Deep Reinforcement Learning in Complex Environments

Nantas Nardelli

St Catherine's College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Michaelmas 2021

Abstract

Deep Reinforcement Learning (DRL), is becoming a popular and mature framework for learning to solve sequential decision making problems. The application of Deep Neural Networks, flexible and powerful function approximators, towards learning policies has effectively enabled RL to solve applications that were thought to be too difficult: from beating professional human players in hard games such as Go, to becoming the foundation for flexible embodied control. We explore what happens when one attempts to learn policies in environments that present complex dynamics and hard and structured tasks. As these environments provide challenges that lie fundamentally at the forefront what most state-of-the-art Reinforcement Learning methods try to tackle, they provide a general view of existing weaknesses, while also providing opportunities for improving the general framework as well as particular algorithms. Firstly, we study and develop methods for Deep Multi-Agent Reinforcement Learning, a setting in which multiple agents are interacting with an (often complex) environment and each other. The presence of multiple agents breaks some of the key assumptions that provide necessary stability to standard learning methods, creating unique and interesting problems. We test these methods by formulating a multi-agent version of the StarCraft micromanagement problem, an extremely complex real-time control and planning problem based on one of the hardest environments currently available in the literature. Secondly, in a single-agent version of the same problem, we investigate how DRL can be used to develop a set of parameter-efficient differentiable planning modules to solve path-planning tasks with complex environment dynamics and variable map sizes. We show that the modules enable learning to plan when the environment also includes stochastic elements, providing a cost-efficient learning system to build low-level size-invariant planners for a variety of interactive, hard navigation problems. Thirdly, and lastly, we present a novel RL benchmark based on one of the oldest and most complex video games ever developed: the NetHack Learning Environment (NLE). NLE provides an environment that is scalable, rich, and challenging for state-of-the-art RL, while maintaining familiarity with standard grid-worlds, and dramatically decreasing the computational requirements compared to existing environments of similar complexity and scope. We believe that this particular intersection of properties will enable the community to employ a single environment both as a debugging tool for increasingly complicated RL agents, and as a target for the next decade of RL research.

Deep Reinforcement Learning in Complex Environments

Nantas Nardelli

St Catherine's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2021

Acknowledgements

Writing a doctoral thesis makes for a truly interesting adventure. You begin with a some kind of bold, hopefully novel idea, you find a supervisor, and you crack on. You are told that this is *your* journey, and that you alone will eventually need to present a unified view of your work. This puts a lot of pressure on the student, some of which probably necessary – but strong pressure nonetheless. Before starting this adventure, I was afraid that I would not be able to enjoy the process (or even conclude it!), especially in the place of incredible research excellence that is Oxford. But Oxford has a few tricks in its arsenal to make sure that research students can successfully and proudly write their own acknowledgemnt sections. Its environment. Its community. Its people.

Like many others before me, here I wish to acknowledge all the support that I received throughout my degree; this thesis and the accomplishments it contains would likely otherwise not have materialised. Let's start first with my supervisor, **Philip Torr**. Phil gave me absolute academic freedom, trusting me to own both questions and answers to what I thought (and still think) the big problems in my field might be. He showed me that research can be engaging, interesting, and fun even after several decades of grind and hard work, and that the process, as difficult as it is, can and should be enjoyed. He constantly reminded me to make the most of my DPhil experience at Oxford, and his relentless support enabled me to do exactly that whilst also generating a series of work that would make any doctoral student proud. Now, I most likely would not have met Phil without the help of **Pushmeet Kohli**. Pushmeet took me under his guidance when I was still an inexperienced undergraduate, opening important doors and steering me onto the right personal and professional paths, continuosly offering the kind of exceptional feedback that very few people will ever get the chance to experience. These two figures taught me a lot, and I very much hope I'll continue to make them proud. My aim is to pass at least some the wisdom they imparted on me, and add more links to the chain.

I wish to thank my friends and colleagues in the Engineering and Computer Science departments, who allowed me to develop my research interest in a safe space, and honestly engaged with all sorts of research and engineering nonsense. They simply made my life better. The list is long, but from the Torr Vision Group (and associate folks) I'd like to mention **Rudy Bunel**, **Alban Desmaison**, **Leonard Berrada**, **Siddharth Narayanaswamy**, **Luca Bertinetto**, **Arslan Chaudhry**, **Namhoon Lee**, **Jack Valmadre**, **Nick Lord**, **Stuart Golodetz**, **Tommaso Cavallari**, **Thalaiyasingam Ajanthan**, **Kyle Shuai Zheng**, **Michael Sapienza**, **Ondrej Miksik**, **Anurag Arnab**, **Daniela Massiceti**, **Oscar Rahnama**, **Saumya Jetley**, **Viveka Kulharia**, **Rodrigo de Bem**, **Qizhu Li**, **Amartya Sanyal**, **Andrew Gambardella**, **Christian Schroeder de Witt**, **Harkirat Singh Behl**, **Arnab Ghosh**, **Bradley Gram-Hansen**, **Robert**

Zinkov, **Jimmy Shi**, **Tom Joy**, **Botos Csaba**, and **Jishnu Mukhoti**. In CS, **Shimon Whiteson** was kind enough to frequently let me join on projects, seminars, research groups, essentially soft-adopteding me as a member of his research group. I owe a lot of my research success to its members, who became some of my closest friends and strongest collaborators. To name a few, the incredible **Jakob Foerster** and **Greg Farquhar**, whose energy and perspective fueled many years of really exciting work, but also **Max Igl**, **Supratik Paul**, **Yannis Assael**, **Brendan Shillingford**, **Tabish Rashid**, **Wendelin Boehmer**, **Kamil Ciosek**, **Jelena Luketina**, **Luisa Zintgraf**, **Vitaly Kurin**, and **Mikayel Samvelyan**.

During my doctoral years, I also spent a lot of time outside of Oxford. **Gabriel Synnaeve** and **Nicolas Usunier** brought me to New York and Paris to do amazing work and learn about the world of industry research while working in the world-class lab that was Facebook AI Research. In my time there I met so many fantastic people, and in particular I have special thanks for **Zeming Lin**, **Soumith Chintala**, **Vasil Khalidov**, **Jonas Gehring**, **Vegard Mella**. At a later time I also ended up helping out with starting an excellent research group in London, and had a wonderful time working with the great **Tim Rocktaschel**, **Heinrich Kuttler**, **Edward Grefenstette**, **Alex Miller**, **Roberta Raileanu**, **Viswanath Sivakumar**, **Thibaut Lavril**, and many others. Finally, I have to thank **Feryal Behbahani** and **Nando de Freitas** for giving me a great and unforgettable experience at DeepMind, and all the members of the Machine Learning, Discovery, and RL teams that made remote work fun and exciting.

I also absolutely must thank **St Catherine's College**, the members of its **Middle Common Room**, and all the folks at the **Catz Boat Club**, for filling my time at Oxford with great experiences, lovely memories. I met hundreds of kind, smart, determined, wildly engaging people, and I'm fairly certain all are going to do great things in life. I grew in so many ways thanks to them; I learnt to love rowing both as a sport and a lifestyle; most importantly, I had tons of fun.

Finally, I could not have achieved any of the things I set out to do without the unwaivering support of my loved ones. My parents, **Vinicio Nardelli** and **Maria Antonietta Giacalone**, who never failed to believe in me, and stubbornly continue to do so. My brother, **Juri Nardelli**, who showed me that – against most popular advice nowadays – one can and should follow their dreams and ideals with no regrets. And my bedrock, **Suzannah Sherman**, whose love has done so much for me throughout these years that any word I attempt to write in response can only end up falling terribly short. Thank you, thank you, thank you. I'm truly grateful and privileged to be able to spend my life together with you all.

Abstract

Deep Reinforcement Learning (DRL), is becoming a popular and mature framework for learning to solve sequential decision making problems. The application of Deep Neural Networks, flexible and powerful function approximators, towards learning policies has effectively enabled RL to solve applications that were thought to be too difficult: from beating professional human players in hard games such as Go, to becoming the foundation for flexible embodied control. We explore what happens when one attempts to learn policies in environments that present complex dynamics and hard and structured tasks. As these environments provide challenges that lie fundamentally at the forefront what most state-of-the-art Reinforcement Learning methods try to tackle, they provide a general view of existing weaknesses, while also providing opportunities for improving the general framework as well as particular algorithms. Firstly, we study and develop methods for Deep Multi-Agent Reinforcement Learning, a setting in which multiple agents are interacting with an (often complex) environment and each other. The presence of multiple agents breaks some of the key assumptions that provide necessary stability to standard learning methods, creating unique and interesting problems. We test these methods by formulating a multi-agent version of the StarCraft micromanagement problem, an extremely complex real-time control and planning problem based on one of the hardest environments currently available in the literature. Secondly, in a single-agent version of the same problem, we investigate how DRL can be used to develop a set of parameter-efficient differentiable planning modules to solve path-planning tasks with complex environment dynamics and variable map sizes. We show that the modules enable learning to plan when the environment also includes stochastic elements, providing a cost-efficient learning system to build low-level size-invariant planners for a variety of interactive, hard navigation problems. Thirdly, and lastly, we present a novel RL benchmark based on one of the oldest and most complex video games ever developed: the NetHack Learning Environment (NLE). NLE provides an environment that is scalable, rich, and challenging for state-of-the-art RL, while maintaining familiarity with standard grid-worlds, and dramatically decreasing the computational requirements compared to existing environments of similar complexity and scope. We believe that this particular intersection of properties will enable the community to employ a single environment both as a debugging tool for increasingly complicated RL agents, and as a target for the next decade of RL research.

Contents

1 Introduction

The past few years have seen the rise of Machine Learning as a revolutionary tool for severely enhancing our innate drive of learning from the extraordinary quantity of data the individuals, society, humans, Earth, and the universe seem to want to continuously generate. Computers are now able to give us a new perspective on large chunks of existing human knowledge, now often going through quantities of data that vastly go beyond what a single human lifetime can feasibly experience. Or thousands. We are effectively speeding up processes that would have normally taken humanity decades or even centuries, affecting the human experience in exciting – and at times scary! – ways.

There are many reasons behind this success, but the main one is that with the increase in availability of fast computing resources, it became sustainable to train Neural Networks (NNs) with large number of parameters, leading to the development of Deep Learning, and the production of massive improvements across the board. What we used to think were extremely hard problems in computer vision, speech recognition, and natural language processing suddenly found solutions significantly better than what got produced in decades of focused, domain-driven, expensive research and development[103]. This thesis itself began to be developed shortly after the release of the first few breakthroughs generated by Deep Learning models, and has been strongly shaped by the changes introduced by this new framework.

Deep Learning involves a fundamentally simple process: pick an expressive function approximators (again, neural nets), choose an objective function that maps to the task that needs to be solved, and optimise the parameters of the function via gradient descent to minimise this objective, until the task is believed to be

sufficiently solved. At its core, it is a wildly democratising process, as it enables engineers and makers with no significant machine learning experience, no particular knowledge about non-convex optimisation, no intuition about probability theory, to just get the job done. It is truly impressive in its impact, and it easily (and somewhat annoyingly!) leads us to make wild speculations about its future, as the subject is still very much in its infancy and we understand extremely little of what we are using.

However, not all is golden in this picture: as an attentive reader might have noticed, this process transforms a *learning* problem into one of *optimisation*. This mapping is somewhat straightforward in the context of *supervised learning*, where we are generally learning a model to predict some unseen data (or variable, or property) via the minimisation of some prediction error. However a lot of what systems do is not just prediction. It's *acting*! *Planning*! *Reasoning*! All processes that don't automatically come together with some clear and quick-to-use functions to optimise against.

1.1 Deep Reinforcement Learning

To tackle and study sequential decision problems, where acting and interacting are the fundamental goals of the system, the AI community has developed an ingenious framework to generalise and reduce all such problems into a neat optimisation challenge: Reinforcement Learning. An agent lives in an environment, and every time step it takes an action that modifies the state of this environment. The environment in turns gives an observation to the agent that maps to some (often unknown) underlying state, and a numerical value that contributes to telling the agent how well the implied task to be solved is being performed (a reward). Provided that a reward function can be defined to a particular task, this framework enables to focus on developing agents that aim to maximise the total reward in their lifetime (which may or may not be defined as an episode) through trial-and-error. Chapter 2.1 goes into details about the mathematical formalisms that effectively enable us to learn from this process in a fairly general manner.

Deep Reinforcement Learning is in a sense the modern version of this framework, and it is primarily concerned with developing and analysing RL algorithms that employ Deep Neural Networks as function approximators. This idea saw being developed in the early 90's to try to deal with high-dimensional state spaces, and (at the time) complex environments. A very early example was TD-Gammon[183], which employed an NN-based value function to play backgammon, achieving performance comparable to the highest level of human players in a significantly shorter time

than these (or any) humans. However, the real explosion of work on non-linear function approximators applied to RL is far more recent, and steamed primarily from the desire of exploiting Deep Learning's success to make similarly impactful breakthroughs as the applications mentioned above.

The first example of truly unexpected results, which fueled the creation of the field, was the work proposed by [117], who showed that one could train a policy approximated by a relatively simple neural network to play a large collection of classic Atari videogames, using pixel-based observations and a common RL algorithm, Q-learning. This was followed by further and similarly impressive work on developing and adapting existing RL techniques to play Go[159, 160], one of the most difficulty abstract games that humans have ever played. And such work succeeded in fundamentally creating agents that can beat all humans who ever played the game (and all that will most likely play it in the future). Deep Reinforcement Learning is now a successful field of research, and its algorithms are being applied to a number of domains, replacing purpose-built robotics architectures, computer vision systems, large-scale optimisation programs, and many others, with general learning algorithms that can successfully exploit the available data.

That said, if Deep Learning can be compared to a baby making its first steps, Deep Reinforcement Learning is an infant that has barely begun attepting to understand the objects surrounding itself and the laws that govern them. Learning policies is fundamentally unstable and source of extreme frustration. It is somewhat well known by practitioners that DRL can unexpectedly – and often randomly – produce awful solutions, as if training deep policies might be heavily dependent on someone in Paris failing to place a lovely amount of cherries on a strawberry and chocolate cake.[1]

So, in the face of this instability and general lack of knowledge, how do we make progress towards understanding and improving the status quo in Deep Reinforcement Learning? The community seems to largely be focused on gradual improvements on well beaten paths (see Section 2.4 for a good discussion on this), but it is unclear what the correct tradeoff is between exploiting what we already know and exploring in the darkness of new problems and solutions. Unsurprisingly, this is what we will set out to do. We will look for answer to questions and problems that are intuitively challenging to the status quo. We will try to check what happens when we try to learn policies at the boundary of what is considered possible; what happens when we try to learn policies that try to estimate complex environment functions; what happens when we go beyond what is commonly tried.

[1]The author of this thesis may or may not believe this to be true.

In this thesis, we attempt to go beyond the standard RL setting, and put the RL framework, (some of) its algorithms, architectures, models, and general assumptions in unconfortable state where common assumptions do not necessarily hold. In the process, we will also try to define what it takes for a benchmark to provide good experimental settings that provide incentives and arising challenges to easily implement these questions into research directions, fixing issues and discovering new methods as we go through this process.

1.2 Thesis Contributions and Format

This is a general outline of the format and the contributions that compose this thesis.

Chapter 2 introduces background required knownledge on Reinforcement Learning. It begins with a quick overview of classic methods, discusses the transition to Deep Reinforcement Learning methods, delves into Multi-Agent RL, and finally quickly reviews the state of benchmarks in the field.

Chapter 3 discusses Deep Multi-Agent Reinforcement Learning, introducing (a) the problem of non-stationarity in methods that employ buffers of historical data, and some solutions to solve it, (b) a new Actor-Critic algorithm closely exploiting the concept of *centralised learning and decentralised execution* to solve the multi-agent credit assignment problem, and (c) the StarCraft Multi-Agent Challenge, a benchmark based on the *StarCraft micromanagement* problem employed to test the methods introduced in the Chapter, which synthesises what we learnt from developing and evaluating the learning framework. The chapter is based of three separate publications [55, 54, 150] (respectively at ICML 2017, AAAI 2018, and AAMAS 2019). I was lead co-author for the first paper together with Jakob Foerster, the second paper was a joint effort among all the junior authors, but particularly led by Jakob Foerster and Gregory Farquhar (we all shared writing duties, and I particularly focused on the experimental setting), utilising ideas originally developed during the prototyping phase of the first paper; the third paper was also a (larger!) team effort, in which I – together with my now-senior collaborators from the previous two papers – primarily acted in a supervisory role and focused on writing and research directions, given the large-scale engineering focus of the project and the many contributors to manage. These three papers, jointly, have strongly contributed to the kickstarting a still unstopped wave of exciting Deep Multi-Agent RL work in the larger community, and to the establishment of StarCraft as a useful and modern multi-agent benchmark.

Chapter 4 discusses Value Propagation Networks, published to ICLR 2019 as [120]. This paper introduces two low-level differentiable modules that can be trained with Reinforcement Learning to produce agents that solve path planning problems in a variety of complex and stochastic scenarios. The work focuses on exploring the use of Deep Reinforcement Learning for a particularly well constrained but complex problem, with the (achieved) objective to gain some generalisation capabilities and improve the state of the art. I was the lead author of the work, with major technical contributions by the rest of the authors, with particular mention of Nicolas Usunier and Gabriel Synnaeve, who provided supervision and technical ideas during the development of the project.

Chapter 5 discusses the NetHack Learning Environment, a novel environment based on NetHack, a wildly complex, long-lived, terminal-based video game. This environment is likely to be the hardest grid-world the community will ever need to work on, and provides in my opinion a next generation grand-challenge that displays particularly critical weaknesses of current SOTA RL architectures. The paper was published at NeurIPS 2020 as [100], and later became the foundation for one of the NeurIPS 2021 competitions (the first, at the present time). The primary technical lead of the paper was Heinrich Küttler, who spearheaded the early prototype of the environment and provided technical leadership and manpower to the project. I strongly contributed to both the required engineering, experimental, and writing efforts together with him and Tim Rocktäschel, but many technical and non-technical contributions generally came from the entirety of the author team.

Chapter 6 concludes the thesis with a retrospective on each of these chapters and the overall document, providing details on some interesting future directions as well as reviewing some of the body of work that the community has already built on the contributions made by this thesis.

1.3 Additional contributions

During the development of this thesis, there were various publications that couldn't be cleanly included in this thesis, as they were either exploring different topics, the contribution was not sufficient to warrant inclusion in the dissertation, or they would decrease the quality of the overall dissertation in some way or another. Here we note their status as publications, and we include their respective abstract:

Playing Doom with SLAM-Augmented Deep Reinforcement Learning [19] (ArXiv 2016) – A number of recent approaches to policy learning in 2D game domains have been successful going directly from raw input images to actions. However when employed in complex 3D environments, they typically suffer from challenges related to partial observability, combinatorial exploration spaces, path planning, and a scarcity of rewarding scenarios. Inspired from prior work in human cognition that indicates how humans employ a variety of semantic concepts and abstractions (object categories, localisation, etc.) to reason about the world, we build an agent-model that incorporates such abstractions into its policy-learning framework. We augment the raw image input to a Deep Q-Learning Network (DQN), by adding details of objects and structural elements encountered, along with the agent's localisation. The different components are automatically extracted and composed into a topological representation using on-the-fly object detection and 3D-scene reconstruction.We evaluate the efficacy of our approach in Doom, a 3D first-person combat game that exhibits a number of challenges discussed, and show that our augmented framework consistently learns better, more effective policies.

