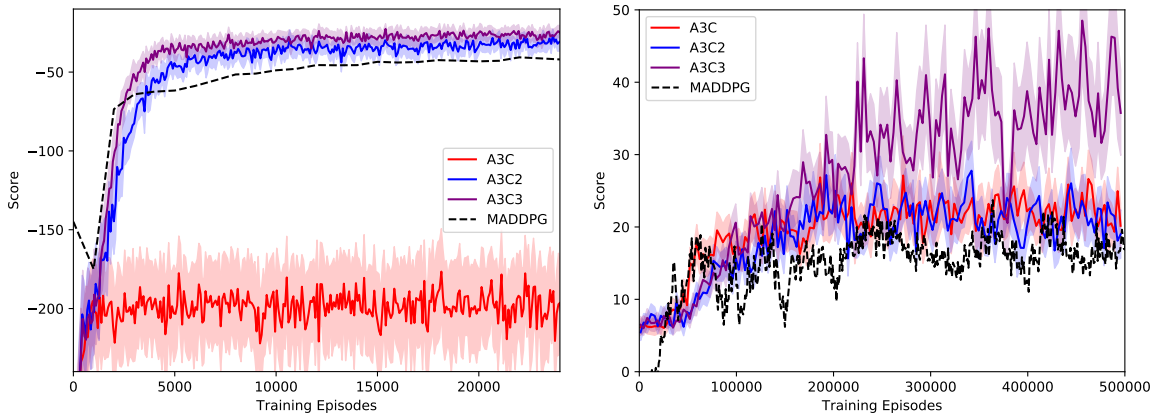
Initially, the performance of policies in two cooperative environments with communication is evaluated: the Cooperative Communication environment, where one agent behaves as a speaker, and informs a listener agent of which of three targets is his; and a Cooperative Navigation environment, with three agents, where these need to cover all the targets available. The performance of teams on the latter environment is given by $r = \sum_l^L -d_l - C$, where L is the total amount of landmarks, d_l is the minimum distance of each landmark l to any agent, and C is the amount of collisions on the environment. The results are shown in Table 7.3. In the Cooperative Navigation environment, A3C3 achieves smaller distances, but more collisions, maximizing the rewards obtained by agents. In Cooperative Communication, A3C3 can learn policies that are more frequently successful and also achieve shorter distances to the target positions than DDPG and MADDPG. Therefore, for both these environments, A3C3 outperforms MADDPG and DDPG in terms of average episode reward.

Algorithms are now tested in the Cooperative Reference and Tag tasks from MPE, and evaluated based on their average obtained reward. Since Tag has competing teams, the opponents used the same static policy previously learned with MADDPG in self-play.

In Cooperative Reference, agents know each other's target landmarks and must communicate this information to each other in order to find their own targets. The Tag challenge is similar to the Pursuit environment, but the observation space is continuous and map vision is global. Predators learned to catch a prey, pre-trained with MADDPG, and able to move at twice the predators' speed.

The results, shown in Figure 7.9, demonstrate the evolution of policies learned by A3C and A3C3 against MADDPG baselines, trained under the same conditions until convergence was achieved. A3C3 agents completed both the proposed tasks and greatly outperformed MADDPG, while A3C was unable to do so in Cooperative Reference, which required commu-

nication.



(a) Cooperative Reference.

(b) Tag Challenge.

Figure 7.9: Results of A3C, A3C3, and MADDPG for the tested environments. The colored plots represent the average reward and standard deviation (over N workers) obtained by A3C and A3C3 agents, and the dashed plot represent MADDPG agents' average reward, over training episodes.

Finally, A3C3, A3C and MADDPG teams are also evaluated after learning policies through self-play. In this case, the Tag task is used as a competitive environment, and each algorithm trains both teams until policies stabilize. The actual average reward obtained by the predatorial team for each algorithm has no meaning, since its performance depends on its opponent's strategy. Weak predators may obtain higher rewards against weak prey than optimal predators against strong prey. Table 7.4 shows an analysis of policies learned by A3C3 and A3C against those learned by MADDPG. Surprisingly, A3C has the best predators against MADDPG, while A3C3 has the best prey. A deeper analysis revealed this is due to overfit policies learned by A3C3, which made A3C3 predators less flexible to different opponents.

A more intuitive way of analyzing each algorithm's performance is to compare the scores of each algorithm when the teams are reversed. Predators try to maximize their own score, catching the prey as often as possible yields points. Prey try to minimize the predators' score, by not getting caught, awarding no points to predators. An algorithm that achieved perfect policies would earn maximum points when playing as Predators, and zero points when playing as Prey. Calculating the difference of points by the Predator team, for A3C and A3C3 against MADDPG, shows that A3C3 achieves the highest difference, 15 points, and therefore outperforms A3C.

7.3.2 Effects of Communication

Figure 7.10 shows the policy evolution of A3C3 across training episodes of the team, applied to the proposed suite of environments. Different amounts of communication channels (CC) are tested, demonstrating how explicitly sharing information with learned protocols can also improve the agent policies. The CC represent the width of the communication network's output layer and the length of sent messages. The communication network's output layer is activated with a hyperbolic tangent function, such that each CC outputs a continuous value $[-1, 1]$.

	A3C		A3C3	
	Predator	Prey	Predator	Prey
MADDPG	**144.66**	141.37	123.27	**108.06**
Score Difference	3		**15**	

Table 7.4: The average score of predators in the Tag environment when pitting teams trained with different algorithms against a team trained with MADDPG. Algorithms try to maximize their score when acting as the Predator team, and minimize their score when acting as Prey.

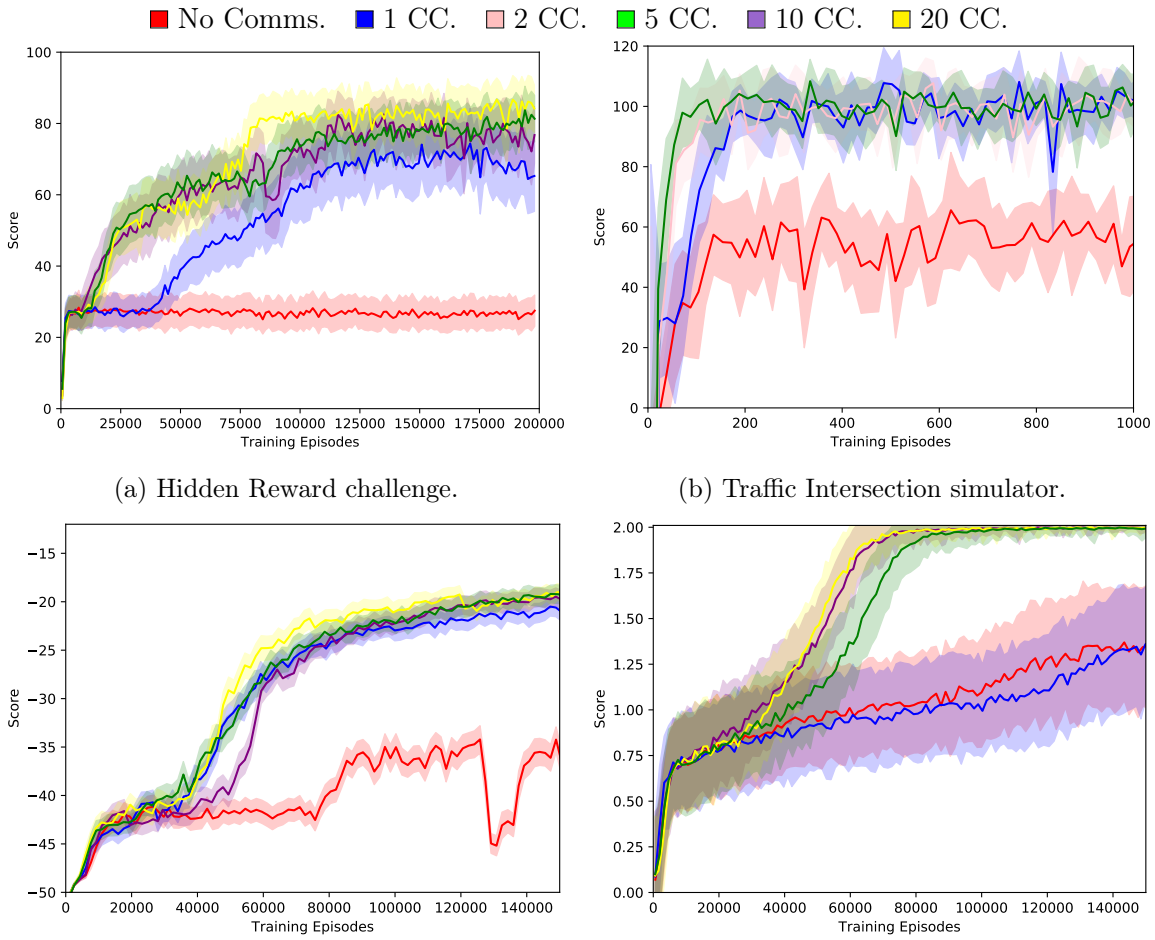
For the Hidden Reward challenge, shown in Figure 7.10(a), without communication, the average reward stagnates around 30, since agents cannot share the reward zone's position, and thus resort to random exploration to find it. However, with a single CC, a better policy can immediately be found. This continuous channel helps reduce the area of exploration, and allows the policy to greatly improve. With multiple channels, agents obtain an average reward of 85, and as we increase the amount of channels up to twenty, the speed at which the policy is found also increases. Agents learn to coordinately explore, and to alert team-mates when the reward has been found, which lets the team converge on it.