A Survey of Reinforcement Learning Informed by Natural Language [110] (IJCAI 2019) – To be successful in real-world tasks, Reinforcement Learning (RL) needs to exploit the compositional, relational, and hierarchical structure of the world, and learn to transfer it to the task at hand. Recent advances in representation learning for language make it possible to build models that acquire world knowledge from text corpora and integrate this knowledge into downstream decision making problems. We thus argue that the time is right to investigate a tight integration of natural language understanding into RL in particular. We survey the state of the field, including work on instruction following, text games, and learning from textual domain knowledge. Finally, we call for the development of new environments as well as further investigation into the potential uses of recent Natural Language Processing (NLP) techniques for such tasks.

TorchBeast: A PyTorch Platform for Distributed RL [98] (ArXiv 2019) – TorchBeast is a platform for reinforcement learning (RL) research in PyTorch. It implements a version of the popular IMPALA algorithm for fast, asynchronous, parallel training of RL agents. Additionally, TorchBeast has simplicity as an explicit design goal: We provide both a pure-Python implementation ("MonoBeast") as well as a multi-machine high-performance version ("PolyBeast"). In the latter, parts of the implementation are written in C++, but all parts pertaining to machine

learning are kept in simple Python using PyTorch, with the environments provided using the OpenAI Gym interface. This enables researchers to conduct scalable RL research using TorchBeast without any programming knowledge beyond Python and PyTorch. In this paper, we describe the TorchBeast design principles and implementation and demonstrate that it performs on-par with IMPALA on Atari. TorchBeast is released as an open-source package under the Apache 2.0 license and is available at https://github.com/facebookresearch/torchbeast.

MVFST-RL: An Asynchronous RL Framework for Congestion Control with Delayed Actions [161] (ML for Systems, NeurIPS 2019) – Effective network congestion control strategies are key to keeping the Internet (or any large computer network) operational. Network congestion control has been dominated by hand-crafted heuristics for decades. Recently, ReinforcementLearning (RL) has emerged as an alternative to automatically optimize such control strategies. Research so far has primarily considered RL interfaces which block the sender while an agent considers its next action. This is largely an artifact of building on top of frameworks designed for RL in games (e.g. OpenAI Gym). However, this does not translate to real-world networking environments, where a network sender waiting on a policy without sending data leads to under-utilization of bandwidth. We instead propose to formulate congestion control with an asynchronous RL agent that handles delayed actions. We present MVFST-RL, a scalable framework for congestion control in the QUIC transport protocol that leverages state-of-the-art in asynchronous RL training with off-policy correction. We analyze modeling improvements to mitigate the deviation from Markovian dynamics, and evaluate our method on emulated networks from the Pantheon benchmark platform. The source code is publicly available at https://github.com/facebookresearch/mvfst-rl.

WordCraft: An Environment for Benchmarking Commonsense Agents [78] (Language in Reinforcement Learning, ICML 2020) – The ability to quickly solve a wide range of real-world tasks requires a commonsense understanding of the world. Yet, how to best extract such knowledge from natural language corpora and integrate it with reinforcement learning (RL) agents remains an open challenge. This is partly due to the lack of lightweight simulation environments that sufficiently reflect the semantics of the real world and provide knowledge sources grounded with respect to observations in an RL environment. To better enable research on agents making use of commonsense knowledge, we propose WordCraft, an RL environment based on Little Alchemy 2. This lightweight environment is fast to run and built

upon entities and relations inspired by real-world semantics. We evaluate several representation learning methods on this new benchmark and propose a new method for integrating knowledge graphs with an RL agent.

Multitask Soft Option Learning [76] (UAI 2020) – We present Multitask Soft Option Learning (MSOL), a hierarchical multitask framework based on Planning as Inference. MSOL extends the concept of options, using separate variational posteriors for each task, regularized by a shared prior. This "soft" version of options avoids several instabilities during training in a multitask setting, and provides a natural way to learn both intra-option policies and their terminations. Furthermore, it allows fine-tuning of options for new tasks without forgetting their learned policies, leading to faster training without reducing the expressiveness of the hierarchical policy. We demonstrate empirically that MSOL significantly outperforms both hierarchical and flat transfer-learning baselines.

Lessons from Reinforcement Learning for Biological Representations of Space [119] (Vision Research, vol. 174) – Neuroscientists postulate 3D representations in the brain in a variety of different coordinate frames (e.g. 'head-centred', 'hand-centred' and 'world-based'). Recent advances in reinforcement learning demonstrate a quite different approach that may provide a more promising model for biological representations underlying spatial perception and navigation. In this paper, we focus on reinforcement learning methods that reward an agent for arriving at a target image without any attempt to build up a 3D 'map'. We test the ability of this type of representation to support geometrically consistent spatial tasks such as interpolating between learned locations using decoding of feature vectors. We introduce a hand-crafted representation that has, by design, a high degree of geometric consistency and demonstrate that, in this case, information about the persistence of features as the camera translates (e.g. distant features persist) can improve performance on the geometric tasks. These examples avoid Cartesian (in this case, 2D) representations of space. Non-Cartesian, learned representations provide an important stimulus in neuroscience to the search for alternatives to a 'cognitive map'.

Simulation-Based Inference for Global Health Decisions [154] (ArXiv 2020) – The COVID-19 pandemic has highlighted the importance of in-silico epidemiological modelling in predicting the dynamics of infectious diseases to inform health policy and decision makers about suitable prevention and containment strategies. Work in this setting involves solving challenging inference and control problems in individual-based models of ever increasing complexity. Here we discuss recent breakthroughs in machine learning, specifically in simulation-based inference, and explore its potential as a novel venue for model calibration to support the design and evaluation of public health interventions. To further stimulate research, we are developing software interfaces that turn two cornerstone COVID-19 and malaria epidemiology models COVID-sim, (https://github.com/mrc-ide/covid-sim/) and OpenMalaria (https://github.com/SwissTPH/openmalaria) into probabilistic programs, enabling efficient interpretable Bayesian inference within those simulators.

2 Background

This chapter provides an overview of the required background information and formalisms for reading through the rest of this thesis. Firstly, we introduce Reinforcement Learning and its modern variant, Deep Reinforcement Learning, in Sections 2.1 and 2.2. Secondly, we look at how Reinforcement Learning changes when applied to multi-agent settings, in Section 2.3, as some modern variants of such algorithms will be discussed in Chapter 3. Finally, we review common RL environments used in the RL literature, so as to facilitate comparisons with the ones presented in this thesis.

Note that concepts that are essential only for specific chapters – e.g., Value Iteration Networks for Chapter 4 – are directly introduced into their relevant chapters, however the literature mentioned in the present chapter largely covers the topics introduced by this thesis in what we believe to be sufficient detail.

2.1 Reinforcement Learning

The general (and basic) Reinforcement Learning formulation is made of an agent interacting with a Markov Decision Process (MDP) (S, A, P, R, γ), where S is the state space of the environment and the agent, A is the set of actions, $P(s_{t+1}|s_t, a_t)$ is the transition dynamics of the environment, $R(s, a)$ is the reward function, and $\gamma \in [0, 1)$ is the discount factor. At time t, the agent in state s_t is allowed to take actions $a_t \in A$, according to its policy $\pi(a_t|s_t)$, after which the state transitions to s_{t+1} according to P, giving the agent a reward $r_t = R(s_t, a_t)$. Overall, the agent's goal is take actions such as to maximise the expected discounted return:

$$R_t = \sum_{t=0}^{\infty} \gamma^t r_t,$$

that is to learn the optimal policy π^* that maximises this return:

$$\begin{aligned} J(\pi) &= \mathbb{E}_{s_0 \sim p(s_0), a_0 \sim \pi(a_0|s_0), s_1 \sim p(s_1), a_1 \sim \pi(a_1|s_1), \ldots}[R_t] \\ &= \mathbb{E}_{s_0, a_0, s_1, a_1, \ldots \sim P, \pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)], \\ \pi^* &= \underset{a_t \in A}{\operatorname{argmax}}\, J(\pi), \end{aligned}$$

assuming an infinite planning horizon. This is an extremely difficult problem in the general case because a priori the agent does not know the dynamics nor the reward function of the MDP, and can only sample transitions by interacting with it, usually in some form of episodic manner without any control over initial states. Readers are particularly invited to check the thorough discussion and overview of the history and development of this framework, and its challenges, found in [170].

Looking at this from a modern machine learning perspective, a subject that nowadays is mostly concerned with learning functions from data, solving the Reinforcement Learning problem equals to learning the parameters θ of the optimal policy π^*_θ, which can be done in a variety of manners. A first straightforward approach is to directly optimise π with respect to the objective above, which is known as *policy-based* learning. A second, possibly less straightforward, approach is to estimate proxy functions from data that can be used to derive the optimal policy – these are often categorised as *value-based* learning (or separately, *model-based*, which we will not discuss in this thesis). Value-based methods usually focus on learning a state-action value function $Q^\pi(s, a)$ (or a value function $V^\pi(s)$), which summarises the return of taking action a from state s, assuming that policy π is followed afterwards:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}, a_{t+1}, \ldots \sim P, \pi | s_t, a_t}[\sum_{t=0}^{\infty} \gamma^{t'-t} r(s'_t, a'_t)].$$

Once the optimal Q-function Q^* is learnt, the optimal policy can be directly derived as:

$$\pi^*(a_t|s_t) = \underset{a_t \in A}{\operatorname{argmax}}\, Q^*(s_t, a).$$

It is very important to note that in reasonable RL problems, the quality of π, Q^π, and other employed functions highly depends on the quality of learnt dynamics and reward functions.

2.2 Deep Reinforcement Learning

The introduction of flexible and powerful function approximators in the form of deep neural networks[103] has greatly expanded and improved the scope of applications available to study and improve Reinforcement Learning. In less than a decade, RL went from being a relatively niche learning framework, to solving real problems on robots. These successes are definitely in large part thanks to the increase of compute and the "unreasonable effectiveness" of Deep Learning, but they are also due to massive efforts in translating a tabular RL algorithms into powerful methods for training these models.

2.2.1 Value-based methods

Above we introduced Q-functions as functions that are relatively straightforward to optimise, while being good proxies for policies. In principle, Q-functions can be evaluated for any policy π, but the optimal $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ obeys Bellman optimality to form the central system for what we call *Q-learning*:

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_a Q^*(s',a')].$$

One of the first RL methods to appropriately employ Deep Neural Networks to deal with high dimensional state spaces was *Deep Q-Network* (DQN) [117] (Figure 2.1), which used a network composed of convnets, fully-connected layers, and some simple non-linear activation functions to represent a Q-function as $Q(s,a;\theta)$ with θ being the network parameters. A DQN is optimised by minimising the Bellman error, i.e.

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right],$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$ is the target for iteration i, and θ_{i-1} are the parameters from the previous learning iteration, held fixed when optimising $L_i(\theta_i)$. To stabilise the process of training these models, DQN introduced two new ideas to the standard Q-learning optimisation process: (a) an *experience replay buffer* used to collect and maintain a history of past traces, and use it as dataset of episodic traces to sample mini-batches for network updates; and (b) the parameters of the *target network* θ_{i-1} are kept constant for a number of iterations to produce more stable gradients wrt. the parameters θ_i of the online network.

By employing *recurrent neural networks* (RNNs) and *LSTMs* (Long Short-Term Memory) to represent the Q-function, the DQN model can be easily extended to

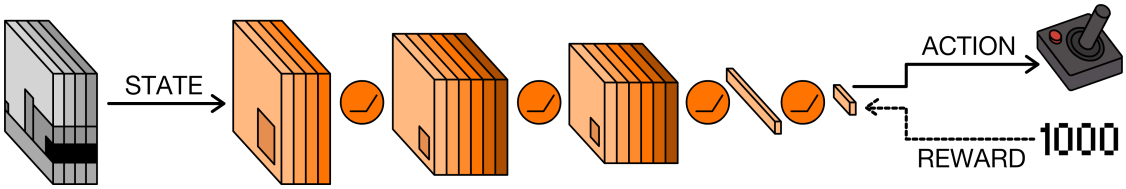


Figure 2.1: Visualisation of the Deep Q-Network architecture employed in [117]. The network takes the state, which is a vector of frames from the Arcade Learning Environment shifted to greyscale and normalised, and puts them through convolutional and fully connected layers using relu nonlinearities in between. The final layer outputs a discrete action corresponding to one of the available actions, which the environment takes and uses to do a step. DQN uses then the the difference between the score obtained in the new step and the score from the previous time step as the reward, which is used to generate the gradients to update the estimate of Q through backpropagation. Taken from [6].

deal with partial observability[68]: irather than directly approximating $Q(s, a)$, a recurrent DQN approximates $Q(\theta, a)$ with an RNN, maintaining an internal state over time from the observation history θ, and adding the hidden state of the network as input h_{t-1} and output h_t to the model. The gradient steps are then automatically backpropagated through time via the process of unrolling the recorded hidden states.

2.2.2 Policy-based methods

A second (and perhaps simpler) system for optimising a policy is known as *Policy Gradients*, which is a type of policy-based approaches. They are considered to be one of the most direct ways of finding the optimal policy, since the optimisation process does not involve surrogate objectives (such as the Value function system discussed above), and gradient descent / ascent is simply applied with respective to the RL objective $J(\pi)$[46].

One of the most basic forms of policy gradient is REINFORCE [200], in which the gradient is defined as::

$$g = \mathbb{E}_{s_{0:\infty}, a_{0:\infty}} \left[\sum_{t=0}^{T} R_t \nabla_{\theta^\pi} \log \pi(a_t | s_t) \right].$$

Due to its simplicity, REINFORCE is commonly employed as a way to make non-differentiable functions actually differentiable (and thus enable end-to-end training via backpropagation); however, it is a relatively weak policy estimator compared to other existing approaches in RL.

A better alternative is *Actor-Critic* (AC) [171, 95, 155], in which the *actor*, i.e., the policy, is trained by following a gradient that depends on a *critic*, which usually estimates a value function. In particular, R_t is replaced by any expression equivalent

to $Q(s_t, a_t) - b(s_t)$, where $b(s_t)$ is a baseline designed to reduce variance [196]. A common choice is $b(s_t) = V(s_t)$, in which case R_t is replaced by $A(s_t, a_t)$. Another option is to replace R_t with the *temporal difference* (TD) error $r_t + \gamma V(s_{t+1}) - V(s)$, which is an unbiased estimate of $A(s_t, a_t)$. In practice, the gradient must be estimated from trajectories sampled from the environment, and the (action-)value functions must be estimated with function approximators. Consequently, the bias and variance of the gradient estimate depends strongly on the exact choice of estimator [95].

Similarly to recurrent DQNs, AC methods can also generally be directly adapted to partially observable settings by using a recurrent network to represent both (or either) the actor and the critic, and conditioning them on the action-observation history θ.

2.3 Multi-Agent Reinforcement Learning

In this thesis we consider a multi-agent setting in which n agents identified by $a \in A \equiv \{1, ..., n\}$ participate in a stochastic game, G, described by a tuple $G = \langle S, U, P, r, Z, O, n, \gamma \rangle$, by choosing sequential actions. Similarly to the single-agent setting, the environment occupies states $s \in S$, in which, at every time step, each agent takes an action $u_a \in U$, forming a joint action $\mathbf{u} \in \mathbf{U} \equiv U^n$.[1] Joint quantities over (usually) the acting agent are set in bold (e.g., $\mathbf{u}$), and joint quantities over any other agent with superscript $-a$ (e.g. $\mathbf{u}^{-a}$). As indeed previously seen, $\gamma \in [0, 1]$ again is a discount factor. Actions applied to the environment induce changes in the environment with state transition probabilities defined by $P(s'|s, \mathbf{u}) : S \times \mathbf{U} \times S \to \mathbb{R}$; similarly, the reward function specifies a reward conditioned on each agent, $r(s, \mathbf{u}, a) : S \times \mathbf{U} \times A \to \mathbb{R}$.

In fully observable settings it is generally possible to learn a unique, centralised policy, $\pi(\mathbf{u}|s_t)$, that given environment states returns a distribution over the joint action space:

$$\pi(\mathbf{u}|s_t) : \mathbf{U} \times S \to [0, 1].$$

However this makes for a challenging approach due to two primary issues: firstly, the join action space $\mathbf{U}$ quickly blows up exponentially with increasing number of agents, and secondly centralised control is rarely a possibility in real-world

[1] Note that this particular notation, a for agents and u for actions, is relatively common in the multi-agent literature, stemming from pre-RL work. We will only use this nomenclature in Chapter 3, as it clashes with most existing body of work in RL.

applications: in most cases agents need to act on separate, local observations. Thus, for the reminder of this thesis we focus on *decentralised control*, where each agent has a policy $\pi^a(\mathbf{u}^a|s_t)$ that is independent of the others and that maps to its own agent-dependent probability distribution. This enables to factorize the joint action space as a product of the individual action spaces,

$$P(\mathbf{u}|s_t) = \prod_a \pi^a(u^a|s_t)$$

,

and allows us to reason on the agent-centric action space and condition on local agent observations. In the general case, it means that agents act based on partial observability; that is these observations $o_t^a \in Z$ are governed by an observation function $O(s,a) : S \times A \to Z$. For notational simplicity, this observation function is deterministic, i.e., we model ony perceptual aliasing and not noise. However, extending this formalisation to noisy observation functions is straightforward. Each agent a conditions its behaviour on its own action-observation history $\tau_a \in T \equiv (Z \times U)^*$, according to its policy $\pi_a(u_a|\tau_a) : T \times U \to [0,1]$. After each transition, the action u_a and new observation $O(s,a)$ are added to τ_a, forming τ_a'. We denote joint quantities over agents in bold, and joint quantities over agents other than a with the subscript $-a$, so that, e.g., $\mathbf{u} = [u_a, \mathbf{u}_{-a}]$. In this thesis we also assume that in cases of fully cooperative settings agents share the same reward function $r(s,\mathbf{u}) : S \times \mathbf{U} \to \mathbb{R}$ in a *general-sum* setting.

2.3.1 Centralised Training, Decentralised Execution

In real world settings, training and execution of policies are often two very disjointed processes. Training is commonly carried out on a simulator, learning policies that are later deployed on (more or less) physical actors. This constraint enables us to focus on settings where agents need to take actions based on their local observation only during *decentralised execution*, whilst during *centralised trainig* they are able to reason, condition, and generally gain access to any type of accessible agent-set-wise information (such as belief state, communication data, training state, and so forth). This is a standard approach for solving Dec-POMDPs[17], and provides a fantastic avenue for learning deep multi-agent policies using RL. We will see this concept employed heavily in Chapter 3, as it'll form the framework around which we set up both methods and experiments.

2.3.2 Foundational algorithms

Multi-agent RL has a rich history [26, 203] but has mostly focused on tabular settings and simple environments. The most commonly used method is independent Q-learning (IQL) [179, 157, 206], perhaps the simplest approach to transitioning from single-agent to multi-agent RL: each agent learns its own Q-function that conditions only on the state and its own action. IQL is appealing because it avoids the scalability problems of trying to learn a joint Q-function that conditions on $\mathbf{u}$, since as we previous mentioned, $|\mathbf{U}|$ grows exponentially in the number of agents. It is also naturally suited to partially observable settings, since, by construction, it learns decentralised policies in which each agent's action conditions only on its own observations, thus enabling to further consider using directly its action-observation history $Q_a(\tau_a, u_a)$. When using deep policies, this can be achieved by having each agent perform DQN using a recurrent neural network trained on its own observations and actions.

However, IQL introduces a key problem: the environment becomes nonstationary from the point of view each agent, as it contains other agents who are themselves learning, ruling out any convergence guarantees. On the one hand, the conventional wisdom is that this problem is not severe in practice, and substantial empirical results have demonstrated success with IQL [114]. On the other hand, such results do not involve deep learning.

Lauer and Riedmiller [101] propose a variation of distributed Q-learning, a coordination-free method. However, they also argue that the simple estimation of the value function in the standard model-free fashion is not enough to solve multi-agent problems, and coordination through means such as communication [113] is required to ground separate observations to the full state function.

More recent work tries to leverage deep learning in multi-agent RL, mostly as a means to reason about the emergence of inter-agent communication. Tampuu et al. [177] apply a framework that combines DQN with independent Q-learning to two-player pong. Foerster et al. [53] propose DIAL, an end-to-end differentiable architecture that allows agents to learn to communicate and has since been used by Jorge et al. [82] in a similar setting. Sukhbaatar et al. [165] also show that it is possible to learn to communicate by backpropagation. Leibo et al. [105] analyse the emergence of cooperation and defection when using multi-agent RL in mixed-cooperation environments such as the wolfpack problem. He et al. [70] address multi-agent learning by explicitly marginalising the opponents' strategy using a mixture of experts in the DQN.

2.3.2.1 Non-stationarity in Multi-Agent settings

As discussed earlier, many popular forms of deep RL relies heavily on experience replay and the combination of experience replay with IQL appears to be problematic: the nonstationarity introduced by IQL means that there exsts a mismatch between the dynamics that generated the data in the agent's replay memory and the current dynamics in which it is learning. While IQL without a replay memory can learn well despite nonstationarity so long as each agent is able to gradually track the other agents' policies, that seems hopeless with a replay memory constantly confusing the agent with obsolete experience. Section 3.2 will propose methods to tackle exactly this issue.

Methods like hyper Q-learning [184], also discussed in Section 3.2, and AWESOME [39] try to tackle nonstationarity by tracking and conditioning each agent's learning process on their teammates' current policy, while Da Silva et al. [41] propose detecting and tracking different classes of traces on which to condition policy learning. Kok and Vlassis [94] show that coordination can be learnt by estimating a global Q-function in the classical distributed setting supplemented with a coordination graph. In general, these techniques have so far not successfully been scaled to high-dimensional state spaces.

Broadly related are methods that attempt to allow for faster convergence of policy networks such as prioritised experience replay [152], a version of the standard replay memory that biases the sampling distribution based on the TD error. However, this method does not account for nonstationary environments and does not take into account the unique properties of the multi-agent setting which, as we'll see in Section 3.2, provide both both unique challenges as well as structure for solving such issues in a relatively simple and straightforward manner.

Wang et al. [193] describe an importance sampling method for using off-policy experience in a single-agent actor-critic algorithm. However, to calculate policy-gradients, the importance ratios become products over potentially lengthy trajectories, introducing high variance that must be partially compensated for by truncation. By contrast, in Section 3.3 we address *off-environment* learning and show that the multi-agent structure results in importance ratios that are simply products over the agents' policies.

2.3.2.2 Multi-Agent policy gradients

Since in Section 3.3 we develop a new multi-agent policy gradient, it is important to clarify some details about how we apply policy gradients in a multi-agent setting. In particular, we generally train critics $f^c(\cdot, \theta^c)$ on-policy to estimate either Q or V, using a variant of TD(λ) [169] adapted for use with deep neural networks. TD(λ) uses a mixture of n-step returns $G_t^{(n)} = \sum_{l=1}^{n} \gamma^{l-1} r_{t+l} + \gamma^n f^c(\cdot_{t+n}, \theta^c)$. In particular, the critic parameters θ^c are updated by minibatch gradient descent to minimise the following loss:

$$\mathcal{L}_t(\theta^c) = (y^{(\lambda)} - f^c(\cdot_t, \theta^c))^2,$$

where $y^{(\lambda)} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$, and the n-step returns $G_t^{(n)}$ are calculated with bootstrapped values estimated by a *target network* [117] with parameters copied periodically from θ^c.

Gupta et al. [62] investigate actor-critic methods for decentralised execution with centralised training. However, in their methods both the actors and the critic condition on local, per-agent, observations and actions, and multi-agent credit assignment is addressed only with hand-crafted local rewards.

[108] concurrently propose a multi-agent policy-gradient algorithm using centralised critics. Their approach does not address multi-agent credit assignment. Unlike our work, it learns a separate centralised critic for each agent and is applied to competitive environments with continuous action spaces.

Also related is work on the emergence of communication between agents, learned by gradient descent [43, 118, 102, 53, 165]. In this line of work, passing gradients between agents during training and sharing parameters are two common ways to take advantage of centralised training. However, these methods do not allow for extra state information to be used during learning and do not address the multi-agent credit assignment problem.

Finally, our work builds directly on the idea of *difference rewards* [202]. The particular relationship with our work is discussed in Section 3.3.2.

2.4 Evaluating Deep Reinforcement Learning

Deep RL is fundamentally driven by empirircal machine learning work, and as such it strongly relies on quality benchmarks to evaluate, rank, and compare methods, algorithms, and models across problems and related state-of-the-art solutions. Benchmarks also function as diagnostic tools, enabling researchers to

look into behaviour of their agents in known settings, which is particularly useful when developing new methods and/or moving to novel problems. Before the Deep Learning era, most common RL benchmarks consituted of simple control problems, small grid-world environments, low-dimensional matrix games, and so forth [170], some of which are still employed today as *unit tests*. The past decade however has seen a (messy) explosion of new high-dimensional, often complex, environments generally correlated with the introduction of new methods poking around issues arising from using Neural Networks and more complex end-to-end architectures.

Arguably, the Atari Learning Environment (ALE) [13] has generated the most impact in the past decade. ALE is a suite of environmnets based on Atari video games, all with a common interface. Agents get access to incoming pixel (or RAM) observations, and can act using a set of actions mapped to a simulated real-life controller. The environments range in difficulty by various metrics (*Pong* is relatively a straightforward visual and control problem to solve, whilst *Montezuma's Revenge* is an extremely hard exploration problem with lots of structure in the state). Incoming successors that provide a similar interface to later generations (of naturally more complex) video games include Retro [122] and Procgen [36].

The community has also produced a number of modern gridworlds that improve upon the classic ones by providing some degree of complexity in the observation, a more varied range of more or less complex tasks, and interfaces for quickly producing task variations. Popular examples include Mazebase [166], minigrid [31], as well as the perhaps more control-focused bsuite [128].

MuJoCo [186], PyBullet [40], and Brax [57] are some of the most popular environments in the current generation of control benchmarks. These environments are backed by engines that enable constrained control and simulation of physical interactions between multi-body systems, with various degrees of physical resolution, ranging all the way from simple linear mechanics to full-body robotic sims. They generally come wit an ever-growing suite of high-level control tasks, such as locomotion, path planning, and object manipulation, and more or less ergonomic interfaces for creating similarly-scoped new ones. However, a major drawback is that physics interaction simulations tend to be grow computationally expensive with respect to both the simulated dynamics and the complexity of the task.

On the higher end of the complexity spectrum lie *Embodied AI* environments such as MineCraft[80, 64], Habitat[151], DM lab [12], the Unity ML-Agents [83], and others. These provide a semi-realistic embodied simulation of human-like tasks with some varying degrees of physics and a wide range of task complexity, all the way from simple 3D grid-world navigation tasks to open-ended, lifelong learning settings.

Most of these environments are at this point in time thought to be 'unsolved', and require significant investment in both compute and research time to improve state-of-the-art performance. Some of these benchmarks have furthermore evolved into being *AI Challenges*. These lose some of their function as tunable debugging tool, and become more specialised to provide a unified set of hard learning problems that the community can target for a long timespan. Older ones include backgammon [183], chess, go[153] and other similar board games, and more recent examples are StarCraft[173, 191] and Dota 2[16]. In Chapter 5, we will introduce an environment that we believe can function as the next RL grand-challenge while minimising prototyping and usability issues that tend to plague AI grand-challenges.

2.4.1 Multi-Agent Environments

The community over time spent a lot of work into designing environments to test and develop Multi-Agent RL methods. However, not many of these focused on providing a qualitatively challenging environment that would provide together elements of partial observability, challenging dynamics, and high-dimensional observation spaces.

Stone et al. [164] presented Keepaway soccer, a domain built on the RoboCup soccer simulator [93], a 2D simulation of a football environment with simplified physics, where the main task consists of keeping a ball within a pre-defined area where agents in teams can reach, steal, and pass the ball, providing a simplified setup for studying cooperative MARL. A more modern version of this task has recently been developed by [97] as the Google Research Football, which significantly scales the problem to encompass a full football game simulation. This domain was later extended to the Half Field Offense task [88, 66], which increases the difficulty of the problem by requiring the agents to not only keep the ball within bounds but also to score a goal. Neither task scales well in difficulty with the number of agents, as most agents need to do little coordination. There is also a lack of interesting environment dynamics beyond the simple 2D physics nor good reward signals, thus reducing the impact of the environment as a testbed.

Multiple gridworld-like environments have also been explored. Lowe et al. [109] released a set of simple grid-world like environments for multi-agent RL alongside an implementation of MADDPG, featuring a mix of competitive and cooperative tasks focused on shared communication and low level continuous control. Leibo et al. [106] show several mixed-cooperative Markov environment focused on testing social dilemmas, however, they did not release an implementation to further explore the tasks. Yang et al. [204] and Zheng et al. [208] present a framework for creating gridworlds focuses on many-agents tasks, where the number

of agents ranges from the hundreds to the millions. This work, however, focuses on testing for emergent behaviour, since environment dynamics and control space need to remain relatively simple for the tasks to be tractable. Resnick et al. [144] propose a multi-agent environment based on the game *Bomberman*, encompassing a series of cooperative and adversarial tasks meant to provide a more challenging set of tasks with a relatively straightforward 2D state observation and simple grid-world-like action spaces.

2.4.2 StarCraft Micromanagement

This thesis is largely based on tackling the *unit micromanagement* task in StarCraft. StarCraft is one of the most complex real time strategy (RTS) games ever created. a categority of video games that has historically seen huge interest from the planning community [25, 3, 127].

These games simulate the control of a large number of units, each with a unique set of abilities, and a large number of simultaneous objectives of varying levels of complexity, usually in an adversarial 2-player setting in a 2D map. The goal of the player is to manage and balance resource collection against in-episode tech and economy development, unit creation, map control, all in real-time (i.e. without turns). Ultimately, a player either gets overwhelmed or completely destroyed by the opponent. These games have durative actions with complex dynamics, that can simultaneous target multiple units, and very often players are forced to make many decisions under extreme partial observability due to *fog of war*, which prevents them from viewing parts of the map that aren't directly observed by their own units. Additionally, players have a large amount of different unit types to play in different combinations – think being able to recruit different types of pawns and rooks in chess with slightly different abilities, strengths, and weaknesses in a rock-paper-scissors fashion. A considerable part of the strategical and the tactical difficulty arises from figuring out when, where, and how to deploy armies.

Learning to play StarCraft games has been indeed investigated in several communities: work ranging from evolutionary algorithms to tabular RL applied has shown that the game is an excellent testbed for both modelling and planning [127]. With the release of TorchCraft [174] and SC2LE [192], interfaces to respectively *StarCraft: BroodWar* and *StarCraft II*, more recent examples of single agent controllers that employ Deep RL have also made significant strides towards conquering this environment Usunier et al. [188], Pang et al. [130], Sun et al. [167], and Vinyals et al. [190], albeit often with outstanding computational requirements. In this thesis we will be employing both interfaces to develop and test our methods.

To summarise, most previous applications of RL to StarCraft micromanagement use a centralised controller, with access to the full state, and control of all units, although the architecture of the controllers exploits the multi-agent nature of the problem. In Chapter 3 we will develop a framework for viewing the micromanagement problem in a multi-agent fashion, and develop a Multi-Agent RL environment to function as a long-term benchmark for a variety of problems.

Furthermore, in Chapter 4 we will investigate how we can use StarCraft micromanagement to make improvements in learning policies for path planning using Reinforcement Learning, avoiding relying on overly-constrained and un-generalisable grid-worlds.

3

Deep Multi-Agent Reinforcement Learning

3.1 Decentralised StarCraft Micromanagement

As mentioned in Section 2.4.2, StarCraft is an example of a complex, stochastic environment whose dynamics cannot easily be simulated. This differs from standard multi-agent settings such as Packet World [198] and simulated RoboCup [67], where often entire episodes can be fully replayed and analysed. This difficulty is typical of real-world problems, and is well suited to the model-free approaches common in deep RL. In StarCraft, *micromanagement* refers to the subtask of controlling single or grouped units to move them around the map and fight enemy units. In our multi-agent variant of StarCraft micromanagement, the centralised player is replaced by a set of agents, each assigned to one unit on the map. Each agent must select from a restricted set of durative actions: `move[direction]`, `attack[enemy_id]`, `stop`, and `noop`. During an episode, each unit is identified by a positive integer initialised on the first time-step.

Reward is the sum of the damage inflicted against opponent units during that timestep, with an additional terminal reward equal to the sum of the health of all units on the team. This is a variation of a naturally arising battle signal, comparable with the one used by Usunier et al. [188]. In each scenario, after a few timesteps after the agents are spawned, they are attacked by opponent units, where the time spent between the first state and the attack depends on the overall units positions and their types (some are faster than others). Opponents are controlled by the game AI, which is set to attack all the time. All of these scenarios require the agents to coordinate their movements to get the opponents into their range of fire

with good positioning, and to focus their firing on each enemy unit so as to destroy them more quickly. Skilled human StarCraft players can typically solve these tasks.

To create a more challenging benchmark that is meaningfully decentralised, we impose a restricted field of view on the agents, equal to the firing range of ranged units' weapons, shown in Figure 3.1. This departure from the standard setup for centralised StarCraft control has three effects.

Figure 3.1: Starting position with example local field of view for the 2d_3z map.

First, it introduces significant partial observability. Second, it means units can only attack when they are in range of enemies, removing access to the StarCraft macro-actions. Third, agents cannot distinguish between enemies who are dead and those who are out of range and so can issue invalid attack commands at such enemies, which results in no action being taken. This substantially increases the average size of the action space, which in turn increases the difficulty of both exploration and control.

Under these difficult conditions, scenarios with even relatively small numbers of units become much harder to solve. As seen in Table 3.1, we compare against a simple hand-coded heuristic that instructs the agents to run forwards into range and then focus their fire, attacking each enemy in turn until it dies. This heuristic achieves a 98% win rate on 5m with a full field of view, but only 66% in our setting. To perform well in this task, the agents must learn to cooperate by positioning properly and focussing their fire, while remembering which enemy and ally units are alive or out of view.

All agents receive the same global reward at each time step, equal to the sum of damage inflicted on the opponent units minus half the damage taken. Killing

an opponent generates a reward of 10 points, and winning the game generates a reward equal to the team's remaining total health plus 200. This damage-based reward signal is comparable to that used by Usunier et al. [188]. Unlike [135], our approach does not require estimating local rewards.

Scenarios are usually defined by the quantity of controlled units X vs opponent units Y, with the following nomenclature: `$typeX_$typeY`. Type of units could be *Terran Marines (m)*, ground units with a fixed range of fire about the length of four stacked units, *zealots (z)*, flying units with a fixed range of fire, *dragoons (d)*, and so forth.

3.2 Beyond Independent Q-learning

When all agents observe the true state, one can model a cooperative multi-agent system as a single meta-agent. However, the size of this meta-agent's action space grows exponentially in the number of agents, and the model becomes generally inapplicable when each agent receives different observations that may not disambiguate the state. In such case, decentralised policies must be learned.

A popular framework is *independent Q-learning* (IQL) [179], in which each agent independently learns its own policy, treating other agents as part of the environment. While IQL avoids the scalability problems of centralised learning, it introduces a new problem: the environment becomes nonstationary from the point of view of each agent, as it contains other agents who are themselves learning, ruling out any convergence guarantees. Fortunately, substantial empirical evidence has shown that IQL often works well in practice [114].

As mentioned in Chapter 2, the use of deep neural networks has dramatically improved the scalability of single-agent RL [117]. However, one element key to the success of most such approaches is the reliance on an *experience replay memory*, which stores experience tuples that are sampled during training. Experience replay not only helps to stabilise the training of a deep neural network, it also improves sample efficiency by repeatedly reusing experience tuples. Unfortunately, the combination of experience replay with IQL appears to be problematic: the nonstationarity introduced by IQL means that the dynamics that generated the data in the agent's replay memory no longer reflect the current dynamics in which it is learning. While IQL without a replay memory can learn well despite nonstationarity so long as each agent is able to gradually track the other agents' policies, that seems hopeless with a replay memory constantly confusing the agent with obsolete experience.

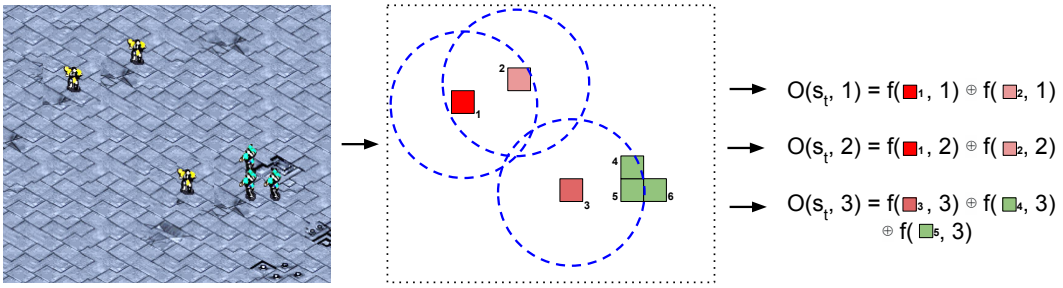


Figure 3.2: An example of the observations obtained by all agents at each time step t. The function f provides a set of features for each unit in the agent's field of view, which are concatenated. The feature set is {`distance`, `relative x`, `relative y`, `health points`, `weapon cooldown`}. Each quantity is normalised by its maximum possible value.

To avoid this problem, previous work on deep multi-agent RL has limited the use of experience replay to short, recent buffers [105] or simply disabled replay altogether [53]. However, these workarounds limit the sample efficiency and threaten the stability of multi-agent RL. Consequently, the incompatibility of experience replay with IQL is emerging as a key stumbling block to scaling deep multi-agent RL to complex tasks.