In Figure 7.10(b), communication is crucial to achieve a decent policy in the Traffic Simulator, where traffic flows easily and without collisions. We estimate the quality of traffic flow by the amount of intersection collisions, where two vehicles intend to follow the same path try to cross the intersection simultaneously, and the amount of intersections stops, where a vehicle at the intersection does not move (possibly to avoid a collision). With no communication, the total average reward for all agents is 60, since agents cannot communicate whether they have to turn or not, which leads to large traffic jams, and agents randomly colliding, with an average of 4 collisions and 95 stops per episode. A single CC is sufficient to achieve a policy where vehicles adhere to traffic rules and agree on who should advance at intersections, achieving a total average reward of 100, an average of 0.5 collisions, and 15 stops per episode. Interestingly, agent populations learn different communication protocols, signaling either their intent to turn or to move forward, depending on the randomly initialized weights.

Regarding the Pursuit game, Figure 7.10(c) shows how the lack of communication leads to underperforming and unstable policies, which obtain an average reward of -35. Analysis of the policy's behavior and of the average time taken to complete an episode shows the majority of time-steps are wasted while a single predator chases a prey, until another predator randomly crosses its path. With at least one CC, predators can now signal that a prey has been spotted, and they can converge on it. Through communication, A3C3 predators coordinate to obtain an average reward of -20, and converge to more stable policies.

In the Navigation task, Figure 7.10(d) shows that, without communication, no suitable policy is found, since agents cannot share their position, resort to pure chance to cover all the beacons, and obtain an average reward of 1.25. Agents try to cover both beacons simultaneously and end up failing the task. However, communication allows agents to coordinate with one another, and while a single CC does not allow enough information to be conveyed, multiple channels lead to optimal policies with an average reward of 2 (the maximum possible reward obtainable). Agents learn to assign tasks to one another, and distribute themselves accordingly.

For all the proposed environments, A3C3's communication techniques allow a team of



(a) Hidden Reward challenge.

(b) Traffic Intersection simulator.

(c) Pursuit Game.

(d) Navigation Environment.

Figure 7.10: Results of A3C3 with different CC for the tested environments. The plots represent the average reward and standard deviation (over N workers) obtained by A3C3 with 0 to 20 CC, over training episodes.

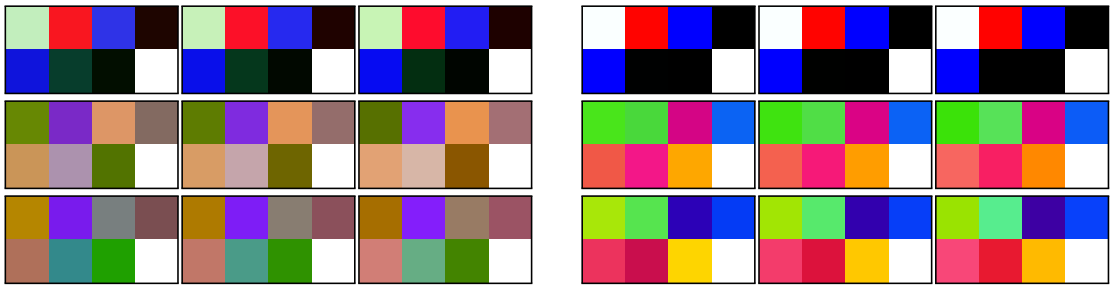
agents to complete them. Policies and communication protocols are learned simultaneously, *tabula rasa*, and demonstrate that information sharing is a tool that can help compensate for local partial-observability of the environment. As more information is transmitted (by increasing the amount of CC), the team's performance increases until the transmitted information is sufficient to achieve successful policies. The learned communication protocols are not easily translated into human-readable protocols. The simpler ones can be easily deduced (like transmitting whether a vehicle intends to turn or not), but with over five channels, most protocols require a non-trivial analysis.

7.3.3 Communication Protocols

This section analyses the learned communication protocols in the partially-observable suite. While the protocol may not be fully understood by humans, it is possible to extract some interesting conclusions on what agents are actually communicating and how those messages

can be interpreted by teammates.

For easier analysis, we convert the protocol channels into colors (each channel representing one of three RGB values composing a pixel). For example, a 20-channel protocol can be represented in seven pixels, the last of which has no *blue* component.



(a) Reward zone not found. (b) Reward zone found.

Figure 7.11: Color-coded 20-channel communication protocol learned in the Hidden Reward environment. Messages are arranged geographically, representing an agent's output message in the edges and center of the map. Figures exemplify messages sent when (a) the Reward zone has not yet been found and (b) the agent has found it and is signaling its position.

Figure 7.11 shows a protocol learned in the Hidden Reward environment. Nine messages are shown and arranged geographically, representing the message an agent outputs in a corresponding location on the map. For instance, the center message is the message sent by an agent to its teammates when at the center of the map. The policy learned by the team consisted on agents forming a vertical line and moving across the X axis to explore the map. The learned communication policy facilitates this by having distinct patterns across the Y axis, so that agents can clearly understand which rows are being explored. The differences on the X axis are more subtle, but are noticeable enough that agents can align in the proper formation during exploration. Figure 7.11 also shows a clear difference between the exploration protocol (a) and the exploitation protocol (b). This difference represents the alert signal agents emit when the reward zone has been found. When receiving an alert message, agents know to converge to the sent coded-coordinates, instead of maintaining their exploration protocol.

Figure 7.12 shows the evolution of two protocols learned in the Traffic environment. Both populations learn to clearly distinguish the intention of turning or going forward. Interestingly, different populations learn opposite protocols, where (a) agents signal their intention of going forward, and (b) agents signal their intention of turning.

Figure 7.13 shows a protocol learned in the Predator/Prey environment. Messages are shown and arranged geographically, representing the message an agent outputs when a prey is found at a corresponding location in its local observation, while the agent is at the center of the map. The center position is where the predator, and no prey can be there (it would be captured instead). The policy learned by the team shows distinct patterns for each location, which alerts other predators to the prey's location. At this point, predators then converge and surround the prey until it is captured. This effectively allows agents to handle the partial-observability of the environment.

Figure 7.14 showcases, for the Navigation environment, the difference between the initial random protocol that agents use, and the final protocol that is used after policies converge.

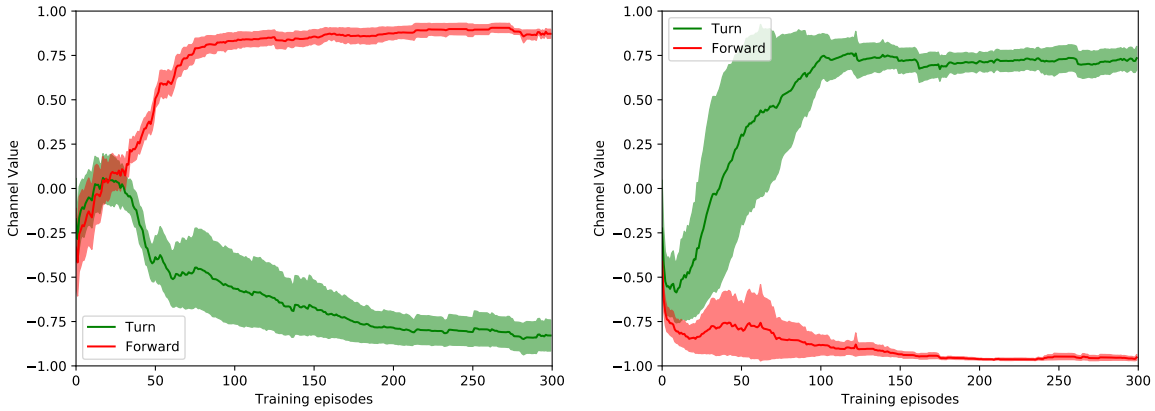


Figure 7.12: The evolution of two separate 1-channel communication protocols learned in the Traffic Simulator environment, using two separate agent populations. The plots represent the output message's single channel value across training episodes, when two agents stand at an intersection. The values are averaged over possible intersection situations (vehicles behind or in front of the agent), and split based on whether the agent intends to turn or move forward.