Here we propose two approaches for effectively incorporating experience replay into multi-agent RL. The first approach interprets the experience in the replay memory as *off-environment* data [35]. By augmenting each tuple in the replay memory with the probability of the joint action in that tuple, according to the policies in use at that time, we can compute an importance sampling correction when the tuple is later sampled for training. Since older data tends to generate lower importance weights, this approach naturally decays data as it becomes obsolete, preventing the confusion that a nonstationary replay memory would otherwise create.

The second approach is inspired by *hyper Q-learning* [184], which avoids the nonstationarity of IQL by having each agent learn a policy that conditions on an estimate of the other agents' policies inferred from observing their behaviour. While it may seem hopeless to learn Q-functions in this much larger space, especially when each agent's policy is a deep neural network, we show that doing so is feasible as each agent need only condition on a low-dimensional *fingerprint* that is sufficient to disambiguate where in the replay memory an experience tuple was sampled from.

3.2.1 Multi-Agent Importance Sampling

We can address the non-stationarity present in IQL by developing an importance sampling scheme for the multi-agent setting. Just as an RL agent can use importance sampling to learn *off-policy* from data gathered when its own policy was different,

so it can learn *off-environment* [35] from data gathered in a different environment. Since IQL treats other agents' policies as part of the environment, off-environment importance sampling can be used to stabilise experience replay. In particular, since we know the policies of the agents at each stage of training, we know exactly the manner in which the environment is changing, and can thereby correct for it with importance weighting, as follows. We consider first a fully-observable multi-agent setting. If the Q-functions can condition directly on the true state s, we can write the Bellman optimality equation for a single agent given the policies of all other agents:

$$Q_a^*(s, u_a|\boldsymbol{\pi}_{-a}) = \sum_{\mathbf{u}_{-a}} \boldsymbol{\pi}_{-a}(\mathbf{u}_{-a}|s)\Big[r(s, u_a, \mathbf{u}_{-a}) + \gamma \sum_{s'} P(s'|s, u_a, \mathbf{u}_{-a}) \max_{u'_a} Q_a^*(s', u'_a)\Big].$$

The nonstationary component of this equation is $\boldsymbol{\pi}_{-a}(\mathbf{u}_{-a}|s) = \Pi_{i\in -a}\pi_i(u_i|s)$, which changes as the other agents' policies change over time. Therefore, to enable importance sampling, at the time of collection t_c, we record $\boldsymbol{\pi}_{-a}^{t_c}(\mathbf{u}_{-a}|s)$ in the replay memory, forming an augmented transition tuple $\langle s, u_a, r, \pi(\mathbf{u}_{-a}|s), s'\rangle^{(t_c)}$.

At the time of replay t_r, we train off-environment by minimising an importance weighted loss function:

$$\mathcal{L}(\theta) = \sum_{i=1}^{b} \frac{\boldsymbol{\pi}_{-a}^{t_r}(\mathbf{u}_{-a}|s)}{\boldsymbol{\pi}_{-a}^{t_i}(\mathbf{u}_{-a}|s)}[(y_i^{DQN} - Q(s, u; \theta))^2],$$

where t_i is the time of collection of the i-th sample.

The derivation of the non-stationary parts of the Bellman equation in the partially observable multi-agent setting is considerably more complex as the agents' action-observation histories are correlated in a complex fashion that depends on the agents' policies as well as the transition and observation functions.

To make progress, we can define an augmented state space $\hat{s} = \{s, \boldsymbol{\tau}_{-a}\} \in \hat{S} = S \times T^{n-1}$. This state space includes both the original state s and the action-observation history of the other agents $\boldsymbol{\tau}_{-a}$. We also define a corresponding observation function $\hat{O}$ such that $\hat{O}(\hat{s}, a) = O(s, a)$. With these definitions in place, we define a new reward function $\hat{r}(\hat{s}, u) = \sum_{\mathbf{u}_{-a}} \boldsymbol{\pi}_{-a}(\mathbf{u}_{-a}|\boldsymbol{\tau}_{-a})r(s, \mathbf{u})$ and a new transition function,

$$\hat{P}(\hat{s}'|\hat{s}, u) = P(s', \tau'|s, \tau, u) = \sum_{\mathbf{u}_{-a}} \boldsymbol{\pi}_{-a}(\mathbf{u}_{-a}|\boldsymbol{\tau}_{-a})P(s'|s, \mathbf{u})p(\boldsymbol{\tau}'_{-a}|\boldsymbol{\tau}_{-a}, \mathbf{u}_{-a}, s').$$

All other elements of the augmented game $\hat{G}$ are adopted from the original game G. This also includes T, the space of action-observation histories. The augmented game is then specified by $\hat{G} = \langle \hat{S}, U, \hat{P}, \hat{r}, Z, \hat{O}, n, \gamma \rangle$. We can now write a Bellman equation for $\hat{G}$:

$$Q(\tau, u) = \sum_{\hat{s}} p(\hat{s}|\tau) \Bigg[\hat{r}(\hat{s}, u) + \gamma \sum_{\tau', \hat{s}', u'} \hat{P}(\hat{s}'|\hat{s}, u) \pi(u', \tau') p(\tau'|\tau, \hat{s}', u) Q(\tau', u') \Bigg].$$

Substituting back in the definitions of the quantities in $\hat{G}$, we arrive at a Bellman equation of a form similar to (3.2.1), where the righthand side is multiplied by $\boldsymbol{\pi}_{-a}(\mathbf{u}_{-a}|\boldsymbol{\tau}_{-a})$:

$$Q(\tau, u) = \sum_{\hat{s}} p(\hat{s}|\tau) \sum_{\mathbf{u}_{-a}} \boldsymbol{\pi}_{-a}(\mathbf{u}_{-a}|\boldsymbol{\tau}_{-a}) \Bigg[r(s, \mathbf{u}) + \gamma \sum_{\tau', \hat{s}', u'} P(s'|s, \mathbf{u}) p(\boldsymbol{\tau}'_{-a}|\boldsymbol{\tau}_{-a}, \mathbf{u}_{-a}, s') \cdot \pi(u', \tau') p(\tau'|\tau, \hat{s}', u) Q(\tau', u') \Bigg].$$

This construction simply allows us to demonstrate the dependence of the Bellman equation on the same nonstationary term $\boldsymbol{\pi}_{-a}(\mathbf{u}_{-a}|s)$ in the partially-observable case. However, unlike in the fully observable case, the righthand side contains several other terms that indirectly depend on the policies of the other agents and are to the best of our knowledge intractable. Consequently, the importance ratio defined above, $\frac{\boldsymbol{\pi}^{t_r}_{-a}(\mathbf{u}_{-a}|s)}{\boldsymbol{\pi}^{t_i}_{-a}(\mathbf{u}_{-a}|s)}$, is only an approximation in the partially observable setting.

3.2.2 Multi-Agent Fingerprints

While importance sampling provides an unbiased estimate of the true objective, it often yields importance ratios with large and unbounded variance [146]. Truncating or adjusting the importance weights can reduce the variance but introduces bias. Consequently, we propose an alternative method that embraces the nonstationarity of multi-agent problems, rather than correcting for it.

The weakness of IQL is that, by treating other agents as part of the environment, it ignores the fact that such agents' policies are changing over time, rendering its own Q-function nonstationary. This implies that the Q-function could be made stationary if it conditioned on the policies of the other agents. This is exactly the philosophy behind *hyper Q-learning* [184]: each agent's state space is augmented

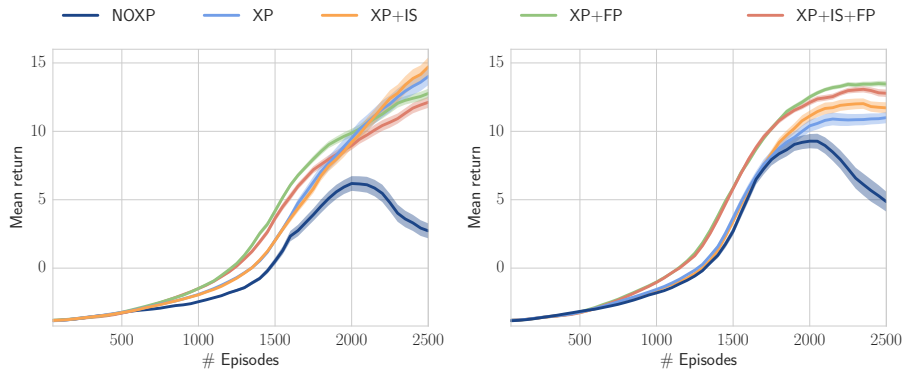
with an estimate of the other agents' policies computed via Bayesian inference. Intuitively, this reduces each agent's learning problem to a standard, single-agent problem in a stationary, but much larger, environment.

The practical difficulty of hyper Q-learning is that it increases the dimensionality of the Q-function, making it potentially infeasible to learn. This problem is exacerbated in deep learning, when the other agents' policies consist of high dimensional deep neural networks. Consider a naive approach to combining hyper Q-learning with deep RL that includes the weights of the other agents' networks, $\boldsymbol{\theta}_{-a}$, in the observation function. The new observation function is then $O'(s) = \{O(s), \boldsymbol{\theta}_{-a}\}$. The agent could in principle then learn a mapping from the weights $\boldsymbol{\theta}_{-a}$, and its own trajectory τ, into expected returns. Clearly, if the other agents are using deep models, then $\boldsymbol{\theta}_{-a}$ is far too large to include as input to the Q-function.

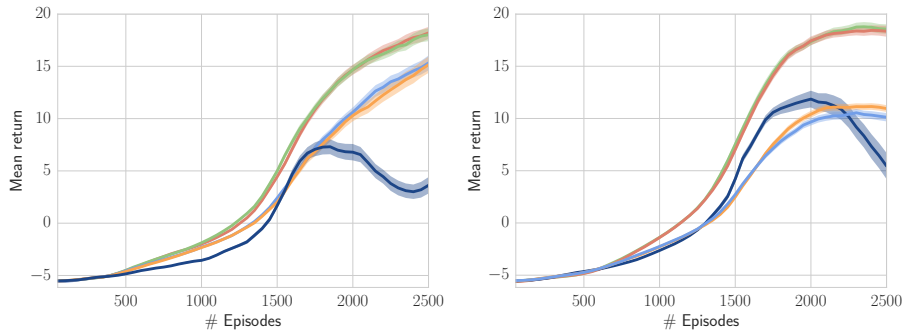
However, a key observation is that, to stabilise experience replay, each agent does not need to be able to condition on any possible $\boldsymbol{\theta}_{-a}$, but only those values of $\boldsymbol{\theta}_{-a}$ that actually occur in its replay memory. The sequence of policies that generated the data in this buffer can be thought of as following a single, one-dimensional trajectory through the high-dimensional policy space. To stabilise experience replay, it should be sufficient if each agent's observations disambiguate where along this trajectory the current training sample originated from.

The question then, is how to design a low-dimensional *fingerprint* that contains this information. Clearly, such a fingerprint must be correlated with the true value of state-action pairs given the other agents' policies. It should typically vary smoothly over training, to allow the model to generalise across experiences in which the other agents execute policies of varying quality as they learn. An obvious candidate for inclusion in the fingerprint is the training iteration number e. One potential challenge is that after policies have converged, this requires the model to fit multiple fingerprints to the same value, making the function somewhat harder to learn and more difficult to generalise from.

Another key factor in the performance of the other agents is the rate of exploration ϵ. Typically, an annealing schedule is set for ϵ such that it varies smoothly throughout training and is quite closely correlated to performance. Therefore, we further augment the input to the Q-function with ϵ, such that the observation function becomes $O'(s) = \{O(s), \epsilon, e\}$. Our results in Section 3.2.3.2 show that even this simple fingerprint is remarkably effective.



(a) 3v3 with recurrent networks **(b)** 3v3 with feed-forward networks



(c) 5v5 with recurrent networks **(d)** 5v5 with feed-forward networks

Figure 3.3: Performance of our methods compared to the two baselines XP and NOXP, for both RNN and FF; (a) and (b) show the 3v3 setting, in which IS and FP are only required with feed-forward networks; (c) and (d) show the 5v5 setting, in which FP clearly improves performance over the baselines, while IS shows a small improvement only in the feedforward setting. Overall, the FP is a more effective method for resolving the nonstationarity and there is no additional benefit from combining IS with FP. Confidence intervals show one standard deviation of the sample mean.

3.2.3 Experiments

We use the recurrent DQN architecture described by Foerster et al. [53] with a few modifications. We do not consider communicating agents, so there are no message connections. As mentioned above, we use two different models: one with a feed-forward model with two fully connected hidden layers, and another with a single-layer GRU. For both models, every hidden layer has 128 neurons.

We linearly anneal ϵ from 1.0 to 0.02 over 1500 episodes, and train the network for $e_{max} = 2500$ training episodes. In the standard training loop, we collect a single episode and add it to the replay memory at each training step. We sample batches of $\frac{30}{n}$ episodes uniformly from the replay memory and train on fully unrolled episodes. In order to reduce the variance of the multi-agent importance weights, we clip them to the interval $[0.01, 2]$. We also normalise the importance weights by the number of agents, by raising them to the power of $\frac{1}{n-1}$. Lastly, we divide the

importance weights by their running average in order to keep the overall learning rate constant. All other hyperparameters are identical to Foerster et al. [53].

The results of our StarCraft experiments, summarised in Figure 3.3. Across all tasks and models, the baseline without experience replay (NOXP) performs poorly. Without the diversity in trajectories provided by experience replay, NOXP overfits to the greedy policy once ϵ becomes small. When exploratory actions do occur, agents visit areas of the state space that have not had their Q-values updated for many iterations, and bootstrap off of values which have become stale or distorted by updates to the Q-function elsewhere. This effect can harm or destabilise the policy. With a recurrent model, performance simply degrades, while in the feed-forward case, it begins to drop significantly later in training. We hypothesise that full trajectories are inherently more diverse than single observations, as they include compounding chances for exploratory actions. Consequently, it is easier to overfit to single observations, and experience replay is more essential for a feed-forward model.

With a naive application of experience replay (XP), the model tries to simultaneously learn a best-response policy to every historical policy of the other agents. Despite the nonstationarity, the stability of experience replay enables XP to outperform NOXP in each case. However, due to limited disambiguating information, the model cannot appropriately account for the impact of any particular policy of the other agents, or keep track of their current policy. The experience replay is therefore used inefficiently, and the model cannot generalise properly from experiences early in training.

3.2.3.1 Importance Sampling

The importance sampling approach (XP+IS) slightly outperforms XP when using feed-forward models. While mathematically sound in the fully observable case, XP+IS is only approximate for our partially observable problem, and runs into practical obstacles. Early in training, the importance weights are relatively well-behaved and have low variance. However, as ϵ drops, the importance ratios become multi-modal with increasing variance. The large majority of importance weights are less than or equal to $\epsilon(1-\epsilon) \approx \epsilon$, so few experiences contribute strongly to learning. In a setting that does not require as strongly deterministic a policy as StarCraft, ϵ could be kept higher and the variance of the importance weights would be lower.

3.2.3.2 Fingerprints

Our results show that the simple fingerprint of adding e and ϵ to the observation (XP+FP) dramatically improves performance for the feed-forward model. This fingerprint provides sufficient disambiguation for the model to track the quality of the other agents' policies over the course of training, and make proper use of the experience buffer. The network still sees a diverse array of input states across which to generalise but is able to modify its predicted value in accordance with the known stage of training.

Figure 3.3 also shows that there is no extra benefit from combining importance sampling with fingerprints (XP+IS+FP). This makes sense given that the two approaches both address the same problem of nonstationarity, albeit in different ways.

Figure 3.4, which shows the estimated value for XP+FS of a single initial state observation with different ϵ inputs, demonstrates that the network learns to smoothly vary its value estimates across different stages of training, correctly associating high values with the low ϵ seen later in training. This approach allows the model to generalise between best responses to different policies of other agents. In effect, a larger dataset is available in this case than when using importance sampling, where most experiences are strongly discounted during training. The fingerprint enables the transfer of learning between diverse historical experiences, which can significantly improve performance.

3.2.3.3 Informative Trajectories

When using recurrent networks, the performance gains of XP+IS and XP+FP are not as large; in the 3v3 task, neither method helps. The reason is that, in StarCraft, the observed trajectories are significantly informative about the state of training, as shown in Figure 3.5a and 3.5b. For example, the agent can observe that it or its allies have taken many seemingly random actions, and infer that the sampled experience comes from early in training. This is a demonstration of the power of recurrent architectures in sequential tasks with partial observability: even without explicit additional information, the network is able to partially disambiguate experiences from different stages of training. To illustrate this, we train a linear model to predict the training ϵ from the hidden state of the recurrent model. Figure 3.5c shows a reasonably strong predictive accuracy even for a model trained with XP but no fingerprint, indicating that disambiguating information is indeed kept in the hidden states. However, the hidden states of a recurrent model trained with a fingerprint (Figure 3.5d) are even more informative.

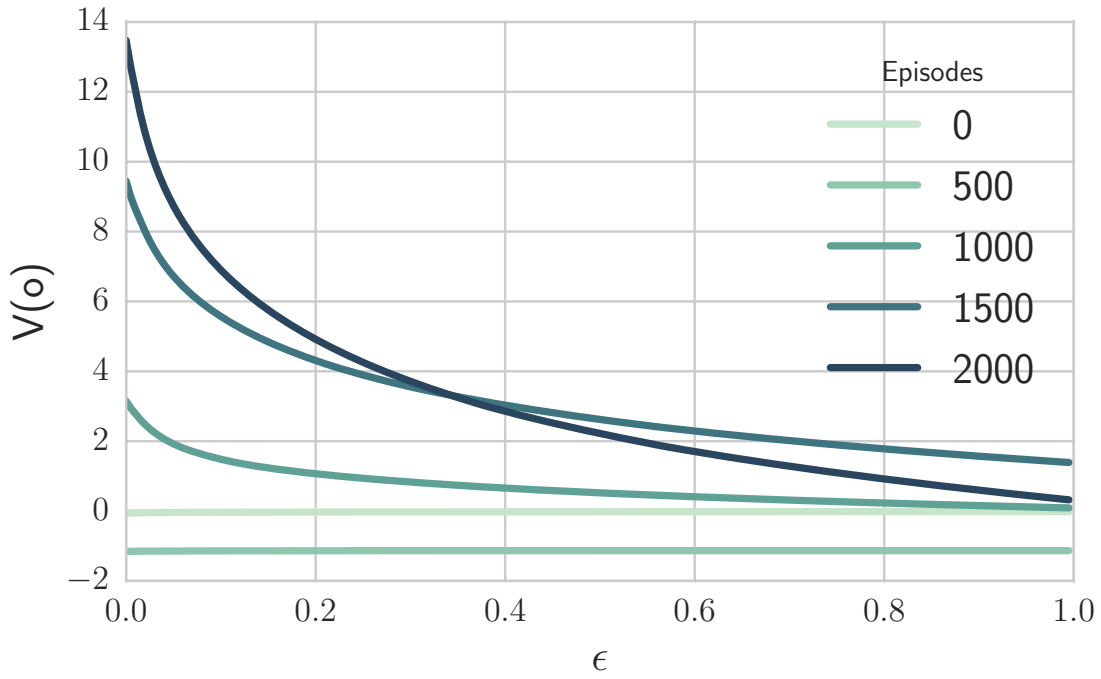


Figure 3.4: Estimated value of a single initial observation with different ϵ in its fingerprint input, at different stages of training. The network learns to smoothly vary its value estimates across different stages of training.