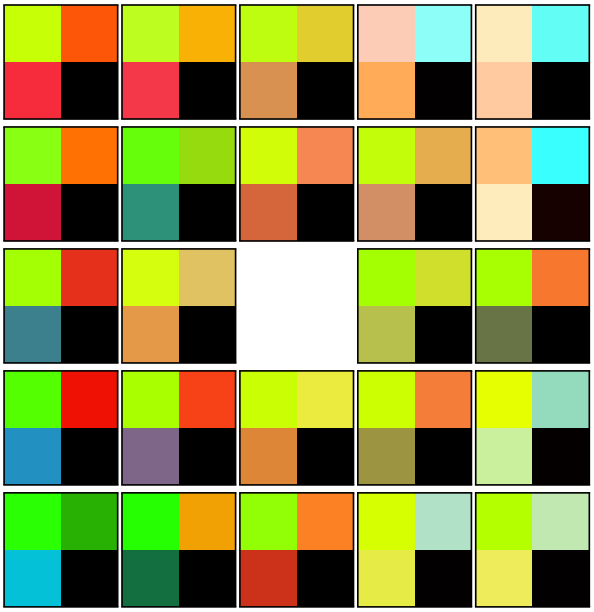
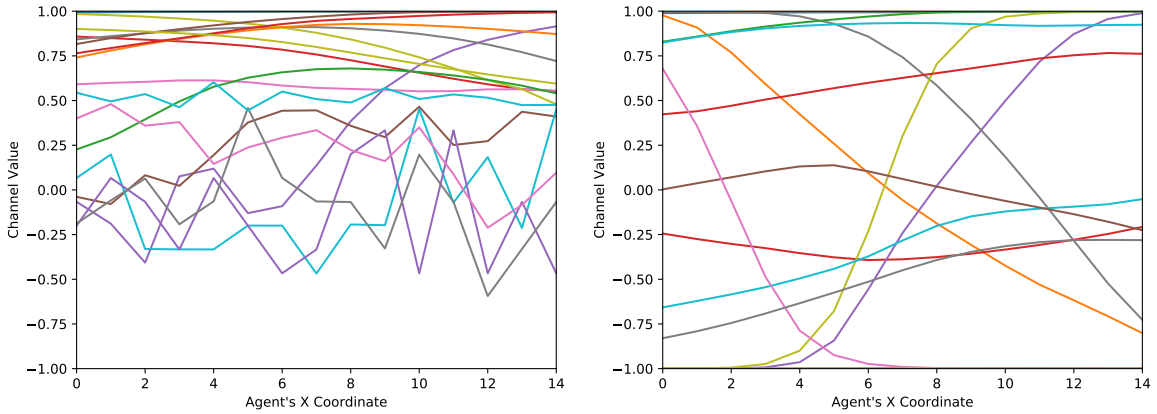


Figure 7.13: Color-coded 10-channel communication protocol learned in the Predator/Prey environment. Messages are arranged geographically, representing an agent's output message when a prey is found in the corresponding location of its local observation.

Initially, each channel behaves without much correlation regarding the agent's location. In other words, agents cannot properly decide which agent covers which beacon, as they cannot interpret the other agents' locations from their messages. However, the protocol evolves in a way that agent coordinates can be extracted from their sent messages. When convergence is



(a) Random initial protocol.

(b) Learned protocol.

Figure 7.14: The 20-channel communication protocol in the Navigation environment, (a) before and (b) after convergence has been achieved. The plots represent the average value of each channel in a message as the agent's X coordinate changes.

achieved, some channels have very obvious correlations with the X coordinate of the agents' locations (and a similar behavior is seen for the Y coordinate). This in turn allows agents to cooperate and coordinate their coverage of the beacons.

7.3.4 Communication Noise

This section analyses the effects of induced noise in the messages exchanged by agents, against a noiseless communication baseline, and a policy with no communication at all. Three major types of noise are considered:

- Loss - Covering messages that are lost, sent messages have a chance P_{loss} of not being delivered. This also covers delays in messages, where a delayed message is considered to be lost.

- Noise - Covering external interference in received messages, as these are continuous-valued vectors. Gaussian noise $\mathcal{N}(0, V_{\text{noise}})$ is added to these values.

- Jumble - Covering internal interference in messages, received messages have a chance P_{jumble} of having been mixed with others. Instead of receiving the original message, that message is instead a sum of all received messages, and there is no indication to the agent whether the message has been jumbled or not.

Each effect is tested individually, as well as all three simultaneously (shown as "All").

Figure 7.15 shows the effects of disturbing the communication channels of the teams. For all environments, communication can be robust to both noisy interference and jumbled messages, as they allow some non-negligible form of coordination. Message loss has the greatest impact on the performance of the team, as it necessarily approximates the team's performance to one without communication. In the Traffic Simulator, for example, losing messages at intersections forces agents to wait for that turn even if they have priority.

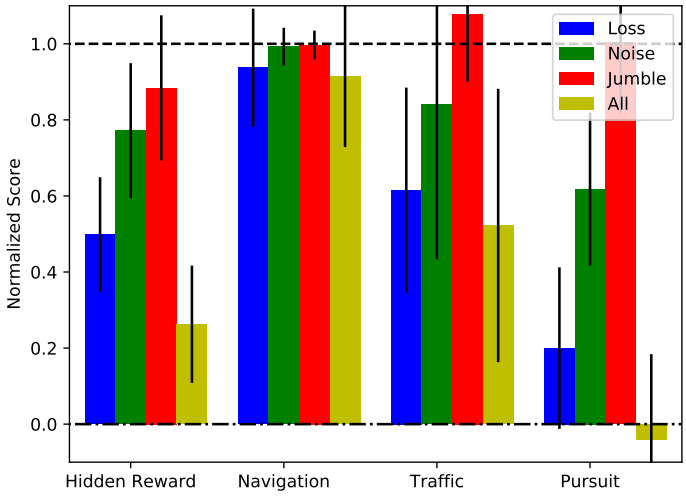


Figure 7.15: Results of the effects of noise for multiple environments. The plots represent the average reward and standard deviation (over N workers) obtained by agents at the end of the training phase, normalized between the average reward with no noise and the average reward with no communication.

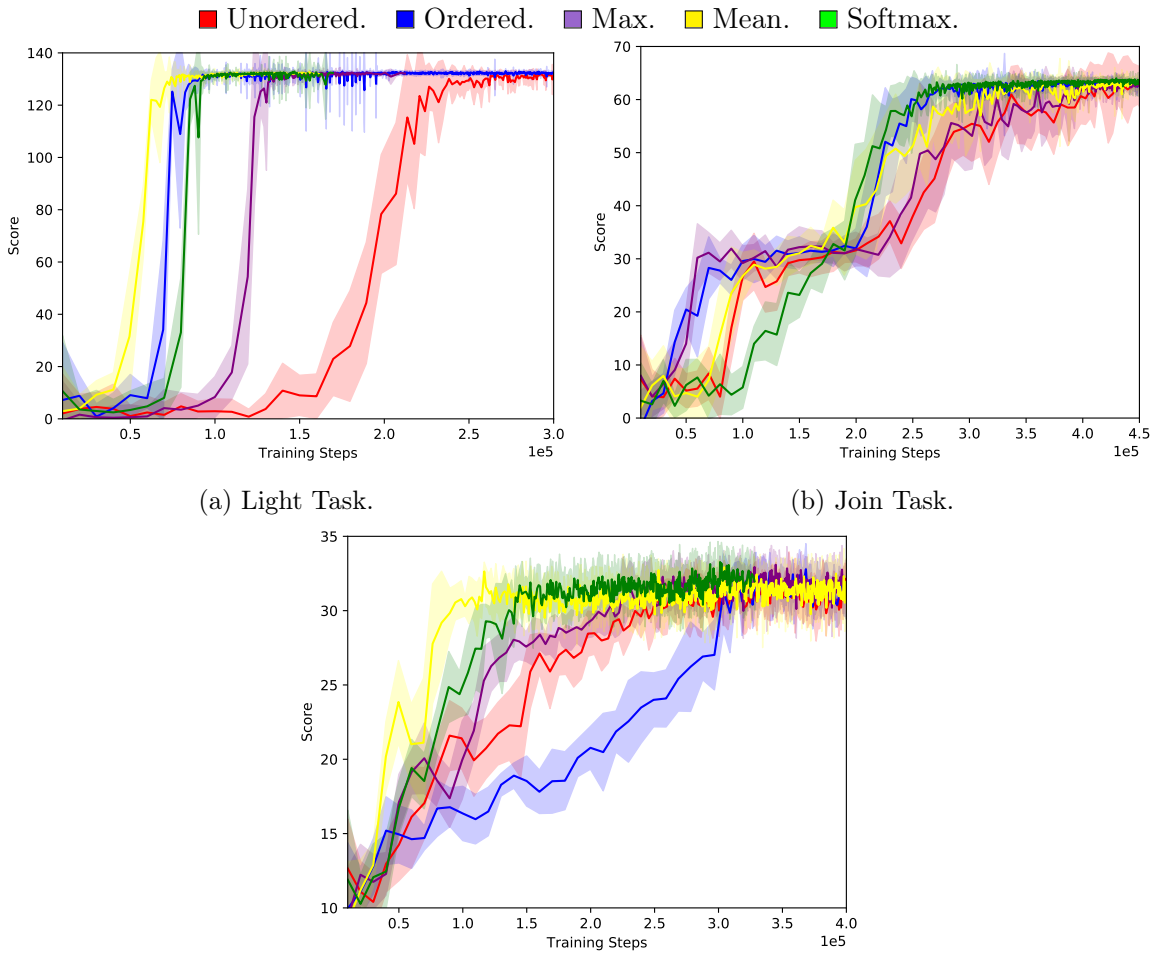
In general, noise makes policies slower to learn, and decreases their overall performance. However, even with all noise types enabled, cooperation is still achieved in all environments, and results are better than policies with no communication at all.

7.3.5 Swarms and Permutation Invariance

This section describes the tests conducted to evaluate the scalability of A3C3 with large teams in the KiloBots environment, described in Section 3.4.3. The continuous action-space of the KiloBots is discretized into a simple set of actions, including rotation, stopping and moving forward. Regarding communication, only two CC are used, and messages are not broadcast to the entire team, but instead sent to the two closest agents.

The performance of teams is evaluated by comparing different methods to handle the large amount of agents. Tests include an *unordered* method, with the standard fully connected A3C3 architecture used on other environments; an *ordered* method, where agent observations are ordered by an average of their X and Y coordinates; the *mean*, *max*, and *softmax* DME methods for permutation invariance through convolutional architectures.

Figure 7.16 shows the evaluation conducted on all five architectures and on all three scenarios. The analysis of the performance of the teams shows that all architectures can converge to successful solutions. In other words, even with a large amount of states, the centralized critic can learn an accurate value estimation for the team. Pre-processing and ordering the input helped in two of the three scenarios, which further demonstrates that it is not a generally applicable solution. However, the MDE methods accelerated the learning phase by a large amount in all scenarios, and shortened the time taken for the teams to achieve optimal policies. The *mean* MDE shows the best overall results, being the fastest in most cases and simpler than the *softmax* MDE.



(a) Light Task.

(b) Join Task.

(c) Split Task.

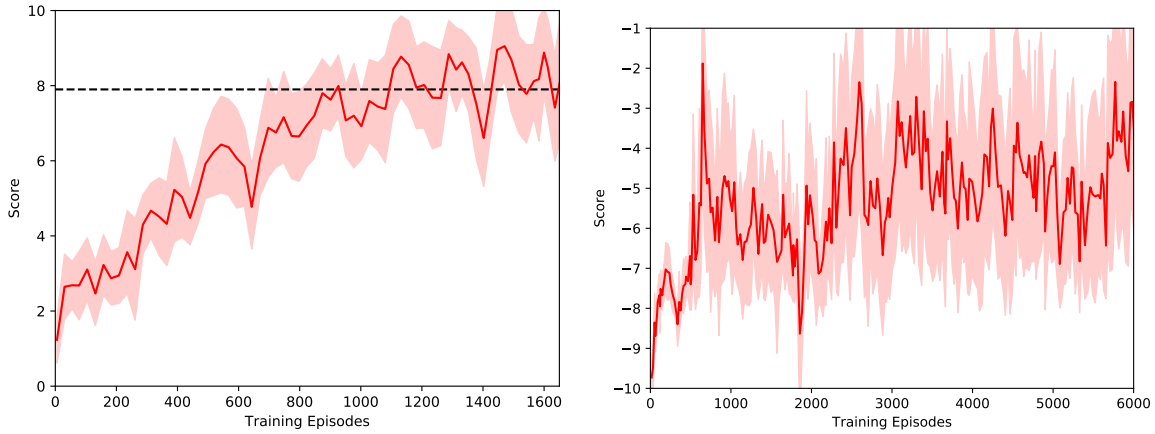
Figure 7.16: The evolution of policies in the three tasks of the KiloBots environment. The plots represent the average reward and standard deviation (over N workers) obtained by the team across training steps, using different architectures to handle permutations in the Centralized Critic.

7.3.6 High-Level Strategy Learning

This section describes the tests conducted in the 3dSSL framework, described in Section 3.5.3, using the 3D Soccer Simulator with the FCPortugal3D team. A sub-set of FCPortugal3D's behaviors was used, such that A3C3 learns a high-level policy that takes advantage of the low-level behaviors already existing in the team. A learning layer was implemented on top of the FCPortugal3D agent such that behaviors executed were allowed to complete, and continuous behaviors (like *standing*) would be executed for multiple time-steps before a new behavior could be chosen. This follows the frame-skipping method [59], where the same action is repeated multiple times, to speed up the learning phase. Because communication is heavily restricted in the 3D Soccer Simulator, and FCPortugal3D agents already have hard-coded communication protocols [108], no CC were used.

The team's performance is shown in Figure 7.17, where agents learn effective strategies

to complete each scenario. In the *Passing* challenge, agents group together such that passes are quicker and more accurate. On the *Keep-Away* scenario, agents do the opposite, and remain in corners of the field, such that the opponent takes a long time to reach the ball. They successfully keep the ball away from the opposite team until the episode reaches a given time-limit. When compared with the hard-coded policies used by the team in competition, the learned policies matched the performance of the *Passing* scenario, but did not outperform FCPortugal3D's *Keep-Away* behavior.



(a) *Passing* scenario.

(b) *Keep-Away* scenario.

Figure 7.17: The evolution of A3C3 policies in two tasks of the 3D Soccer Simulation environment. The plots represent the average reward and standard deviation (over N workers) obtained by the team across episodes.

Interestingly, agents learn to take advantage of the implementation details of the scenarios, and abused them. In the *Passing* challenge, agents converged in the center of the map and learned to alternately move towards and away from the ball, which would make the environment score a successful pass (as a different agent was now closer to the ball). In the *Keep-Away* challenge, agents discovered that the opponent would not move beyond the field lines, so they converged to a policy where they just kicked the ball outside the field and would no longer need to move. Both scenarios were fixed with regards to these exploitations.

7.3.7 Augmenting Centralized-Critic Inputs

The effects of different inputs in the centralized critic are also evaluated. Along with the aggregate of agents' local observations, the actions of other agents and/or the messages received by the current agent in this cycle are also concatenated. However, the analysis of the policy evolutions shows that the different critic inputs don't have much impact on the learned policies. The most noticeable negative impact appears in the Traffic Intersection environment, where actions of all agents are now included in the critic network, despite having no impact on agents not on the same intersection.

In general, introducing agent actions or messages brings no major advantage to the A3C3 algorithm. The added complexity of the network offsets the stability or simplicity of the value function estimation, and causes the learning process to take longer.

7.3.8 Architecture Variance

To test the robustness of A3C3 with respect to its network architecture, grid parameter search was conducted, and the performance and learning time for multiple configurations was evaluated. Configurations included one to three fully connected hidden layers, whose sizes ranged from 10 to 20 nodes. These included a single hidden layer with 10 nodes, two hidden layers with 20 and 10 nodes, and three hidden layers with 20, 10, and 10 nodes. These layer sizes were then multiplied by multiple network layer size multipliers x, ranging from one to six. At the ends of the spectrum, the simplest network had a single hidden layer of 10 nodes, and the most complex had three layers of 120, 60 and 60 nodes, respectively. Orthogonally, the ReLU, ELU and Sigmoid activation functions were also evaluated.

These configurations were used in the actor and centralized critic networks, while the communication network had a single hidden layer with the same size as the smallest layer in each configuration. All hyper-parameters apart from the network architecture remained the same in these tests, and the results for the Navigation environment are shown in Figure 7.18.

	ReLU				ELU				Sigmoid			
	1	2	4	6	1	2	4	6	1	2	4	6
10	0.32	0.49	1	1	0.55	0.72	0.95	1	0.49	0.71	0.85	1
20,10	0.81	1	1	1	1	1	1	1	0.95	1	1	1
20,10,10	0.61	1	1	1	1	1	1	1	1	1	1	1

(a) The average normalized reward $r \in [0, 1]$ obtained by agents at the end of 150 thousand training episodes.

	ReLU				ELU				Sigmoid			
	1	2	4	6	1	2	4	6	1	2	4	6
10			116	111				140				114
20,10		71	72	47		79	52	49		69	47	38
20,10,10		60	59	32	124	53	34	31	131	80	46	34

(b) The thousands of training episodes $t \in [0, 150]$ required to find the optimal strategy (if ever).

Figure 7.18: The grid parameter search for adequate network architectures. Rows represent hidden layer configurations (e.g., bottom row represents a network with three hidden layers), whose sizes are multiplied by a network layer size multiplier x in each column (e.g., far right column represents multiplying hidden layer sizes by a factor of six).

Results show that A3C3 is fairly robust to various network architectures, from the point where they are complex enough to successfully approximate the target functions. The simpler networks with a single hidden layer were unable to converge and successfully complete the task. Analogously, the most complex networks are successful and feature the fastest convergence to optimal policies.

7.4 Conclusion

This chapter described A3C3, a multi-agent deep reward-based learning algorithm, where distributed worker threads use Actor-Critic methods to optimize value, policy and communication networks for agents. The algorithm features a centralized learning phase, distributed

execution, and inter-agent communication. A3C3 supports partially observable domains, noisy communications, heterogeneous reward functions, distributed independent execution, and a variable amount of agents. It can be horizontally scaled, and supports inter- and intra-agent parameter sharing, techniques which increase the convergence speed of policies.

A3C3 works by implicitly sharing information during a centralized learning stage, through the Centralized Critic network, which improves the convergence of other networks and increases the performance of learned strategies. It also explicitly shares relevant information, through the Communication network, which is optimized based on the performance of other agents. In other words, the network is optimized such that other agents perform better, thus enforcing coordination between agents. Even with noise and partial observability, agents can learn successful policies and communication protocols *tabula rasa*. The shared information may not be human-readable, as it is represented by vectors of continuous-valued messages. However, agents use them to share local information, alert other, assign targets, and coordinate exploration. Logically, agents from different populations cannot communicate with each other, as different communication protocols will likely have been learned.

A3C3 is formally described and its behavior is shown in multiple multi-agent domains. It is compared against other multi-agent algorithms, exceeding their performance, as well as against independent single-agent implementations. The effects of communication noise, critic-augmentation, and permutation invariant architectures, are also analyzed. A3C3 can be framed as a more general version of previous works, and its source-code is publicly available.

Despite its generality, A3C3 carries multiple assumptions. Environments are expected to be cooperative, with a discrete action space, and agents communicate in order to improve each other's policies. Each worker thread must also act as a centralized learning environment, with access to all agents and communications. Ideally, if the environment's state can be sampled, A3C3 can benefit by approximate a more accurate value function. The communication protocols learned by agents are also difficult to interpret and translate to human-readable information. While agents can be robust to random noise, messages can be externally interfered with to disrupt the team's behavior. A major drawback of A3C3 is that actor-critic algorithms are on-policy and sample inefficient. While the algorithm scales horizontally in simulated environments to alleviate this problem, learning in real-world robotic tasks will likely require an initial simulated phase to achieve an adequate initial solution before deploying policies on robotic agents.