3.3 Counterfactual Multi-Agent Policy Gradients

Another crucial challenge is *multi-agent credit assignment* [29]: in cooperative settings, joint actions typically generate only global rewards, making it difficult for each agent to deduce its own contribution to the team's success. Sometimes it is possible to design individual reward functions for each agent. However, these rewards are not generally available in cooperative settings and often fail to encourage individual agents to sacrifice for the greater good. This often substantially impedes multi-agent learning in challenging tasks, even with relatively small numbers of agents.

In order to address these issues, we propose a new multi-agent RL method called *counterfactual multi-agent* (COMA) policy gradients. COMA takes an *actor-critic* [95] approach, in which the *actor*, i.e., the policy, is trained by following a gradient estimated by a *critic*. COMA is based on three main ideas.

First, COMA uses a centralised critic. The critic is only used during learning, while only the actor is needed during execution. Since learning is centralised, we can therefore use a centralised critic that conditions on the joint action and all available state information, while each agent's policy conditions only on its own action-observation history.

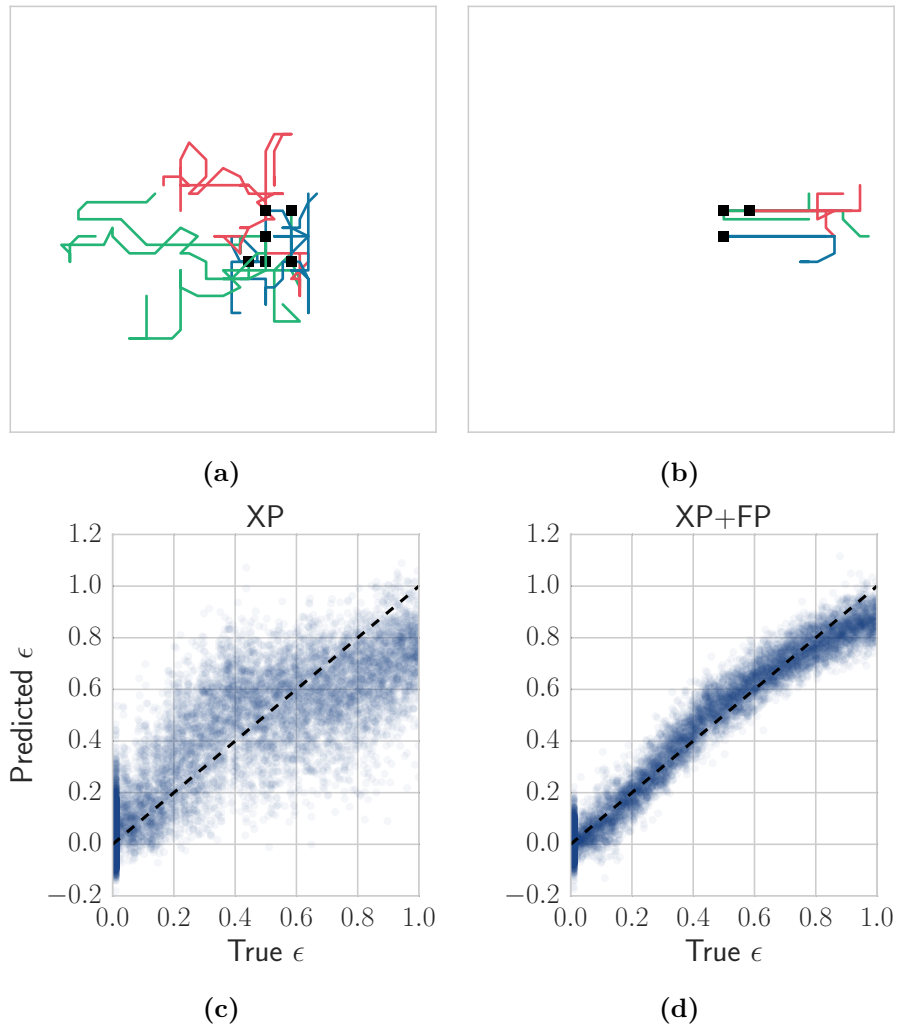


Figure 3.5: (upper) Sampled trajectories of agents, from the beginning (a) and end (b) of training. Each agent is one colour and the starting points are marked as black squares. (lower) Linear regression predictions of ϵ from the hidden state halfway through each episode in the replay buffer: (c) with only XP, the hidden state still contains disambiguating information drawn from the trajectories, (d) with XP+FP, the hidden state is more informative about the stage of training.

Second, COMA uses a *counterfactual baseline*. The idea is inspired by *difference rewards* [202, 187], in which each agent learns from a shaped reward that compares the global reward to the reward received when that agent's action is replaced with a *default action*. While difference rewards are a powerful way to perform multi-agent credit assignment, they require access to a simulator or estimated reward function, and in general it is unclear how to choose the default action. COMA addresses this by using the centralised critic to compute an agent-specific *advantage function* that compares the estimated return for the current joint action to a counterfactual baseline that marginalises out a single agent's action, while keeping

the other agents' actions fixed. This is similar to calculating an *aristocrat utility* [202], but avoids the problem of a recursive interdependence between the policy and utility function because the expected contribution of the counterfactual baseline to the policy gradient is zero. Hence, instead of relying on extra simulations, approximations, or assumptions regarding appropriate default actions, COMA computes a separate baseline for each agent that relies on the centralised critic to reason about counterfactuals in which only that agent's action changes.

Third, COMA uses a critic representation that allows the counterfactual baseline to be computed efficiently. In a single forward pass, it computes the Q-values for all the different actions of a given agent, conditioned on the actions of all the other agents. Because a single centralised critic is used for all agents, all Q-values for all agents can be computed in a single batched forward pass.

3.3.1 Independent Actor-Critic

The simplest way to apply policy gradients to multiple agents is to have each agent learn independently, with its own actor and critic, from its own action-observation history. This is essentially the idea behind *independent Q-learning* [179], which is perhaps the most popular multi-agent learning algorithm, but with actor-critic in place of Q-learning. Hence, we call this approach *independent actor-critic* (IAC).

In our implementation of IAC, we speed learning by sharing parameters among the agents, i.e., we learn only one actor and one critic, which are used by all agents. The agents can still behave differently because they receive different observations, including an agent-specific ID, and thus evolve different hidden states. Learning remains independent in the sense that each agent's critic estimates only a local value function, i.e., one that conditions on u^a, not $\mathbf{u}$. Though we are not aware of previous applications of this specific algorithm, we do not consider it a significant contribution but instead merely a baseline algorithm.

We consider two variants of IAC. In the first, each agent's critic estimates $V(\tau^a)$ and follows a gradient based on the TD error, as described in Section 2.3. In the second, each agent's critic estimates $Q(\tau^a, u^a)$ and follows a gradient based on the advantage: $A(\tau^a, u^a) = Q(\tau^a, u^a) - V(\tau^a)$, where $V(\tau^a) = \sum_{u^a} \pi(u^a|\tau^a)Q(\tau^a, u^a)$. Independent learning is straightforward, but the lack of information sharing at training time makes it difficult to learn coordinated strategies that depend on interactions between multiple agents, or for an individual agent to estimate the contribution of its actions to the team's reward.

3.3.2 Counterfactual Multi-Agent Policy Gradients

The difficulties discussed above arise because, beyond parameter sharing, IAC fails to exploit the fact that learning is centralised in our setting. In this section, we propose *counterfactual multi-agent* (COMA) policy gradients, which overcome this limitation. Three main ideas underlie COMA: 1) centralisation of the critic, 2) use of a counterfactual baseline, and 3) use of a critic representation that allows efficient evaluation of the baseline. The remainder of this section describes these ideas.

First, COMA uses a centralised critic. Note that in IAC, each actor $\pi(u^a|\tau^a)$ and each critic $Q(\tau^a, u^a)$ or $V(\tau^a)$ conditions only on the agent's own action-observation history τ^a. However, the critic is used only during learning and only the actor is needed during execution. Since learning is centralised, we can therefore use a centralised critic that conditions on the true global state s, if it is available, or the joint action-observation histories $\boldsymbol{\tau}$ otherwise. Each actor conditions on its own action-observation histories τ^a, with parameter sharing, as in IAC. Figure 3.6a illustrates this setup.

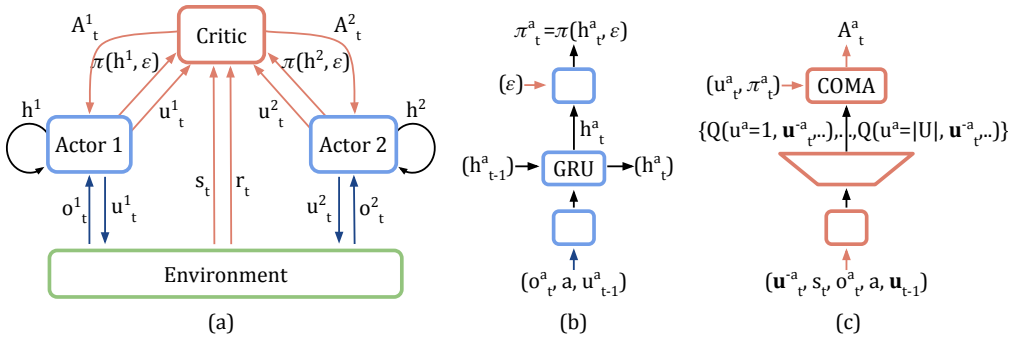


Figure 3.6: In (a), information flow between the decentralised actors, the environment and the centralised critic in COMA; red arrows and components are only required during centralised learning. In (b) and (c), architectures of the actor and critic.

A naive way to use this centralised critic would be for each actor to follow a gradient based on the TD error estimated from this critic:

$$g = \nabla_{\theta^\pi} \log \pi(u|\tau_t^a) \left(r + \gamma V(s_{t+1}) - V(s_t)\right).$$

However, such an approach fails to address a key credit assignment problem. Because the TD error considers only global rewards, the gradient computed for each actor does not explicitly reason about how that particular agent's actions contribute to that global reward. Since the other agents may be exploring, the gradient for that agent becomes very noisy, particularly when there are many agents.

Therefore, COMA uses a *counterfactual baseline*. The idea is inspired by *difference rewards* [202], in which each agent learns from a shaped reward $D^a = r(s, \mathbf{u}) - r(s, (\mathbf{u}^{-a}, c^a))$ that compares the global reward to the reward received when the action of agent a is replaced with a *default action* c^a. Any action by agent a that improves D^a also improves the true global reward $r(s, \mathbf{u})$, because $r(s, (\mathbf{u}^{-a}, c^a))$ does not depend on agent a's actions.

Difference rewards are a powerful way to perform multi-agent credit assignment. However, they typically require access to a simulator in order to estimate $r(s, (\mathbf{u}^{-a}, c^a))$. When a simulator is already being used for learning, difference rewards increase the number of simulations that must be conducted, since each agent's difference reward requires a separate counterfactual simulation. Proper and Tumer [138] and Colby et al. [38] propose estimating difference rewards using function approximation rather than a simulator. However, this still requires a user-specified default action c^a that can be difficult to choose in many applications. In an actor-critic architecture, this approach would also introduce an additional source of approximation error.

A key insight underlying COMA is that a centralised critic can be used to implement difference rewards in a way that avoids these problems. COMA learns a centralised critic, $Q(s, \mathbf{u})$ that estimates Q-values for the joint action $\mathbf{u}$ conditioned on the central state s. For each agent a we can then compute an advantage function that compares the Q-value for the current action u^a to a counterfactual baseline that marginalises out u^a, while keeping the other agents' actions $\mathbf{u}^{-a}$ fixed:

$$A^a(s, \mathbf{u}) = Q(s, \mathbf{u}) - \sum_{u'^a} \pi^a(u'^a|\tau^a) Q(s, (\mathbf{u}^{-a}, u'^a)).$$

Hence, $A^a(s, u^a)$ computes a separate baseline for each agent that uses the centralised critic to reason about counterfactuals in which only a's action changes, learned directly from agents' experiences instead of relying on extra simulations, a reward model, or a user-designed default action.

This advantage has the same form as the *aristocrat utility* [202]. However, optimising for an aristocrat utility using value-based methods creates a self-consistency problem because the policy and utility function depend recursively on each other. As a result, prior work focused on difference evaluations using default states and actions. COMA is different because the counterfactual baseline's expected contribution to the gradient, as with other policy gradient baselines, is zero. Thus, while the baseline does depend on the policy, its expectation does not. Consequently, COMA can use this form of the advantage without creating a self-consistency problem.

While COMA's advantage function replaces potential extra simulations with evaluations of the critic, those evaluations may themselves be expensive if the critic is a deep neural network. Furthermore, in a typical representation, the number of output nodes of such a network would equal $|U|^n$, the size of the joint action space, making it impractical to train. To address both these issues, COMA uses a critic representation that allows for efficient evaluation of the baseline. In particular, the actions of the other agents, $\mathbf{u}_t^{-a}$, are part of the input to the network, which outputs a Q-value for each of agent a's actions, as shown in Figure 3.6c. Consequently, the counterfactual advantage can be calculated efficiently by a single forward pass of the actor and critic, for each agent. Furthermore, the number of outputs is only $|U|$ instead of $(|U|^n)$. While the network has a large input space that scales linearly in the number of agents and actions, deep neural networks can generalise well across such spaces.

In this paper, we focus on settings with discrete actions. However, COMA can be easily extended to continuous actions spaces by estimating the expectation in (3.3.2) with Monte Carlo samples or using functional forms that render it analytical, e.g., Gaussian policies and critic.

The following lemma establishes the convergence of COMA to a locally optimal policy. The proof follows directly from the convergence of single-agent actor-critic algorithms [171, 95], and is subject to the same assumptions.

Lemma 1. *For an actor-critic algorithm with a compatible TD(1) critic following a COMA policy gradient*

$$g_k = \mathbb{E}_{\boldsymbol{\pi}}\left[\sum_a \nabla_{\theta_k} \log \pi^a(u^a|\tau^a) A^a(s, \mathbf{u})\right]$$

at each iteration k,

$$\liminf_k ||\nabla J|| = 0 \quad w.p.\ 1.$$

Proof. The COMA gradient is given by

$$g = \mathbb{E}_{\boldsymbol{\pi}}\left[\sum_a \nabla_\theta \log \pi^a(u^a|\tau^a) A^a(s, \mathbf{u})\right],$$
$$A^a(s, \mathbf{u}) = Q(s, \mathbf{u}) - b(s, \mathbf{u}^{-a}),$$

where θ are the parameters of all actor policies, e.g. $\theta = \{\theta^1, \ldots, \theta^{|A|}\}$, and $b(s, \mathbf{u}^{-a})$ is the counterfactual baseline defined in equation 3.3.2.

First consider the expected contribution of the baseline $b(s, \mathbf{u}^{-a})$:

$$g_b = -\mathbb{E}_{\boldsymbol{\pi}}\left[\sum_a \nabla_\theta \log \pi^a(u^a|\tau^a) b(s, \mathbf{u}^{-a})\right],$$

where the expectation $\mathbb{E}_{\boldsymbol{\pi}}$ is with respect to the state-action distribution induced by the joint policy $\boldsymbol{\pi}$. Now let $d^{\boldsymbol{\pi}}(s)$ be the discounted ergodic state distribution as defined by Sutton et al. [171]:

$$\begin{aligned} g_b &= -\sum_s d^{\boldsymbol{\pi}}(s) \sum_a \sum_{\mathbf{u}^{-a}} \boldsymbol{\pi}(\mathbf{u}^{-a}|\boldsymbol{\tau}-a) \cdot \\ &\quad \sum_{u^a} \pi^a(u^a|\tau^a) \nabla_\theta \log \pi^a(u^a|\tau^a) b(s, \mathbf{u}^{-a}) \\ &= -\sum_s d^{\boldsymbol{\pi}}(s) \sum_a \sum_{\mathbf{u}^{-a}} \boldsymbol{\pi}(\mathbf{u}^{-a}|\boldsymbol{\tau}-a) \cdot \\ &\quad \sum_{u^a} \nabla_\theta \pi^a(u^a|\tau^a) b(s, \mathbf{u}^{-a}) \\ &= -\sum_s d^{\boldsymbol{\pi}}(s) \sum_a \sum_{\mathbf{u}^{-a}} \boldsymbol{\pi}(\mathbf{u}^{-a}|\boldsymbol{\tau}-a) b(s, \mathbf{u}^{-a}) \nabla_\theta 1 \\ &= 0. \end{aligned}$$

Clearly, the per-agent baseline, although it reduces variance, does not change the expected gradient, and therefore does not affect the convergence of COMA.

The remainder of the expected policy gradient is given by:

$$\begin{aligned} g &= \mathbb{E}_{\boldsymbol{\pi}} \left[\sum_a \nabla_\theta \log \pi^a(u^a|\tau^a) Q(s, \mathbf{u}) \right] \\ &= \mathbb{E}_{\boldsymbol{\pi}} \left[\nabla_\theta \log \prod_a \pi^a(u^a|\tau^a) Q(s, \mathbf{u}) \right]. \end{aligned}$$

Writing the joint policy as a product of the independent actors:

$$\boldsymbol{\pi}(\mathbf{u}|s) = \prod_a \pi^a(u^a|\tau^a),$$

yields the standard single-agent actor-critic policy gradient:

$$g = \mathbb{E}_{\boldsymbol{\pi}} \left[\nabla_\theta \log \boldsymbol{\pi}(\mathbf{u}|s) Q(s, \mathbf{u}) \right].$$

Konda and Tsitsiklis [95] prove that an actor-critic following this gradient converges to a local maximum of the expected return $J^{\boldsymbol{\pi}}$, given that:

1. the policy $\boldsymbol{\pi}$ is differentiable,
2. the update timescales for Q and $\boldsymbol{\pi}$ are sufficiently slow, and that $\boldsymbol{\pi}$ is updated sufficiently slower than Q, and
3. Q uses a representation compatible with $\boldsymbol{\pi}$,

amongst several further assumptions. The parameterisation of the policy (i.e., the single-agent joint-action learner is decomposed into independent actors) is immaterial to convergence, as long as it remains differentiable. Note however that COMA's centralised critic is essential for this proof to hold. □

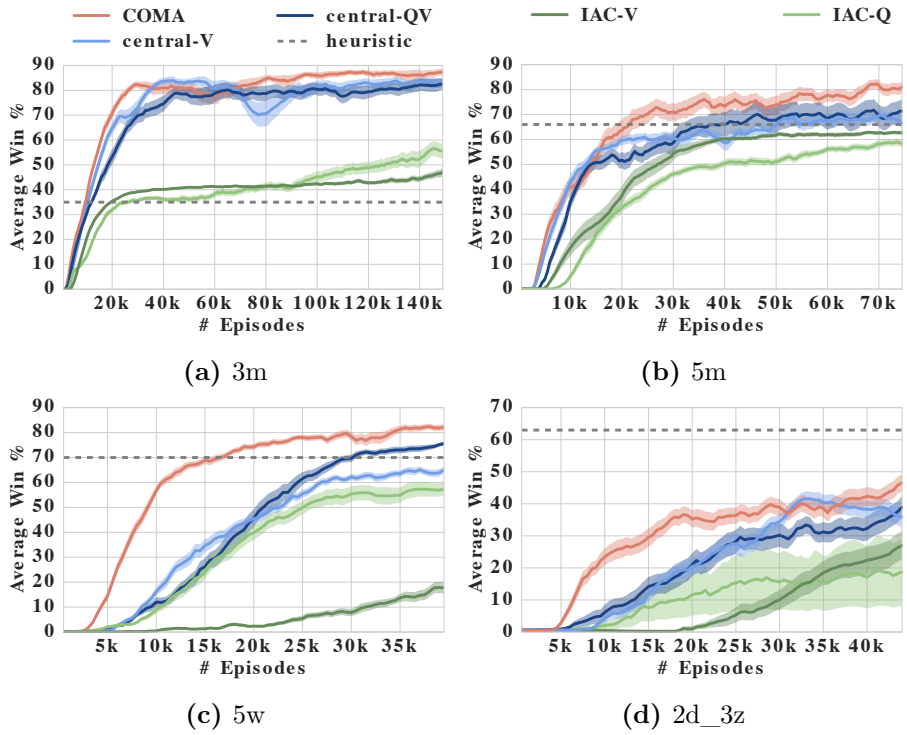


Figure 3.7: Win rates for COMA and competing algorithms on four different scenarios. COMA outperforms all baseline methods. Centralised critics also clearly outperform their decentralised counterparts. The legend at the top applies across all plots.