A3C3 can be improved with two main concepts. The first is to describe a new loss function for the communication networks for non-cooperative environments. In a fully competitive environment, agents would attempt to communicate in such a way as to minimize the opponent's rewards, while maximizing their own. The second is to integrate the Value-Decomposition Network used by QMIX, as well as COMA's credit assignment strategy, to further improve the centralized critic's value function approximation for the team.

Future work also includes testing A3C3 with recursive neural networks, like RNN or LSTM, such that partially-observable environments can be further exploited. With feedforward networks, agents can only share current information, but recursive architectures would allow past observations to be exploited as well.

Chapter 8

Conclusion

The field of MARL has witnessed a large growth in recent years, with many novel algorithms and techniques surfacing to tackle its challenges. This has been partly due to the re-emergence of deep learning as a solution to handling complex environments, which in turn increases the applications and motivations of developing general MARL algorithms. During the writing of this thesis, many algorithms were proposed as state-of-the-art novel contributions to the field, and we expect this tendency to remain. Such algorithms rely on techniques used in the A3C3 algorithm, like actor-critic, asynchronous deep learning, centralized critics, differentiable communication, among others.

However, rapid development of a field also has a few drawbacks. Firstly, it is harder for researchers to keep up-to-date with all the new contributions, and integrate them on their work. Practices like sharing source-code and formal algorithm descriptions alleviate the situation, but are often insufficient. Secondly, it is harder to distinguish which proposals have benefits over others and in which cases, as techniques are often unfairly evaluated and compared with others, and other researchers have limited time to corroborate the results being demonstrated. The problem worsens with deep learning algorithms that often require hours of computation time to achieve their results. The results showed in this thesis took over a year of computation time, not including time spent with tests, bugs, or other problems. That being said, we have described valuable contributions to the body of knowledge of multi-agent systems and machine learning.

In this thesis, two chapters focused on competitive environments. The WPL algorithm was extended with a new update rule for environments with deterministic NE strategies. Its behavior was kept in other situations, and its performance drastically increased in these. However, tabular algorithms are not adequate for complex competitive environments, like Starcraft 2 or Pokémon, given their complex state-space. As a solution, WPL and three other algorithms were extended to the deep learning paradigm and evaluated both in the traditional game-theoretic environments and in multi-state games with noisy or partial observations. Of all the tested algorithms, GIGAθ and WPLθ showed the best results, with GIGAθ achieving stronger deterministic strategies and WPLθ seeming more robust to hyper-parameters. However, both algorithms fall under the **Forget** category, continuously adapting to the non-stationary behavior of the opponent, and converge to the NE solution, which is not always Pareto optimal. For example, in Prisoner's Dilemma, the Pareto optimal solution is not a NE, but both agents achieve overall higher rewards. Solutions like recursive neural networks and opponent modeling techniques [215, 55] may increase GIGAθ and WPLθ's performance,

allowing agents to remember opponent strategies and adapt accordingly.

Two other chapters in this thesis focused on cooperative environments. The Double DQN algorithm was applied on the multi-agent paradigm, using the IL and JAL approach. Not only is JAL a more restrictive approach, but results also show that it does not scale or generalize as well as the IL approach. We showed how Independent MADDQN can achieve coordination in cooperative environments, and generalize to harder tasks with larger teams. However, MADDQN is a greedy deterministic algorithm in the **Ignore** category, which uses a replay memory with possibly outdated samples, and where agents do not exchange information. Recent solutions [158] to improve the replay memory may increase its performance. The A3C3 algorithm instead uses asynchronous updates to avoid the replay memory requirement. A centralized critic allows policies to robustly converge with implicit coordination, and agents communicate relevant information between them. This helps with partial-observability, as local information can now be shared between the team. The algorithm only requires a centralized learning stage, and otherwise assumes independent agents with local partial observations and possibly noisy communications. The algorithm is shown to scale to large team sizes through a permutation invariant network architecture. It can benefit from critic augmentation techniques [99] as well as structural credit assignment solutions [131].

The combination of cooperative with competitive solutions is not trivial, but is necessary for mixed environments, where agents have to coordinate with both team members, adversaries, and other neutral entities. In A3C3's setting, optimizing communication to hinder opponents and help team mates is an intuitive solution. However, how to determine which agents are considered opponents or team mates is also not trivial. That being said, considering solely cooperative environments, A3C3's biggest drawback is its sample inefficiency, requiring millions of environment interactions, which makes its deployment in real-world scenarios much harder. Its mechanisms, however, are end-to-end differentiable, and can likely be adapted to other more efficient deep reinforcement learning algorithms.

Looking at the field of MARL as a whole, or even regarding single-agent reward-based learning, the next great step should move in the direction of *concept formation*. The vast majority of algorithms assumes a stationary action-space, and researchers commonly group policies by hand to use them as higher-level strategies. For example, the FCPortugal3D team optimizes low-level behaviors where the action-space are the agent's joints. After a behavior is successfully optimized, it is defined as a *kick* or some other movement, and that middle-level behavior is then executed by the higher-level strategy. In nature this happens intuitively. Humans do not think about moving their legs to take a step, taking steps to walk, or walking to reach a point, but the hierarchy of behaviors is obvious. We believe that the next big step for machine learning research, and particularly for deep reinforcement learning, is to find these behaviors that create a perceivable change in the agent's state, and define them as additional actions that can be performed. Continuing with the example of FCPortugal3D, an algorithm would start with only the possibility of moving its joints, and eventually add new more complex behaviors to its action-space, such as standing up, or kicking the ball. These next steps will form the basis for life-long learning [216] and artificial general intelligence, in our opinion.

Bibliography

[1] Frederick Ducatelle, Gianni A Di Caro, Alexander Förster, Michael Bonani, Marco Dorigo, Stéphane Magnenat, Francesco Mondada, Rehan O'Grady, Carlo Pinciroli, Philippe Rétornaz, et al. Cooperative navigation in robotic swarms. *Swarm Intelligence*, 8(1):1–33, 2014.

[2] Michael Hoy, Alexey S Matveev, and Andrey V Savkin. Collision free cooperative navigation of multiple wheeled robots in unknown cluttered environments. *Robotics and Autonomous Systems*, 60(10):1253–1266, 2012.

[3] Gregor HW Gebhardt, Kevin Daun, Marius Schnaubelt, and Gerhard Neumann. Learning robust policies for object manipulation with robot swarms. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7688–7695. IEEE, 2018.

[4] Lynne E Parker and Claude Touzet. Multi-robot learning in a cooperative observation task. In *Distributed Autonomous Robotic Systems 4*, pages 391–401. Springer, 2000.

[5] Lili Ma and Naira Hovakimyan. Vision-based cyclic pursuit for cooperative target tracking. *Journal of Guidance, Control, and Dynamics*, 36(2):617–622, 2013.

[6] Cyril Robin and Simon Lacroix. Multi-robot target detection and tracking: taxonomy and survey. *Autonomous Robots*, 40(4):729–760, 2016.

[7] Hyondong Oh, Seungkeun Kim, Hyo-Sang Shin, Antonios Tsourdos, and Brian White. Coordinated standoff tracking of groups of moving targets using multiple uavs. In *Control & Automation (MED), 2013 21st Mediterranean Conference on*, pages 969–977. IEEE, 2013.

[8] Tim Brys, Tong T Pham, and Matthew E Taylor. Distributed learning and multi-objectivity in traffic light control. *Connection Science*, 26(1):65–83, 2014.

[9] Patrick Mannion, Jim Duggan, and Enda Howley. An experimental review of reinforcement learning algorithms for adaptive traffic signal control. In *Autonomic Road Transport Support Systems*, pages 47–66. Springer, 2016.

[10] P Skobelev, E Simonova, and A Zhilyaev. Using multi-agent technology for the distributed management of a cluster of remote sensing satellites. *Complex Systems: Fundamentals & Applications*, 90:287, 2016.

[11] Haibo Min, Shicheng Wang, Fuchun Sun, Zhijie Gao, and Jinsheng Zhang. Decentralized adaptive attitude synchronization of spacecraft formation. *Systems & Control Letters*, 61(1):238–246, 2012.

[12] Jerzy Korczak, Marcin Hernes, and Maciej Bac. Risk avoiding strategy in multi-agent trading system. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 1131–1138. IEEE, 2013.

[13] Tim Baarslag, Katsuhide Fujita, Enrico H Gerding, Koen Hindriks, Takayuki Ito, Nicholas R Jennings, Catholijn Jonker, Sarit Kraus, Raz Lin, Valentin Robu, et al. Evaluating practical negotiating agents: Results and analysis of the 2011 international competition. *Artificial Intelligence*, 198:73–103, 2013.

[14] Prabal Dutta, Mike Grimmer, Anish Arora, Steven Bibyk, and David Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 70. IEEE Press, 2005.

[15] Victor Lesser, Milind Tambe, and Charles L. Ortiz, editors. *Distributed Sensor Networks: A Multiagent Perspective*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[16] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *Proceedings of the first international conference on Autonomous agents*, pages 340–347. ACM, 1997.

[17] Mark Yim, Ying Zhang, John Lamping, and Eric Mao. Distributed control for 3d metamorphosis. *Autonomous Robots*, 10(1):41–56, 2001.

[18] Nuno Lau, Artur Pereira, Andreia Melo, António Neves, and João Figueiredo. Ciberrato: A simulation environment for mobile and autonomous robots. *Electrónica e Telecomunicações*, 3(7):647–650, 2002.

[19] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M. Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Yuhuai Wu, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[20] OpenAI. OpenAI Five. `https://blog.openai.com/openai-five/`, 2018.

[21] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[22] Rui Prada, Phil Lopes, Joao Catarino, Joao Quiterio, and Francisco S Melo. The geometry friends game AI competition. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 431–438. IEEE, 2015.

[23] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6379–6390, 2017.

[24] José M. Vidal. *Fundamentals of Multiagent Systems: Using NetLogo Models*. Unpublished, 2010. `http://www.multiagent.com`.

[25] Sascha Ossowski. Coordination in multi-agent systems: Towards a technology of agreement. In *German Conference on Multiagent System Technologies*, pages 2–12. Springer, 2008.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[27] Vlad Firoiu, William F. Whitney, and Joshua B. Tenenbaum. Beating the world's best at super smash bros. with deep reinforcement learning. *CoRR*, abs/1702.06230, 2017.

[28] Shani Gamrian and Yoav Goldberg. Transfer learning for related reinforcement learning tasks via image-to-image translation. *arXiv preprint arXiv:1806.07377*, 2018.

[29] Devendra Singh Chaplot, Guillaume Lample, Kanthashree Mysore Sathyendra, and Ruslan Salakhutdinov. Transfer deep reinforcement learning in 3d environments: An empirical study. In *NIPS Deep Reinforcemente Leaning Workshop*, 2016.

[30] André Barreto, Diana Borsa, John Quan, Tom Schaul, David Silver, Matteo Hessel, Daniel Mankowitz, Augustin Žídek, and Remi Munos. Transfer in deep reinforcement learning using successor features and generalised policy improvement. *arXiv preprint arXiv:1901.10964*, 2019.

[31] Pablo Hernandez-Leal, Michael Kaisers, Tim Baarslag, and Enrique Munoz de Cote. A survey of learning in multiagent environments: Dealing with non-stationarity. *arXiv preprint arXiv:1707.09183*, 2017.

[32] Stefano V Albrecht and Peter Stone. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence*, 258:66–95, 2018.

[33] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E Taylor. Is multiagent deep reinforcement learning the answer or the question? a brief survey. *arXiv preprint arXiv:1810.05587*, 2018.

[34] L. Busoniu, R. Babuska, and B. De Schutter. A comprehensive survey of multiagent reinforcement learning. *Trans. Sys. Man Cyber Part C*, 38(2):156–172, March 2008.

[35] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[36] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI

'98, pages 746–752, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[37] Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, pages 195–210. Morgan Kaufmann Publishers Inc., 1996.

[38] Nuno Lau and Luis Paulo Reis. *FC Portugal - High-level coordination methodologies in soccer robotics*. InTech Education and Publishing, Vienna, Austria, December 2007.

[39] Prasanna Velagapudi, Oleg Prokopyev, Paul Scerri, and Katia Sycara. A token-based approach to sharing beliefs in a large multiagent team. In *Optimization and Cooperative Control Strategies*, pages 417–429. Springer, 2009.

[40] Peter Stone and Manuela Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273, 1999.

[41] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[42] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

[43] Mihai Barbuceanu and Mark Fox. Cool: A language for describing coordination in multi agent systems. *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 17–24, 01 1995.

[44] Sainbayar Sukhbaatar, Rob Fergus, et al. Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems*, pages 2244–2252, 2016.

[45] David B D'Ambrosio, Skyler Goodell, Joel Lehman, Sebastian Risi, and Kenneth O Stanley. Multirobot behavior synchronization through direct neural network communication. In *International Conference on Intelligent Robotics and Applications*, pages 603–614. Springer, 2012.

[46] Bikramjit Banerjee and Jing Peng. Performance bounded reinforcement learning in strategic interactions. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pages 2–7. AAAI Press, 2004.

[47] Chongjie Zhang and Victor Lesser. Multi-agent learning with policy prediction. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 927–934. AAAI Press, 2010.

[48] Junling Hu and Michael P Wellman. Nash q-learning for general-sum stochastic games. *Journal of machine learning research*, 4(Nov):1039–1069, 2003.

[49] Vincent Conitzer and Tuomas Sandholm. Awesome: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents. *Machine Learning*, 67(1-2):23–43, 2007.

[50] Reinaldo AC Bianchi, Murilo F Martins, Carlos HC Ribeiro, and Anna HR Costa. Heuristically-accelerated multiagent reinforcement learning. *IEEE transactions on cybernetics*, 44(2):252–265, 2014.

[51] Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215 – 250, 2002.

[52] Michael Bowling. Convergence and no-regret in multiagent learning. In *Proceedings of the 17th International Conference on Neural Information Processing Systems*, NIPS'04, pages 209–216, Cambridge, MA, USA, 2004. MIT Press.

[53] M. D. Awheda and H. M. Schwartz. Exponential moving average q-learning algorithm. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 31–38, April 2013.

[54] Sherief Abdallah and Victor Lesser. A multiagent reinforcement learning algorithm with non-linear dynamics. *Journal of Artificial Intelligence Research*, 33:521–549, 2008.

[55] Jakob Foerster, Richard Y Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 122–130. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[56] David Simões, Nuno Lau, and Luís Paulo Reis. Multi-agent Double Deep Q-Networks. In Eugénio Oliveira, João Gama, Zita Vale, and Henrique Lopes Cardoso, editors, *Progress in Artificial Intelligence*, Lecture Notes in Computer Science, vol. 10423, pages 123–134. Springer International Publishing, 2017.

[57] David Simões, Nuno Lau, and Luís Paulo Reis. Mixed-policy asynchronous deep q-learning. In Anibal Ollero, Alberto Sanfeliu, Luis Montano, Nuno Lau, and Carlos Cardeira, editors, *ROBOT 2017: Third Iberian Robotics Conference*, Advances in Intelligent Systems and Computing, vol. 694, pages 129–140. Springer International Publishing, 2018.

[58] David Simões, Nuno Lau, and Luís Paulo Reis. Adjusted bounded weighted policy learner. In *Robocup 2018: Robot World Cup XXII*, Lecture Notes in Computer Science, vol. 11374, pages 324–336. Springer, 2018.

[59] David Simões, Nuno Lau, and Luís Paulo Reis. Guided deep reinforcement learning in the geofriends2 environment. In *IJCNN 18: International Joint-Conference on Neural Networks*, pages 375–381. IEEE, 2018.

[60] David Simoes, Simão Reis, Nuno Lau, and Luis Paulo Reis. Competitive deep reinforcement learning over a pokémon battling simulator. In *Autonomous Robot Systems and Competitions (ICARSC), 2020 IEEE International Conference on*. IEEE, April 2020.

[61] David Simões, Nuno Lau, and Luís Paulo Reis. Multi-agent neural reinforcement-learning system with communication. In Álvaro Rocha, Hojjat Adeli, Luís Paulo Reis, and Sandra Costanzo, editors, *New Knowledge in Information Systems and Technologies*, Advances in Intelligent Systems and Computing, vol. 931, pages 3–12. Springer International Publishing, 2019.

[62] David Simões, Nuno Lau, and Luís Paulo Reis. Multi-agent deep reinforcement learning with emergent communication. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019.

[63] David Simões, Pedro Amaro, Tiago Silva, Nuno Lau, and Luís Paulo Reis. Learning low-level behaviors and high-level strategies in humanoid soccer. In *ROBOT 2019: Fourth Iberian Robotics Conference*, Advances in Intelligent Systems and Computing. Springer International Publishing, 2019.

[64] David Simões, Nuno Lau, and Luís [Paulo Reis]. Multi-agent actor centralized-critic with communication. *Neurocomputing*, 390:40 – 56, 2020.

[65] David Simoes, Nuno Lau, and Luis Paulo Reis. Multi agent deep learning with cooperative communication. *Journal of Artificial Intelligence and Soft Computing Research*, 2020.

[66] David Simoes, Nuno Lau, and Luis Paulo Reis. Exploring communication protocols and centralized critics in multi-agent deep learning. *Integrated Computer-Aided Engineering*, 2020.

[67] Abbas Abdolmaleki, David Simões, Nuno Lau, Luis Paulo Reis, and Gerhard Neumann. Contextual relative entropy policy search with covariance matrix adaptation. In *Autonomous Robot Systems and Competitions (ICARSC), 2016 IEEE International Conference on*, pages 94–99. IEEE, May 2016.

[68] Abbas Abdolmaleki, David Simões, Nuno Lau, Luis Paulo Reis, and Gerhard Neumann. Learning a humanoid kick with controlled distance. In Sven Behnke, Raymond Sheh, Sanem Sarıel, and Daniel D. Lee, editors, *RoboCup 2016: Robot World Cup XX*, pages 45–57, Cham, 2017. Springer International Publishing.

[69] S. Mohammadreza Kasaei, David Simões, Nuno Lau, and Artur Pereira. A hybrid zmp-cpg based walk engine for biped robots. In Anibal Ollero, Alberto Sanfeliu, Luis Montano, Nuno Lau, and Carlos Cardeira, editors, *ROBOT 2017: Third Iberian Robotics Conference*, pages 743–755. Springer International Publishing, 2018.