3.3.3 Experiments

Similarly to Section 3.2.3, the actor and critic receive different input features, corresponding to local observations and global state, respectively. Both include features for allies and enemies. *Units* can be either allies or enemies, while *agents* are the decentralised controllers that command ally units.

The local observations for every agent are drawn only from a circular subset of the map centred on the unit it controls and include for each unit within this field of view: `distance`, `relative x`, `relative y`, `unit type` and `shield`.[1] All features are normalised by their maximum values. We do not include any information about the units' current target.

The global state representation consists of similar features, but for all units on the map regardless of fields of view. Absolute distance is not included, and x-y locations are given relative to the centre of the map rather than to a particular agent. The global state also includes `health points` and `cooldown` for all agents. The representation fed to the centralised Q-function critic is the concatenation of

[1] After firing, a unit's `cooldown` is reset, and it must drop before firing again. Shields absorb damage until they break, after which units start losing health. Dragoons and zealots have shields, but marines do not.

the global state representation with the local observation of the agent whose actions are being evaluated. Our centralised critic that estimates $V(s)$, and is therefore agent-agnostic, receives the global state concatenated with all agents' observations. The observations contain no new information but include the egocentric distances relative to that agent.

The actor consists of 128-bit *gated recurrent units* (GRUs) [33] that use fully connected layers both to process the input and to produce the output values from the hidden state, h_t^a. The IAC critics use extra output heads appended to the last layer of the actor network. Action probabilities are produced from the final layer, $\mathbf{z}$, via a bounded softmax distribution that lower-bounds the probability of any given action by $\epsilon/|U|$: $P(u) = (1-\epsilon)\text{softmax}(\mathbf{z})_u + \epsilon/|U|)$. We anneal ϵ linearly from 0.5 to 0.02 across 750 training episodes. The centralised critic is a feedforward network with multiple ReLU layers combined with fully connected layers. Hyperparameters were coarsely tuned on the 5m scenario and then used for all other maps. We found that the most sensitive parameter was TD(λ), but settled on $\lambda = 0.8$, which worked best for both COMA and our baselines.

We experimented with critic architectures that are factored at the agent level and further exploit internal parameter sharing. However, we found that the bottleneck for scalability was not the centralisation of the critic, but rather the difficulty of multi-agent exploration. Hence, we defer further investigation of factored COMA critics to future work.

We perform ablation experiments to validate three key elements of COMA. First, we test the importance of centralising the critic by comparing against two IAC variants, IAC-Q and IAC-V. These critics take the same decentralised input as the actor, and share parameters with the actor network up to the final layer. IAC-Q then outputs $|U|$ Q-values, one for each action, while IAC-V outputs a single state-value. Note that we still share parameters between agents, using the egocentric observations and ID's as part of the input to allow different behaviours to emerge. The cooperative reward function is still shared by all agents.

Second, we test the significance of learning Q instead of V. The method *central-V* still uses a central state for the critic, but learns $V(s)$, and uses the TD error to estimate the advantage for policy gradient updates.

Third, we test the utility of our counterfactual baseline. The method *central-QV* learns both Q and V simultaneously and estimates the advantage as $Q-V$, replacing COMA's counterfactual baseline with V. All methods use the same architecture and training scheme for the actors, and all critics are trained with TD(λ).

	Local Field of View (FoV)							Full FoV, Central Control		
map	heur.	IAC-V	IAC-Q	cnt-V	cnt-QV	COMA mean	COMA best	heur.	DQN	GMEZO
3m	35	47 (3)	56 (6)	83 (3)	83 (5)	**87** (3)	98	74	-	-
5m	66	63 (2)	58 (3)	67 (5)	71 (9)	**81** (5)	95	98	99	100
5w	70	18 (5)	57 (5)	65 (3)	76 (1)	**82** (3)	98	82	70	74
2d_3z	**63**	27 (9)	19 (21)	36 (6)	39 (5)	47 (5)	65	68	61	90

Table 3.1: Mean win percentage averaged across final 1000 evaluation episodes for the different maps, for all methods and the hand-coded heuristic in the decentralised setting with a limited field of view. The highest mean performances are in bold, while the values in parentheses denote the 95% confidence interval, for example $87(3) = 87 \pm 3$. Also shown, maximum win percentages for COMA (decentralised), in comparison to the heuristic and published results (evaluated in the centralised setting).

Figure 3.7 shows average win rates as a function of episode for each method and each StarCraft scenario. For each method, we conducted 35 independent trials and froze learning every 100 training episodes to evaluate the learned policies across 200 episodes per method, plotting the average across episodes and trials. Also shown is one standard deviation in performance.

The results show that COMA is superior to the IAC baselines in all scenarios. Interestingly, the IAC methods also eventually learn reasonable policies in 5m, although they need substantially more episodes to do so. This may seem counterintuitive since in the IAC methods, the actor and critic networks share parameters in their early layers, which could be expected to speed learning. However, these results suggest that the improved accuracy of policy evaluation made possible by conditioning on the global state outweighs the overhead of training a separate network.

Furthermore, COMA strictly dominates central-QV, both in training speed and in final performance across all settings. This is a strong indicator that our counterfactual baseline is crucial when using a central Q-critic to train decentralised policies.

Learning a state-value function has the obvious advantage of not conditioning on the joint action. Still, we find that COMA outperforms the central-V baseline in final performance. Furthermore, COMA typically achieves good policies faster, which is expected as COMA provides a shaped training signal. Training is also more stable than central-V, which is a consequence of the COMA gradient tending to zero as the policy becomes greedy. Overall, COMA is the best performing and most consistent method.

Usunier et al. [188] report the performance of their best agents trained with their state-of-the-art centralised controller labelled GMEZO (greedy-MDP with episodic

zero-order optimisation), and for a centralised DQN controller, both given a full field of view and access to attack-move macro-actions. These results are compared in Table 3.1 against the best agents trained with COMA for each map. Clearly, in most settings these agents achieve performance comparable to the best published win rates despite being restricted to decentralised policies and local fields of view.

3.4 The StarCraft Multi-Agent Challenge

In the previous sections of this chapter, we have discussed how to effectively produce modern versions of multi-agent RL algorithms to tackle the ever present issues in these frameworks. However, there are still many challenges that remain to be addressed, and the literature on multi-agent RL is as variegated as it is disorganised. Over time, researchers have proposed one-off environments with overly simple dynamics, or directly tuned to certain algorithms.

Some testbeds have emerged for other multi-agent regimes, such as Poker [71], Pong [178], Keepaway Soccer [164], or simple gridworld-like environments [109, 106, 204, 208]. Nonetheless, we identify a clear gap in challenging and standardised testbeds for the important set of domains described above. Since it is fairly clear that standard environments such as the Arcade Learning Environment [13] and MuJoCo [137], have enabled great progress in single-agent RL, we aim to fill the gap so as to facilitate good research progress in the multi-agent setting. We do so by the StarCraft Multi-Agent Challenge (SMAC).

SMAC is built on the popular real-time strategy game StarCraft II[2] and makes use of the SC2LE environment [192]. Following what we learnt by in Section 3.1, instead of tackling the full game of StarCraft with centralised control, we focus on employing StarCraft II as a generator for fixed, decentralised micromanagement challenges (Figure 3.8). Our work presents the first standardised testbed for decentralised control in this space.

By introducing decentralisation and local observability, our agents are excessively restricted compared to normal full gameplay. SMAC is therefore not intended as an environment to train agents for use in full gameplay. Instead, we use the StarCraft II game engine to build rich and interesting multi-agent problems. In these challenges, each of our units is controlled by an independent, learning agent that has to act based only on local observations, while the opponent's units are controlled by the hand-coded built-in StarCraft II AI. We offer a diverse set of scenarios that challenge

[2]StarCraft II is the sequel to the game StarCraft and its expansion set Brood War. StarCraft and StarCraft II are trademarks of Blizzard Entertainment™.

algorithms to handle high-dimensional inputs and partial observability, and to learn coordinated behaviour even when restricted to fully decentralised execution.

In the single agent setting, AlphaStar [45] has recently shown an impressive level of play on a StarCraft II matchup using a centralised controller. In contrast, SMAC is not intended as an environment to train agents for use in full StarCraft II gameplay. Instead, by introducing strict decentralisation and local partial observability, we use the StarCraft II game engine to build a new set of rich cooperative multi-agent problems that bring well-known unique multi-agent challenges such as non-stationary dynamics and credit assignment already presented in Sections 3.2 and 3.3), while also providing a benchmark for specific Deep Multi-Agent RL issues such as value function representation of joint actions [140].

Very importantly, SMAC comes with a set of guidelines for best practices in evaluations using our benchmark, including the reporting of standardised performance metrics, sample efficiency, and computational requirements (see Appendix A.1.3).

3.4.1 Methods

In *SMAC*, we focus on tasks that require a team of agents to solve a given task, such as defeating all enemies, in a *cooperative manner*. Furthermore, we investigate tasks that are under partial observability and/or communication constraints and therefore require control policies to be decentralised. This means that each agent needs to be able to execute its own policy given only its own action-observation history as input. Formally, such *fully cooperative multi-agent tasks* can be described as *Dec-POMDPs*[125]. We broadly summarise what we have discussed so far.

In their most general form, cooperative multi-agent reinforcement learning problems require the learning a centralised state-action value function Q_{tot} that conditions on the global state and the joint action. As both joint action and global state space grow exponentially with the number of agents, the learning of Q_{tot} is often intractable for many-agent tasks. Also, it is usually not clear how to execute policies derived from Q_{tot} in a decentralised fashion. A naive way of achieving decentralisation is to forgo centralised learning altogether and instead learn each agent's policy independently, treating all other agents as part of the environment. *Independent Q-learning (IQL)* [180] uses Q-learning[194] to learn individual state-action value functions for each agent. Similarly, *independent actor-critic (IAC)* [54] uses a single-agent actor-critic approach in order to learn a critic and a policy for each agent.

If input spaces are high-dimensional, both IQL and IAC can be readily adapted to use deep function approximators. In its deep form, IQL's per-agent loss function is akin to *deep Q-learning (DQN)*[116], i.e.

$$\mathcal{L}_{IQL}(\theta) = \sum_{i=1,a=1}^{i=b,a=n} \left[\left(y_i^{DQN} - Q(\tau^a, u^a; \theta) \right)^2 \right]$$

where a runs over the number of agents n, b indexes elements in a batch of size b, τ^a is agent a's action-observation history, u^a is agent a's action taken, $y^{DQN} = r + \gamma \max_{u'} Q(s', u'; \theta^-)$, θ^- are the parameters of a *target network* that are periodically copied from the critic's network parameters θ and kept constant for a number of iterations. Note that batches are drawn from a experience replay buffer, which may be stabilised [56].

In IAC, the per-agent loss follows a standard *actor-critic* approach:

$$\mathcal{L}_{IAC}(\theta_{\pi,A}) = \sum_{i=1,a=1}^{i=b,a=n} \log \pi_{\theta_\pi}(u_{a,i}|\tau_{a,i}) A_{\theta_A}(\tau_{a,i})$$

where, in its simplest form, A is given by the per-agent advantage $(r+\gamma V_{t+1} - V_t)$, given a per-agent critic estimate $V^a(\tau_t^a)$. Similarly to DQN, both π and A may be approximated using deep neural networks with weights $\theta_{\pi,A}$, in which case V makes use of a target network. As actor-critic methods are inherently on-policy, IAC does not use experience replay.

If the learning process proceeds in simulation (e.g. StarCraft II) or in a laboratory, then it can be conducted in a centralised fashion. This gives rise to the paradigm of *centralised training with decentralised execution*, which has been well-studied in the planning community [126] [96]. We have recently developed deep multi-agent reinforcement learning algorithms that can make use of extra state information available in centralised training in order to speed up the learning of decentralisable policies [54][140].

Additionally to the presented methods, we additionally test *QMIX*[140], which uses deep Q-learning to find a joint state-action value Q^{tot} function that factorises into per-agent *utility functions* Q^a. Generalising *VDN*[168], the factorization $Q^{tot} = f(Q^1, \ldots, Q^n, s)$ is chosen to be monotonic in Q^a, which ensures that greedy action selection performed on the utility functions is consistent with Q^{tot}:

$$\left[\arg\max_{\mathbf{u}_{\text{joint}}} Q^{tot}(\mathbf{u}_{\text{joint}}) \right]_a = \arg\max_{u_a} Q^a$$

, where s is the global state only available during training.

Hence Q^{tot} can make use of s during training, but execution only requires the decentralised utility function Q^a. Furthermore, greedy evaluation scales only linearly with the number of agents and actions.

Like *Central-V*[54], *COMA*[54] uses a joint critic that conditions on the central state s, while keeping agent policies decentralisable. Also, *COMA* employs a variance-reducing *counterfactual baseline*, that helps mitigate the environment non-stationarity introduced due to independent agent learning. This improves over *IAC* by using an advantage function of the form instead of the critic:

$$A^a(s, \mathbf{u}) = Q(s, \mathbf{u}) - \sum_{u'^a} \pi^a(u'^a|\tau^a) Q\left(s, (\mathbf{u}^{-a}, u'^a)\right)$$

where $\mathbf{u}^{-a}$ signifies all actions other than agent a's and as before, s is the global state only available during training.

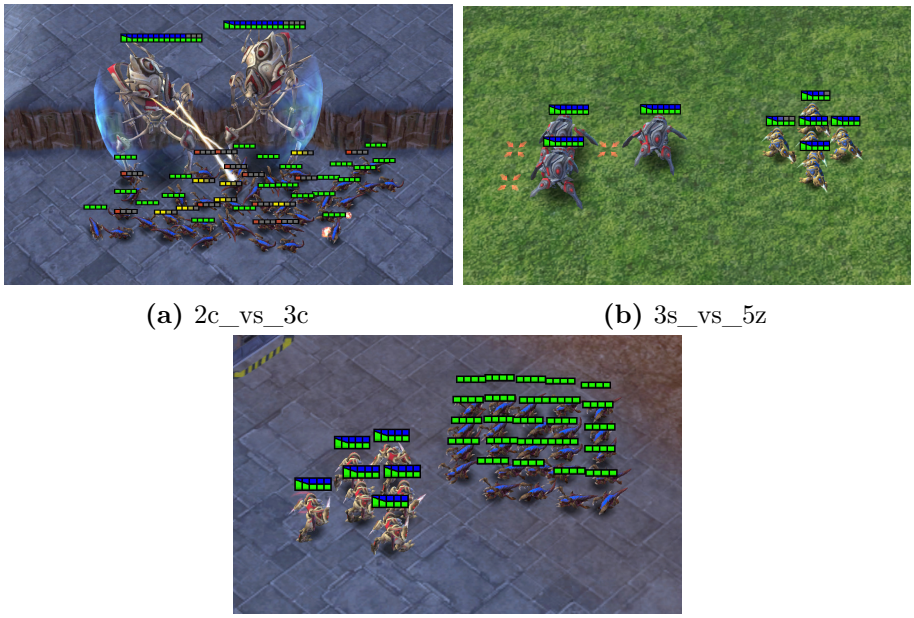
Both *QMIX* and *COMA* yield state-of-the-art results on StarCraft II for Q-learning and actor-critic based approaches, respectively.

3.4.2 SMAC

In order to build a rich multi-agent testbed, we focus solely on StarCraft micromanagement. Unit micromanagement is a vital aspect of StarCraft gameplay with a high skill ceiling, and is practised in isolation by amateur and professional players. For SMAC, we leverage the natural multi-agent structure of micromanagement by proposing a modified version of the problem designed specifically for decentralised control. In particular, we require that each unit be controlled by an independent agent that conditions only on local observations restricted to a limited field of view centred on that unit (see Figure 3.9). Groups of these agents must be trained to solve challenging combat scenarios, battling an opposing army under the centralised control of the game's built-in scripted AI.

Proper micro of units during battles maximises the damage dealt to enemy units while minimising damage received, and requires a range of skills. For example, one important technique is *focus fire*, i.e., ordering units to jointly attack and kill enemy units one after another. When focusing fire, it is important to avoid *overkill*: inflicting more damage to units than is necessary to kill them.

Other common micro techniques include: assembling units into formations based on their armour types, making enemy units give chase while maintaining enough distance so that little or no damage is incurred (*kiting*), coordinating the positioning of units to attack from different directions or taking advantage of



(a) 2c_vs_3c **(b)** 3s_vs_5z

(c) corridor

Figure 3.8: Screenshots of three SMAC scenarios.

the terrain to defeat the enemy. Figure 3.8b illustrates how three Stalker units kite five enemy Zealots units.

SMAC thus provides a convenient environment for evaluating the effectiveness of MARL algorithms. The simulated StarCraft II environment and carefully designed scenarios require learning rich cooperative behaviours under partial observability, which is a challenging task. The simulated environment also provides an additional state information during training, such as information on all the units on the entire map. This is crucial for facilitating algorithms to take full advantage of the centralised training regime and assessing all aspects of MARL methods. SMAC is a qualitatively challenging environment that provides together elements of partial observability, challenging dynamics, and high-dimensional observation spaces.

Scenarios

SMAC consists of a set of StarCraft II micro scenarios which aim to evaluate how well independent agents are able to learn coordination to solve complex tasks. These scenarios are carefully designed to necessitate the learning of one or more micromanagement techniques to defeat the enemy. Each scenario is a confrontation between two armies of units. The initial position, number, and type of units in each army varies from scenario to scenario, as does the presence or absence of elevated or impassable terrain. Figures 3.8 include screenshots of several SMAC micro scenarios.