[70] Abbas Abdolmaleki, David Simões, Nuno Lau, Luís Paulo Reis, and Gerhard Neumann. Contextual direct policy search. *Journal of Intelligent & Robotic Systems*, 96(2):141–157, Nov 2019.

[71] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[72] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous agents and multi-agent systems*, 11(3):387–434, 2005.

[73] Nikos Vlassis. A concise introduction to multiagent systems and distributed artificial intelligence. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 1(1):1–71, 2007.

[74] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice Hall, Upper Saddle River, 2003.

[75] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[76] Jeffery A. Clouse. Learning from an automated training agent. In *Adaptation and Learning in Multiagent Systems*. Springer Verlag, 1996.

[77] Lisa Torrey and Matthew E Taylor. Help an agent out: Student/teacher learning in sequential decision tasks. In *Proceedings of the Adaptive and Learning Agents workshop (at AAMAS-12)*, 2012.

[78] Bob Price and Craig Boutilier. Accelerating reinforcement learning through implicit imitation. *Journal of Artificial Intelligence Research*, 19:569–629, 2003.

[79] Saleha Raza and Sajjad Haider. Using imitation to build collaborative agents. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11(1):3, 2016.

[80] Erfu Yang and Dongbing Gu. A survey on multiagent reinforcement learning towards multi-robot systems. In *IEEE 2005 Symposium on Computational Intelligence and Games, CIG'05*, pages 292–299. IEEE, 2005.

[81] Laetitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. Independent reinforcement learners in cooperative markov games: a survey regarding coordination problems. *The Knowledge Engineering Review*, 27(01):1–31, 2012.

[82] Michael Bowling and Manuela Veloso. Rational and convergent learning in stochastic games. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'01, pages 1021–1026, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[83] Sanyam Kapoor. Multi-agent reinforcement learning: A report on challenges and approaches. *CoRR*, abs/1807.09427, 2018.

[84] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[85] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[86] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[87] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[88] Sun-Chong Wang. Artificial neural network. In *Interdisciplinary Computing in Java Programming*, pages 81–100. Springer, 2003.

[89] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[90] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[91] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *International Conference for Learning Representations*, 2016.

[92] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[93] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[94] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[95] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[96] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations, San Diego*, 2015.

[97] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[98] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[99] Tabish Rashid, Mikayel Samvelyan, Christian Schröder de Witt, Gregory Farquhar, Jakob N. Foerster, and Shimon Whiteson. QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning. *CoRR*, abs/1803.11485, 2018.

[100] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[101] Andrea Bonarini and Vito Trianni. Learning fuzzy classifier systems for multi-agent coordination. *Information Sciences*, 136(1):215–239, 2001.

[102] Sandip Sen, Mahendra Sekaran, John Hale, et al. Learning to coordinate without sharing information. In *AAAI*, pages 426–431, 1994.

[103] B Gerkey and Maja J Mataric. Are (explicit) multi-robot coordination and multi-agent coordination really so different. In *Proceedings of the AAAI spring symposium on bridging the multi-agent and multi-robotic research gap*, pages 1–3, 2004.

[104] Michael R Genesereth, Matthew L Ginsberg, and Jeffrey S Rosenschein. *Cooperation without communication.* Heuristic Programming Project, Computer Science Department, Stanford University, 1984.

[105] Michael Isik, Freek Stulp, Gerd Mayer, and Hans Utz. Coordination without negotiation in teams of heterogeneous robots. In *Robot Soccer World Cup*, pages 355–362. Springer, 2006.

[106] George W Brown. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13(1):374–376, 1951.

[107] YC Jiang, P Yi, SY Zhang, and YP Zhong. Constructing agents blackboard communication architecture based on graph theory. *Computer Standards & Interfaces*, 27(3):285–301, 2005.

[108] Luis Paulo Reis and Nuno Lau. FC Portugal team description: RoboCup 2000 simulation league champion. In *Robot Soccer World Cup*, pages 29–40. Springer, 2000.

[109] Abhishek Das, Satwik Kottur, José MF Moura, Stefan Lee, and Dhruv Batra. Learning cooperative visual dialog agents with deep reinforcement learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2951–2960, 2017.

[110] Igor Mordatch and Pieter Abbeel. Emergence of grounded compositional language in multi-agent populations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[111] Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, 2002.

[112] Michael Bowling. Convergence and no-regret in multiagent learning. *Advances in neural information processing systems*, 17:209–216, 2005.

[113] Rob Powers and Yoav Shoham. New criteria and a new algorithm for learning in multi-agent systems. In *Advances in neural information processing systems*, pages 1089–1096, 2004.

[114] Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002.

[115] Milos Hauskrecht. Value-function approximations for partially observable markov decision processes. *Journal of Artificial Intelligence Research*, 13:33–94, 2000.

[116] John F Nash et al. Equilibrium points in n-person games. *Proc. Nat. Acad. Sci. USA*, 36(1):48–49, 1950.

[117] Jelle Rogier Kok. *Coordination and learning in cooperative multiagent systems.* PhD thesis, Universiteit van Amsterdam, 2006.

[118] Aleksander Byrski, Rafał Dreżewski, Leszek Siwik, and Marek Kisiel-Dorohinicki. Evolutionary multi-agent systems. *The Knowledge Engineering Review*, 30(02):171–186, 2015.

[119] Amy Greenwald, Keith Hall, and Roberto Serrano. Correlated q-learning. In *ICML*, volume 3, pages 242–249, 2003.

[120] David Carmel and Shaul Markovitch. Opponent modeling in multi-agent systems. In *International Joint Conference on Artificial Intelligence*, pages 40–52. Springer, 1995.

[121] William Uther and Manuela Veloso. Adversarial reinforcement learning. Technical report, Technical report, Carnegie Mellon University, 1997. Unpublished, 1997.

[122] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored mdps. In *NIPS*, volume 1, pages 1523–1530, 2001.

[123] Jelle R Kok and Nikos Vlassis. Sparse cooperative q-learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 61. ACM, 2004.

[124] Lucian Busoniu, Bart De Schutter, and Robert Babuska. Multiagent reinforcement learning with adaptive state focus. In *BNAIC*, pages 35–42, 2005.

[125] Maja J Matarić. Reinforcement learning in the multi-robot domain. In *Robot colonies*, pages 73–83. Springer, 1997.

[126] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, pages 1038–1044, 1996.

[127] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–503, 2016.

[128] Olivier Buffet, Alain Dutech, and François Charpillet. Shaping multi-agent systems with gradient reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 15(2):197–220, 2007.

[129] Peter Stone. *Layered learning in multiagent systems: A winning approach to robotic soccer*. MIT Press, 1998.

[130] Yu han Chang, Tracey Ho, and Leslie P. Kaelbling. All learning is local: Multi-agent learning in global reward games. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 807–814. MIT Press, 2004.

[131] Jakob N Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[132] Michael R Genesereth, Richard E Fikes, et al. *Knowledge interchange format-version 3.0: reference manual*. Computer Science Department, Stanford University San Francisco, CA, 1992.

[133] Keith S Decker and Victor R Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(02):319–346, 1992.

[134] Cristiano Castelfranchi. Commitments: From individual intentions to groups and organizations. In *ICMAS*, volume 95, pages 41–48, 1995.

[135] Peter Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press, Cambridge, MA, USA, 2000.

[136] Luís Paulo Reis, Nuno Lau, and Eugénio Oliveira. Situation based strategic positioning for coordinating a team of homogeneous agents. In *Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, volume 2103 of *LNCS*, pages 175–197. Springer, 2001.

[137] Spiros Kapetanakis and Daniel Kudenko. Reinforcement learning of coordination in cooperative multi-agent systems. In *Eighteenth National Conference on Artificial Intelligence*, pages 326–331, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[138] Martin Lauer and Martin Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*, pages 535–542. Morgan Kaufmann, 2000.

[139] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[140] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *International Conference for Learning Representations*, 2016.

[141] Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *CoRR*, abs/1703.10069, 2017.

[142] Jakob Foerster, Ioannis Alexandros Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2137–2145, 2016.

[143] Lloyd S Shapley. A value for n-person games. *Contributions to the Theory of Games*, 2(28):307–317, 1953.

[144] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the eleventh international conference on machine learning*, volume 157, pages 157–163, 1994.

[145] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press, 2016.

[146] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992.

[147] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *Proceedings of the International Conference on Learning Representations*, 2017.

[148] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[149] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[150] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *CoRR*, abs/1602.02672, 2016.

[151] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PLOS ONE*, 12(4):1–15, 04 2017.

[152] Maxim Egorov. Multi-Agent Deep Reinforcement Learning. Technical report, University of Stanford, Department of Computer Science, 2016.

[153] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 66–83. Springer, 2017.

[154] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, pages 2085–2087, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

[155] Angeliki Lazaridou, Alexander Peysakhovich, and Marco Baroni. Multi-agent cooperation and the emergence of (natural) language. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[156] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[157] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[158] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1146–1155. JMLR. org, 2017.

[159] Mike Lewis, Denis Yarats, Yann Dauphin, Devi Parikh, and Dhruv Batra. Deal or no deal? end-to-end learning of negotiation dialogues. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2443–2453. Association for Computational Linguistics, 2017.