Table 3.2: SMAC challenges.

Name	Ally Units	Enemy Units
2s3z	2 Stalkers & 3 Zealots	2 Stalkers & 3 Zealots
3s5z	3 Stalkers & 5 Zealots	3 Stalkers & 5 Zealots
1c3s5z	1 Colossus, 3 Stalkers & 5 Zealots	1 Colossus, 3 Stalkers & 5 Zealots
5m_vs_6m	5 Marines	6 Marines
10m_vs_11m	10 Marines	11 Marines
27m_vs_30m	27 Marines	30 Marines
3s5z_vs_3s6z	3 Stalkers & 5 Zealots	3 Stalkers & 6 Zealots
MMM2	1 Medivac, 2 Marauders & 7 Marines	1 Medivac, 3 Marauders & 8 Marines
2s_vs_1sc	2 Stalkers	1 Spine Crawler
3s_vs_5z	3 Stalkers	5 Zealots
6h_vs_8z	6 Hydralisks	8 Zealots
bane_vs_bane	20 Zerglings & 4 Banelings	20 Zerglings & 4 Banelings
2c_vs_64zg	2 Colossi	64 Zerglings
corridor	6 Zealots	24 Zerglings

The first army is controlled by the learned allied agents. The second army consists of enemy units controlled by the built-in game AI, which uses carefully handcrafted non-learned heuristics. At the beginning of each episode, the game AI instructs its units to attack the allied agents using its scripted strategies. An episode ends when all units of either army have died or when a pre-specified time limit is reached (in which case the game is counted as a defeat for the allied agents). The goal is to maximise the win rate, i.e., the ratio of games won to games played.

The complete list of challenges is presented in Table 3.2[3]. More specifics on the SMAC scenarios and environment settings can be found in Appendices A.1.3.2 and A.1.3.3 respectively.

State and Observations

At each timestep, agents receive local observations drawn within their field of view. This encompasses information about the map within a circular area around each unit and with a radius equal to the *sight range* (Figure 3.9). The sight range makes the environment partially observable from the standpoint of each agent. Agents can only observe other agents if they are both alive and located within the sight range. Hence, there is no way for agents to distinguish between teammates that are far away from those that are dead.

The feature vector observed by each agent contains the following attributes for both allied and enemy units within the sight range: `distance`, `relative x`,

[3]The list of SMAC scenarios has been updated from the earlier version. All scenarios, however, are still available in the repository.

`relative y`, `health`, `shield`, and `unit_type`. Shields serve as an additional source of protection that needs to be removed before any damage can be done to the health of units. All Protos units have shields, which can regenerate if no new damage is dealt. In addition, agents have access to the last actions of allied units that are in the field of view. Lastly, agents can observe the terrain features surrounding them, in particular, the values of eight points at a fixed radius indicating height and walkability.

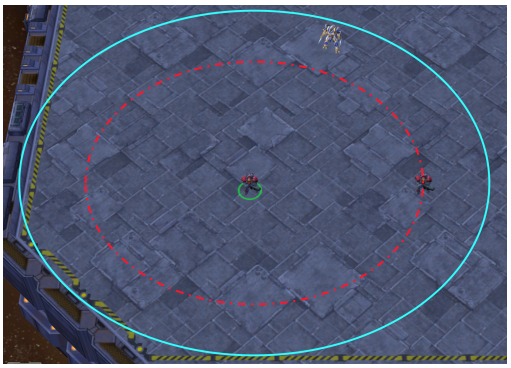


Figure 3.9: The cyan and red circles respectively border the sight and shooting range of the agent.

The global state, which is only available to agents during centralised training, contains information about all units on the map. Specifically, the state vector includes the coordinates of all agents relative to the centre of the map, together with unit features present in the observations. Additionally, the state stores the `energy` of Medivacs and `cooldown` of the rest of the allied units, which represents the minimum delay between attacks. Finally, the last actions of all agents are attached to the central state. All features, both in the state as well as in the observations of individual agents, are normalised by their maximum values. The sight range is set to nine for all agents.

Action Space

The discrete set of actions that agents are allowed to take consists of `move[direction]`[4], `attack[enemy_id]`, `stop` and `no-op`.[5] As healer units, Medivacs use `heal[agent_id]` actions instead of `attack[enemy_id]`. The maximum number of actions an agent can take ranges between 7 and 70, depending on the scenario. To ensure decentralisation of the task, agents can use the `attack[enemy_id]` action only on enemies in their *shooting range* (Figure 3.9). This additionally constrains the ability of the units to use the built-in *attack-move* macro-actions on the enemies that are far away. We set the shooting range equal to 6 for all agents. Having a larger sight range than a shooting range forces agents to make use of the move commands before starting to fire.

[4]Four directions: north, south, east, or west.
[5]Dead agents can only take `no-op` action while live agents cannot.

Rewards

The overall goal is to maximise the win rate for each battle scenario. The default setting is to use the *shaped reward*, which produces a reward based on the hit-point damage dealt and enemy units killed, together with a special bonus for winning the battle. The exact values and scales for each of these events can be configured using a range of flags. To produce fair comparisions we encourage using this default reward function for all scenarios. We also provide another *sparse reward* option, in which the reward is +1 for winning and -1 for losing an episode.

3.4.3 Experiments

In this section, we present results for scenarios included as part of SMAC. The purpose of these results is to demonstrate the performance of the current state-of-the-art methods in our chosen MARL paradigm.

The evaluation procedure is similar to the one in [140]. The training is paused after every 10000 timesteps during which 32 test episodes are run with agents performing action selection greedily in a decentralised fashion. The percentage of episodes where the agents defeat all enemy units within the permitted time limit is referred to as the *test win rate*.

The architectures and training details are presented in Appendix A.1.1. A table of the results is included in Appendix A.1.2.

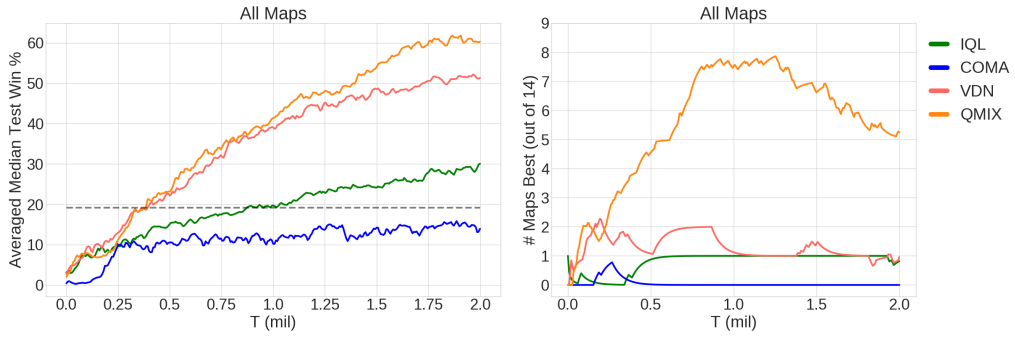


Figure 3.10: Left: The median test win %, averaged across all 14 scenarios. Heuristic's performance is shown as a dotted line. Right: The number of scenarios in which the algorithm's median test win % is the highest by at least 1/32 (smoothed).

Figure 3.10 plots the median test win percentage averaged across all scenarios to compare the algorithms across the entire SMAC suite. We also plot the performance of a simple heuristic AI that selects the closest enemy unit (ignoring partial observability) and attacks it with the entire team until it is dead, upon which the next closest enemy unit is selected. This is a basic form of *focus-firing*, which

is a crucial tactic for achieving good performance in micromanagement scenarios. The relatively poor performance of the heuristic AI shows that the suite of SMAC scenarios requires more complex behaviour than naively focus-firing the closest enemy, making it an interesting and challenging benchmark.

Overall QMIX achieves the highest test win percentage and is the best performer on up to eight scenarios during training. Additionally, IQL, VDN, and QMIX all significantly outperform COMA, demonstrating the sample efficiency of off-policy value-based methods over on-policy policy gradient methods.

Based on the results, we broadly group the scenarios into 3 categories: *Easy*, *Hard*, and *Super-Hard* based on the performance of the algorithms.

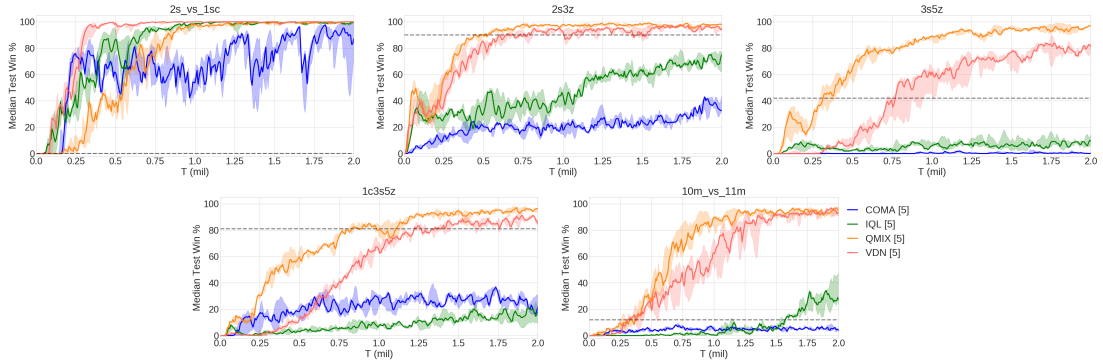


Figure 3.11: Easy scenarios. The heuristic AI's performance shown as a dotted black line.

Figure 3.11 shows that IQL and COMA struggle even on the *Easy* scenarios, performing poorly on four of the five scenarios in this category. This shows the advantage of learning a centralised but factored centralised Q_{tot}. Even though QMIX exceeds 95% test win rate on all of five *Easy* scenarios, they serve an important role in the benchmark as sanity checks when implementing and testing new algorithms.

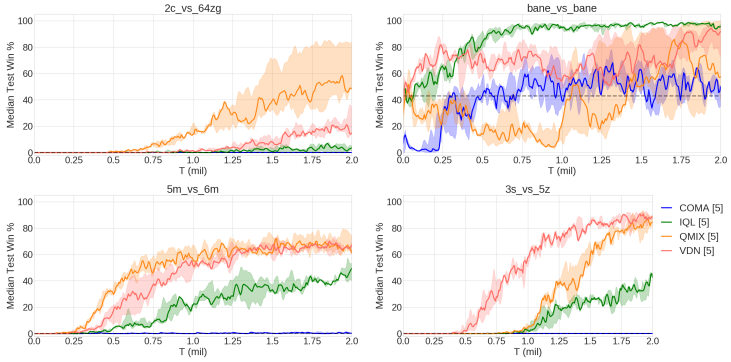


Figure 3.12: Hard scenarios. The heuristic AI's performance shown as a dotted black line.

The *Hard* scenarios in Figure 3.12 each present their own unique problems. *2c_vs_64zg* only contains 2 allied agents, but 64 enemy units (the largest in the SMAC benchmark) making the action space of the agents much larger than the other scenarios. *bane_vs_bane* contains a large number of allied and enemy units, but the results show that IQL easily finds a winning strategy whereas all other methods struggle and exhibit large variance. *5m_vs_6m* is an asymmetric scenario that requires precise control to win consistently, and in which the best performers (QMIX and VDN) have plateaued in performance. Finally, *3s_vs_5z* requires the three allied stalkers to *kite* the enemy zealots for the majority of the episode (at least 100 timesteps), which leads to a delayed reward problem.

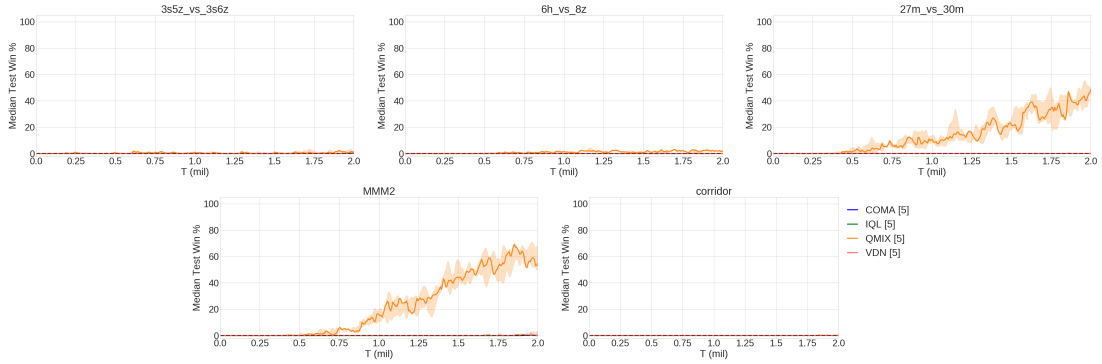


Figure 3.13: Super Hard scenarios. The heuristic AI's performance shown as a dotted black line.

The scenarios shown in Figure 3.13 are categorised as *Super Hard* because of the poor performance of all algorithms, with only QMIX making meaningful progress on two of the five. We hypothesise that exploration is a bottleneck in many of these scenarios, providing a nice test-bed for future research in this domain.

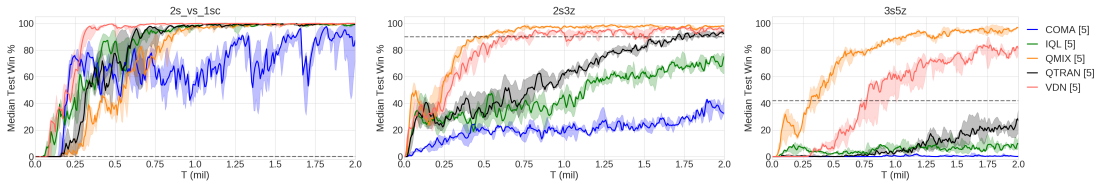


Figure 3.14: 3 scenarios including QTRAN.

We also test QTRAN [162] on three of the easiest scenarios, as shown in Figure 3.14. QTRAN fails to achieve good performance on *3s5z* and takes far longer to reach the performance of VDN and QMIX on *2s3z*. In preliminary experiments, we found the QTRAN-Base algorithm slightly more performant and more stable than QTRAN-Alt. For more details on the hyperparameters and architectures considered, please see the Appendix.

Additionally, we compare the necessity of using an agent's action-observation history by comparing the performance of an agent network with and without an RNN in Figure 3.15. On the easy scenario of *3s5z* we can see that an RNN is not required to use action-observation information from previous timesteps, but on the harder scenarios of *3s_vs_5z* it is crucial to being able to learn how to kite effectively.

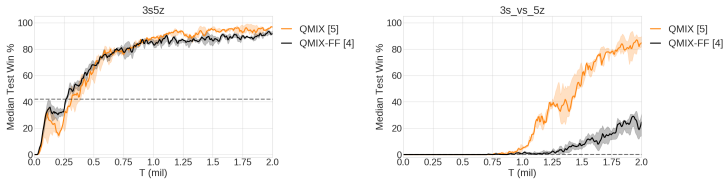


Figure 3.15: Comparing agent networks with and without RNNs (QMIX-FF) on 2 scenarios.

3.5 Conclusion

We presented a variety of methods for solving emerging problems in the application of function approximators to Multi-Agent Reinforcement Learning. Firstly we proposed two methods for stabilising experience replay in deep multi-agent reinforcement learning: (a) using a multi-agent variant of importance sampling to naturally decay obsolete data and (b) conditioning each agent's value function on a fingerprint that disambiguates the age of the data sampled from the replay memory. Results confirmed that these methods enable the successful combination of experience replay with multiple agents. Secondly, we developed COMA policy gradients, a method that uses a centralised critic in order to estimate a counterfactual advantage for decentralised policies in mutli-agent RL. COMA addresses the challenges of multi-agent credit assignment by using a counterfactual baseline that marginalises out a single agent's action, while keeping the other agents' actions fixed.

These methods were presented both in a decentralised *StarCraft unit micromanagement* environment built on TorchCraft[172], as well as a novel benchmark called the StarCraft Multi-Agent Challenge, which took lessons learnt from developing the earlier environment and produced a reusable benchmark for Multi-Agent Reinforcement Learning and its community.

4
Value Propagation Networks

Planning is a key component for artificial agents in a variety of domains. However, a limit of classical planning algorithms is that one needs to know how to search for an optimal – or at least reasonable – solution for each instantiation of every possible type of plan. As the environment dynamics and states complexity increase, this makes writing planners difficult, cumbersome, or simply entirely impractical. This is among the reasons why "learning to plan" has been an active research area to address these shortcomings [148, 87]. To be useful in practice we propose that methods that enable to learn planners should have at least two properties: they should be *traces free*, i.e. not require traces from an optimal planner, and they should *generalize*, i.e. learn planners that are able to function on plans of the same type but of unseen instance and/or planning horizons.

In Reinforcement Learning (RL), learning to plan can be framed as the problem of finding a policy that maximises the expected return from the environment, where such policy is a greedy function that selects actions that will visit states with a higher value for the agent. This in turns shifts the problem to the one of obtaining good estimates of state values. One of the most commonly used algorithms to solve this problem is Value Iteration (VI), which estimates the state values by collecting and propagating the observed rewards until a fixed point is reached. A policy – or a plan – can then be constructed by rolling out the obtained value function on the desired state-action pairs.

When the environment can be represented as an occupancy map, a 2D grid, it is possible to approximate this planning algorithm using a deep convolutional neural network (CNN) to propagate the rewards on the grid cells. This enables

one to differentiate directly through the planner steps and perform end-to-end learning of the value function. Tamar et al. [176] train such models – Value Iteration Networks (VIN) – with a supervised loss on the trace from a search/planning algorithm, with the goal to find the parameters that can solve the shortest path task in such environments by iteratively learning the value function using the convnet. However, this baseline requires good target value estimates, violating our wished trace free property, and limiting its usage in interactive, dynamic settings. Furthermore, it doesn't take advantage of the model structure to generalise to harder instances of the task.

In this work we extend the formalization used in VIN to more accurately represent the structure of grid-world-like scenarios, enabling Value Iteration network modules to be naturally used within the reinforcement learning framework beyond the scope of the initial work, while also removing some of the limitations and underlying assumptions constraining the original architecture. We show that our models can not only learn to plan and navigate in dynamic environments, but that their hierarchical structure provides a way to generalize to navigation tasks where the required planning horizon and the size of the map are much larger than the ones seen at training time. Our main contributions include: (1) introducing VProp and MVProp, network planning modules which successfully learn to solve pathfinding tasks via reinforcement learning using minimal parametrization, (2) demonstrating the ability to generalize to large unseen maps when training exclusively on much smaller ones, and (3) showing that our modules can learn to plan in environments with more complex dynamics than a static grid world, both in terms of transition function and observation complexity.

4.1 Related work

Model-based planning with end-to-end architectures has recently shown promising results on a variety of tasks and environments, often using Deep Reinforcement Learning as the algorithmic framework [158, 124, 197, 61, 51, 156, 163]. 3D and 2D navigation tasks have also been tackled within the RL framework [115], with methods in some cases building and conditioning on 2D occupancy maps to aid the process of localization and feature grounding [19, 207, 10].