[160] Qiyang Li, Xintong Du, Yizhou Huang, Quinlan Sykora, and Angela P. Schoellig. Learning of coordination policies for robotic swarms. *CoRR*, abs/1709.06620, 2017.

[161] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[162] George Cybenko and Daniella Rus. Agent-based systems engineering. Technical report, DTIC Document, 2005.

[163] Alan J Robinson and Lee Spector. Using genetic programming with multiple data types and automatic modularization to evolve decentralized and coordinated navigation in multi-agent systems. In *GECCO Late Breaking Papers*, pages 391–396, 2002.

[164] Valentino Crespi, George Cybenko, Daniela Rus, and Massimo Santini. Decentralized control for coordinated flow of multi-agent systems. In *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, volume 3, pages 2604–2609, 2002.

[165] Martin Saska, Jan Chudoba, Libor Přeučil, Justin Thomas, Giuseppe Loianno, Adam Třešňák, Vojtěch Vonásek, and Vijay Kumar. Autonomous deployment of swarms of micro-aerial vehicles in cooperative surveillance. In *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, pages 584–595. IEEE, 2014.

[166] K Ovchinnikov, A Semakova, and A Matveev. Cooperative surveillance of unknown environmental boundaries by multiple nonholonomic robots. *Robotics and Autonomous Systems*, 72:164–180, 2015.

[167] Shaofei Chen, Feng Wu, Lincheng Shen, Jing Chen, and Sarvapali D. Ramchurn. Multi-agent patrolling under uncertainty and threats. *PLoS ONE*, 10(6):1–19, 06 2015.

[168] Fabrice Lauri and Abderrafiaa Koukam. Robust multi-agent patrolling strategies using reinforcement learning. In *International Conference on Swarm Intelligence Based Optimization*, pages 157–165. Springer, 2014.

[169] Julia Nilsson and Jonas Sjöberg. Strategic decision making for automated driving on two-lane, one way roads using model predictive control. In *2013 IEEE Intelligent Vehicles Symposium (IV)*, pages 1253–1258. IEEE, 2013.

[170] Yeping Hu, Alireza Nakhaei, Masayoshi Tomizuka, and Kikuo Fujimura. Interaction-aware decision making with adaptive strategies under merging scenarios. *arXiv preprint arXiv:1904.06025*, 2019.

[171] Ana LC Bazzan and Franziska Klügl. A review on agent-based technology for traffic and transportation. *The Knowledge Engineering Review*, 29(03):375–403, 2014.

[172] Nitin Maslekar, Joseph Mouzna, Mounir Boussedjra, and Houda Labiod. Cats: An adaptive traffic signal system based on car-to-car communication. *Journal of network and computer applications*, 36(5):1308–1315, 2013.

[173] Oleg N Granichin, Petr Skobelev, Alexander Lada, Igor Mayorov, and Alexander Tsarev. Comparing adaptive and non-adaptive models of cargo transportation in multi-agent system for real time truck scheduling. In *IJCCI*, pages 282–285, 2012.

[174] Jörg P Müller and Markus Pischel. An architecture for dynamically interacting agents. *International Journal of Intelligent and Cooperative Information Systems*, 3(01):25–45, 1994.

[175] Adrian K Agogino and Kagan Tumer. A multiagent approach to managing air traffic flow. *Autonomous Agents and Multi-Agent Systems*, 24(1):1–25, 2012.

[176] Frederic Marc, Amal El Fallah-Seghrouchni, and Irene Degirmenciyan-Cartault. Coordination of complex systems based on multi-agent planning: Application to the aircraft simulation domain. In *International Workshop on Programming Multi-Agent Systems*, pages 224–248. Springer, 2004.

[177] Javad Ansari, Amin Gholami, and Ahad Kazemi. Multi-agent systems for reactive power control in smart grids. *International Journal of Electrical Power & Energy Systems*, 83:411–425, 2016.

[178] Luis Hernandez, Carlos Baladron, Javier M Aguiar, Belen Carro, Antonio Sanchez-Esguevillas, Jaime Lloret, David Chinarro, Jorge J Gomez-Sanz, and Diane Cook. A multi-agent system architecture for smart grid management and forecasting of energy demand in virtual power plants. *IEEE Communications Magazine*, 51(1):106–113, 2013.

[179] Arne Schuldt. Multiagent coordination enabling autonomous logistics. *KI-Künstliche Intelligenz*, 26(1):91–94, 2012.

[180] Zhou He, Shouyang Wang, and TCE Cheng. Competition and evolution in multi-product supply chains: An agent-based retailer model. *International Journal of Production Economics*, 146(1):325–336, 2013.

[181] Zhou He, TCE Cheng, Jichang Dong, and Shouyang Wang. Evolutionary location and pricing strategies in competitive hierarchical distribution systems: A spatial agent-based model. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(7):822–833, 2014.

[182] Smogon University. Competitive pokémon battling. `https://www.smogon.com/`, 2019.

[183] Markus Waibel, Laurent Keller, and Dario Floreano. Genetic team composition and level of selection in the evolution of cooperation. *IEEE Transactions on Evolutionary Computation*, 13(3):648–660, 2009.

[184] Alan Schultz, John J. Grefenstette, and William Adams. Robo-shepherd: Learning complex robotic behaviors. In *In Robotics and Manufacturing: Recent Trends in Research and Applications, Volume 6*, pages 763–768. ASME Press, 1996.

[185] Mitchell A Potter, Lisa A Meeden, and Alan C Schultz. Heterogeneity in the coevolved behaviors of mobile robots: The emergence of specialists. In *International joint conference on artificial intelligence*, volume 17, pages 1337–1343. Citeseer, 2001.

[186] Miroslav Benda. On optimal cooperation of knowledge sources. *Technical Report BCS-G2010-28*, 1985.

[187] Nuno Lau, Luís Paulo Reis, Nima Shafii, Rui Ferreira, and Abbas Abdolmaleki. FC Portugal 3D simulation team: Team description paper, 2013.

[188] R Dias, AJR Neves, JL Azevedo, B Cunha, J Cunha, P Dias, A Domingos, L Ferreira, P Fonseca, N Lau, E Pedrosa, A Pereira, R Serra, J Silva, and A Trifan. CAMBADA 2013 Team Description Paper, 2013.

[189] Peter Stone, Gregory Kuhlmann, Matthew E. Taylor, and Yaxin Liu. Keepaway soccer: From machine learning testbed to benchmark. In Ansgar Bredenfeld, Adam Jacoff, Itsuki Noda, and Yasutake Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, pages 93–105, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[190] David Simões, Rui Brás, Nuno Lau, and Artur Pereira. A coordinated team of agents to solve mazes. In *Robot 2015: Second Iberian Robotics Conference*, pages 381–392. Springer, 2016.

[191] Robert Weihmayer and Hugo Velthuijsen. Application of distributed ai and cooperative problem solving to telecommunications. *AI Approaches to Telecommunications and Network Management*, 1994.

[192] Justin A Boyan and Michael L Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems*, pages 671–671, 1994.

[193] Michael Rubenstein, Christian Ahler, Nick Hoff, Adrian Cabrera, and Radhika Nagpal. Kilobot: A low cost robot with scalable operations designed for collective behaviors. *Robotics and Autonomous Systems*, 62(7):966–975, 2014.

[194] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawai, and Hitoshi Matsubara. Robocup: A challenge problem for ai and robotics. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 1–19, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[195] Syamimi Shamsuddin, Luthffi Idzhar Ismail, Hanafiah Yussof, Nur Ismarrubie Zahari, Saiful Bahari, Hafizan Hashim, and Ahmed Jaffar. Humanoid robot nao: Review of control and motion exploration. In *2011 IEEE International Conference on Control System, Computing and Engineering*, pages 511–516. IEEE, 2011.

[196] Nintendo Game Freak, Creatures Inc. Pokemon series. Web: `https://www.pokemon.com`, 1996.

[197] Zarel. Pokémon showdown. Web: `https://pokemonshowdown.com/`, 2019.

[198] coyotte508. Pokémon online. Web: `http://pokemon-online.eu/`, 2019.

[199] Arnaud Durand. Pokémon battle api. Web: `https://github.com/DurandA/pokemon-battle-api`, 2019.

[200] Paul Hallet. Pokéapi. Web: `https://pokeapi.co/`, 2019.

[201] S. Lee and J. Togelius. Showdown ai competition. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 191–198, Aug 2017.

[202] Johannes Ackermann, Volker Gabler, Takayuki Osa, and Masashi Sugiyama. Reducing overestimation bias in multi-agent domains using double centralized critics. *arXiv preprint arXiv:1910.01465*, 2019.

[203] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

[204] Bikramjit Banerjee and Jing Peng. Generalized multiagent learning with performance bound. *Autonomous Agents and Multi-Agent Systems*, 15(3):281–312, 2007.

[205] Rajesh Rao. Decision making under uncertainty: A neural model based on partially observable markov decision processes. *Frontiers in Computational Neuroscience*, 4:146, 2010.

[206] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[207] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[208] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *International Conference for Learning Representations*, 2016.

[209] Ronald J. Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

[210] G.H.W. Gebhardt, M. Hüttenrauch, and G. Neumann. Using m-embeddings to learn control strategies for robot swarms. *Submitted to Swarm Intelligence*, submitted.

[211] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[212] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga,

Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[213] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[214] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019.

[215] He He, Jordan Boyd-Graber, Kevin Kwok, and Hal Daumé III. Opponent modeling in deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1804–1813, 2016.

[216] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 2019.