Other work has furthermore explored the usage of VIN-like architectures for navigation problems: [123] present a generalization of VIN able to learn modules on more generic graph structures by employing a graph convolutional operator to convolve through each node of the graph. [143] demonstrate a method for

multi-agent planning in a cooperative setting by training multiple VI modules and composing them into one network, while also adding an orientation state channel to simulate non-holonomic constraints often found in mobile robotics. [63] and [92] propose to tackle partially observable settings by constructing hierarchical planners that use VI modules in a multi-scale fashion to generate plans and condition the model's belief state.

4.2 Background

We consider the control of an agent in a "grid world" environment, in which entities can interact with each other. The entities have some set of attributes, including a uniquely defined type, which describes how they interact with each other, the immediate rewards of such interactions, and how such interactions affect the next state of the world. The goal is to *learn to plan* through reinforcement learning, that is learning a policy trained on various configurations of the environment that can generalize to arbitrary other configurations of the environment, including larger environments, and ones with a larger number of entities. In the case of a standard navigation task, this boils down to learning a policy which, given an observation of the world, will output actions that take the agent to the goal as quickly as possible. An agent may observe such environments as 2D images of size $d_{\mathrm{x}} \times d_{\mathrm{y}}$, with d_{pix} input panes, which are then potentially passed through an embedding function Φ (such as a 2D convnet) to extract the entities and generates some local embedding based on their positions and features.

4.2.1 Reinforcement Learning

The problem of reinforcement learning is typically formulated in terms of computing optimal policies for a Markov Decision Problem (MDP) [170]. Note that Chapter 2.1 contains an overview of the topic, but we provide a refresher to make it easier to follow the next few sections.

An MDP is defined by the tuple (S, A, T, R, γ), where S is a finite set of states, A is the set of actions a that the agent can take, $T : s \to a \to s'$ is a function describing the state-transition matrix, R is a reward function, and γ is a discount factor. In this setting, an optimal policy π^* is a distribution over the state-action space that maximises in expectation the discounted sum of rewards $\sum_k \gamma^k r_k$, where r_k is the single-step reward. A standard method to find the optimal policy $\pi : s \to a$ is to iteratively compute the value function, $Q^\pi(s, a)$, updating it based on rewards received from the environment [195]. Using this framework, we can view learning

to plan as a *structured prediction* of rewards with the planning algorithm Value Iteration [18] as inference procedure. Beside value-based algorithms, there exist other types which are able to find optimal policies, such as *policy gradient* methods [171], which directly regress to the policy function π instead of approximating the value function. These methods however suffer from high variance estimates in environments that require many steps. Finally, a third type is represented by actor-critic algorithms, which combine the policy gradient methods' advantage of being able to compute the policy directly, with the low-variance performance estimation of value-based RL used as a more accurate feedback signal to the policy estimator [95].

4.2.2 Value Iteration Module

Tamar et al. [176] motivate the *Value Iteration module* by observing that for problems such as navigation, and more generally pathfinding, value iteration can be unrolled as a graph convolutional network, where nodes are possible positions of the agent and edges represent possible transitions given the agent's actions. In the simpler case of 2D grids, the graph structure corresponds to a neighborhood in the 2D space, and the convolution structure is similar to a convolutional network that takes the entire 2D environment as input.

More precisely, let us denote by s the current observation of the environment (e.g., a bird's-eye view of the 2D grid), q^0 the zero tensor of dimensions $(A, d_{\mathrm{x}}, d_{\mathrm{y}})$, where $d_{\mathrm{x}}, d_{\mathrm{y}}$ are the dimension of the 2D grid, and A the number of actions of the agent. Then, the Value Iteration module is defined by an embedding function of the state $\Phi(s) \in \mathbb{R}^{d_{\mathrm{rew}} \times d_{\mathrm{x}} \times d_{\mathrm{y}}}$ (where d_{rew} depends on the model), a transition function h, and performs the following computations for steps $k = 1...K$ where K is the depth of the VI module:

$$\begin{aligned} \forall (i,j) \in [\![\, d_{\mathrm{x}} \,]\!] \times [\![\, d_{\mathrm{y}} \,]\!], \quad v_{ij}^k &= \max_{a=1..A} q_{a,i,j}^k, \\ q^k &= h(\Phi(s), v^{k-1}) \,. \end{aligned}$$

Given the agent's position (x_0, y_0) and the current observation of the environment s, the control policy π is then defined by $\pi(s, (x_0, y_0)) = \mathrm{argmax}_{a=1..A}\, q_{a,x_0,y_0}^K$.

We can rewrite the transition function h as a convolutional layer:

$$\begin{aligned} &\text{(a)} && \bar{r}_{i,j} = \Phi(s)_{i,j}, \\ &\text{(b)} && v_{i,j}^0 = 0, \\ & && q_{a,i,j}^k = \sum_{(i',j') \in \mathcal{N}(i,j)} p_{a,i'-i,j'-j}^{(v)} * v_{i',j'}^{k-1} + p_{a,i'-i,j'-j}^{(r)} * \bar{r}_{i',j'}, \\ & && v_{i,j}^k = \max_a q_{a,i,j}^{k-1}, \end{aligned}$$

where $\mathcal{N}(i,j)$ is the set of neighbors of the cell (i,j) and the cell itself. In practice, $\Phi(s)$ has several output channels d_{rew} and a varying number of parameters when used within a VI module on different tasks. $p^{(v)}_{a,i'-i,j'-j} \in \mathbb{R}$ and $p^{(r)}_{a,i'-i,j'-j} \in \mathbb{R}$ instead purely represent the parameters of the VI module.

This model is appealing as a neural network architecture because the step to compute each $q^k_{a,i,j}$ is a convolutional layer, thus enabling computing $q^K_{a,i,j}$ with a convolutional network of depth K with shared weights and as many output channels as actions. The computation of $v^k_{i,j}$ are then the non-linearities of the network, which correspond to a max-pooling in the dimension of the output channels.

To clarify the relationship with the original value iteration algorithm for grid worlds, let us denote by $R_{a,i,j,i',j'}$ the immediate reward when taking action a at position (i,j) with arrival position (i',j'), $P_{a,i,j,i',j'}$ the corresponding transition probabilities, and γ the discount factor. Then value iteration in a 2D grid can be generally written as:

$$\begin{aligned} \forall (i,j) \in [\![\, d_{\text{x}} \,]\!] \times [\![\, d_{\text{y}} \,]\!], \;\; V^k_{ij} &= \max_{a=1..A} Q^k_{a,i,j}, \\ \forall (a,i,j) \in [\![\, A \,]\!] \times [\![\, d_{\text{x}} \,]\!] \times [\![\, d_{\text{y}} \,]\!], \;\; Q^k_{a,i,j} &= \sum_{(i',j') \in \mathcal{N}(i,j)} P_{a,i,j,i',j'} \left(R_{a,i,j,i',j'} + \gamma V^{k-1}_{i,j} \right). \end{aligned}$$

where $\mathcal{N}(i,j)$ is prior knowledge about the states that are accessible from (i,j). The implementation of the value iteration module described above thus corresponds to a specific parameterization of the reward $R_{a,i,j,i',j'}$ as functions of the starting and arrival states. More importantly, the implementation of the value iteration module with a convolutional network means that the transition probabilities, represented by the parameters $p^{(v)}_{a,i'-i,j'-j}$, are *translation-invariant*.

4.3 Models

The value iteration modules is appealing from a computational point of view because they can be efficiently implemented as convolutional neural networks. They are also conceptually appealing for the design of neural planning architectures because they give a clear motivation and interpretation for sharing weights between layers, and provide guidance as to the necessary depth of the network: the depth should be sufficient for the reward signals to propagate from "goal" states to the agent, and thus be some function of the length of the shortest path.

While the parametrization with shared weights reduces sample complexity, this architecture is also amenable to an interesting form of generalization: learning to navigate in small environments (small $d_{\text{x}}, d_{\text{y}}$ for training), and generalize to larger

instances (larger $d_{\mathrm{x}}, d_{\mathrm{y}}$). That is, from the analogy with value iteration it follows that generalization to larger instances should require deeper networks, which is possible because the weights are shared. Yet, the experiments in Tamar et al. [176], as well as our experiments presented in Section 4.4, show that learning VI modules with reinforcement learning remains very challenging and doesn't naturally generalize as thought, suggesting that the sample complexity is still a major issue.

We propose in this section two alternative approaches to the value iteration modules described above, with the objective to provide a minimal parametrization for better sample complexity and generalization, while keeping the convolutional structure and the idea of a deep network with shared weights. We mostly revisit the two design choices made in the VI module: first, we drop the main assumption of VI modules that the transition probabilities are translation-invariant by making them a function of the state, and look for the minimal parametrization of these state-dependent transition probabilities suitable for grid worlds. Second, we propose more constrained parametrizations of the immediate reward function to increase sample efficiency using reinforcement learning, thus allowing to exploit the structure of the architecture to achieve generalization.

4.3.1 Value-Propagation Module

Let us observe that in the simplest version of a grid world the dynamics are deterministic and actions only consist of moving into an adjacent cell. The world model should account for blocking cells (e.g. terrain or obstacles), while the reward function should account for goal states. Furthermore – and very importantly when implementing value iteration for episodic tasks – one needs to account for terminal states, which can be represented by absorbing states, i.e. states for which the only possible transition is a loop to itself with reward 0 [170].

We propose to take these elements into account in the following architecture, which we call *Value Propagation* (VProp). Similarly to a VI module, it is here implemented with deep convolutional networks with weights shared across layers:

$$
\begin{aligned}
&\text{(a)} \qquad \bar{r}^{\mathrm{in}}_{i,j}, \bar{r}^{\mathrm{out}}_{i,j}, p_{i,j} = \Phi(s)_{i,j}, \\
&\text{(b)} \qquad v^0_{i,j} = 0, \\
&\qquad\qquad v^{(k)}_{i,j} = \max\left(v^{(k-1)}_{i,j}, \max_{(i',j')\in\mathcal{N}(i,j)}\left(p_{i,j} v^{(k-1)}_{i',j'} + \bar{r}^{\mathrm{in}}_{i',j'} - \bar{r}^{\mathrm{out}}_{i,j}\right)\right), \\
&\text{(c)} \qquad \pi(s,(i_0,j_0)) = \operatorname*{argmax}_{i',j'\in\mathcal{N}(i_0,j_0)} v^K_{i',j'}\,.
\end{aligned}
$$

In VProp, all parameters are in the embedding function Φ – the iteration layers do not have any additional parameters. This embedding function has two types of

output for each position (i, j). The first type is a vector $(\bar{r}^{\text{in}}_{i,j}, \bar{r}^{\text{out}}_{i,j})$, which is used to model the full reward function $R_{a,i,j,i',j'}$ as $\bar{r}^{\text{in}}_{i',j'} - \bar{r}^{\text{out}}_{i,j}$. This is done to correctly deal with absorbing states: $\bar{r}^{\text{in}}$ intuitively models the reward obtained when the agent enters position i, j (typically high for a goal state, negative for positions we should avoid), and $\bar{r}^{\text{out}}$ is the cost of going out of a position. Absorbing states can then be represented by setting $\bar{r}^{\text{in}}_{i,j} = \bar{r}^{\text{out}}_{i,j}$.

The second output given by Φ is a single value $p_{i,j}$, which represent a *propagation* parameter associated to the position, or simply a state-dependent discount factor [199]: $p_{i,j} \approx 1$ means that the neighboring values $v_{i',j'}$ propagate through i, j, while $p_{i,j} \approx 0$ means that position i, j is blocking, which typically arises for cells containing obstacles. In our implementations, all $\bar{r}^{\text{in}}$, $\bar{r}^{\text{out}}$, p are kept in $[0, 1]$ using a sigmoid activation function.

VProp corresponds to the value iteration algorithm in which the dynamics are deterministic and there is a one-to-one mapping between actions and adjacent cells. This mapping is assumed in equation *(c)*, in which the output is a position rather than an action. In practice, since (i', j') is an adjacent cell, it is trivial to know which action needs be taken to get to that cell. However, in case the mapping is unknown, one can use $\pi(s) = F\left([v^{(K)}_{i',j'}]_{(i',j')\in\mathcal{N}(i_0,j_0)}\right)$ for some F that takes the neighborhood of the agent as input and performs the mapping from the propagated values to actual actions.

It's important to stress that this particular architecture was designed for environments that can be represented and behave like 2D grid structures (e.g. robotic navigation), however the formulation can be easily extended to more general graph structures. More details on this can be found in Appendix B.1.

4.3.2 Max-Propagation Module

One of the main difficulties we observed with both the VI module and VProp when generalizing to larger environments is that obstacles/blocking cells on a fixed size grid can be represented in two different ways: either with a small value of propagation or with a large output reward. In practice, a network trained on grids of bounded sizes may learn to any valid configuration, but a configuration based on negative rewards and high propagation does not generalize to larger environments: a negative reward for going through an obstacle is not, in general, sufficient to compensate for the length of a correct path that needs to get around the obstacle. When the environment has fixed size, the maximum length of going around of an obstacle is known, and the reward can be set accordingly. But as the environment

increases in size, the cost of getting around the obstacle increases and is not well represented by the negative reward anymore.

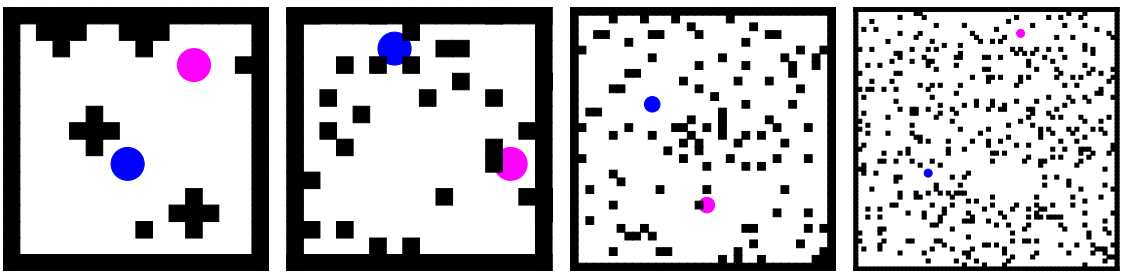
To overcome this difficulty, we propose an alternative formulation, the Max-Propagation module (MVProp), in which only positive rewards are propagated. This implies that the only way to represent blocking paths is by not propagating rewards – negative rewards are not a solution anymore. The MVProp module is defined as follows:

$$
\begin{aligned}
&\text{(a)} && \bar{r}_{i,j}, p_{i,j} = \Phi(s)_{i,j}, \\
&\text{(b)} && v^0_{i,j} = \bar{r}_{i,j}, \\
& && v^{(k)}_{i,j} = \max\left(v^{k-1}_{i,j}, \max_{(i',j')\in\mathcal{N}(i,j)}\left(\bar{r}_{i,j} + p_{i,j}(v^{(k-1)}_{i',j'} - \bar{r}_{i,j})\right)\right), \\
&\text{(c)} && \pi(s,(i_0,j_0)) = \operatorname*{argmax}_{i',j'\in\mathcal{N}(i_0,j_0)} v^K_{i',j'}\,.
\end{aligned}
$$

Like VProp, MVProp is another implementation of value iteration with a deterministic mapping from the agent actions to position, however this time the model is constrained to propagate only positive rewards, since the propagation equation $\bar{r}_{i,j} + p_{i,j}(v^{(k-1)}_{i',j'} - \bar{r}_{i,j})$ can be rewritten as $p_{i,j}v^{(k-1)}_{i',j'} + \bar{r}_{i,j}(1 - p_{i,j})$ so the major difference with VProp is that MVProp focuses only on propagating positive rewards. Note therefore that all other remarks concerning more general versions of VProp also apply to MVProp.

4.4 Experiments

We focus on evaluating our modules strictly using Reinforcement Learning, since we are interested in moving towards scenarios that better simulate tasks requiring



(a) VIN **(b)** 14x14 MazeBase **(c)** 32x32 MazeBase **(d)** 64x64 MazeBase

Figure 4.1: Comparison between a random map of the VIN dataset, and a few random configuration of our training environment. In our custom grid-worlds, the number of blocks increases with size, but their percentage over the total available space is kept fixed. Agent and goal are shown as circles for better visualization, however they still occupy a single cell.

interaction with the environment. The architecture contains the policy π_θ described in the previous sections, together with a value function V_w, which takes the same input as the softmax layer of the policy, concatenated with the 3×3 neighborhood of the agent. w and θ share all their weights until the end of the convolutional recurrence. At training time, given the stochastic policy at time step t denoted by π_{θ^t}, we sample a minibatch of B transitions, denoted $\mathcal{B}$, uniformly at random from the last L transitions, and perform gradient ascent over importance-weighted rewards:

$$\begin{aligned}\theta^{t+1} \leftarrow & \theta^t + \eta \sum_{(s,a,r,p,s') \in \mathcal{B}} \min\Big(\frac{\pi_{\theta^t}(s,a)}{p(a)}, C\Big)\Big(r + \mathop{\mathbf{1}}_{\{s' \neq \emptyset\}} \gamma V_{w^t}(s') - V_{w^t}(s)\Big)\Big(\nabla_{\theta^t} \log \pi_{\theta^t}(s,a)\Big) \\ & + \lambda \sum_{(s,a,r,p,s') \in \mathcal{B}} \sum_{a'} p(a')\Big(\nabla_{\theta^t} \log \pi_{\theta^t}(s,a')\Big), \\ w^{t+1} \leftarrow & w^t - \eta' \sum_{(s,a,r,p,s') \in \mathcal{B}} \min\Big(\frac{\pi_{\theta^t}(s,a)}{p(a)}, C\Big)\Big(V_{w^t}(s) - r - \mathop{\mathbf{1}}_{\{s' \neq \emptyset\}} \gamma V_{w^t}(s')\Big)\nabla_{w^t} V_{w^t}(s)\,,\end{aligned}$$

where $\mathop{\mathbf{1}}_{\{s' \neq \emptyset\}}$ is 1 if s' is terminal and 0 otherwise. The capped importance weights $\min\left(\frac{\pi_{\theta^t}(s,a)}{p(a)}, C\right)$ are standard in off-policy policy gradient [193]. The capping constant ($C = 10$ in our experiments) controls the variance of the gradients at the expense of some bias. The second term of the update acts as a regularizer and forces the current predictions to be close enough the the ones that were made by the older model. The learning rates η, λ and η' also control the relative weighting of the different objectives when the weights are shared.

4.4.1 Grid-world setting

Our experimental setting consists of a 2D grid-world of fixed dimensions where all entities are sampled based on some fixed distribution (Figure 4.1). The agent is allowed to move in all 8 directions at each step, and a terminal state is reached when the agent either reaches the goal or hits one of the walls. We use MazeBase [166] to generate the configurations of our world and the agent interface for both training and testing phases. Additionally we also evaluate our trained agents on maps uniformly sampled from the 16×16 dataset originally used by Tamar et al. [176], so as to get a direct comparison with the previous work, and to confirm the quality of our baseline. We tested all the models on the other available datasets (8×8 and 28×28) too, without seeing significant changes in relative performance, so they are omitted from our evaluation. We employ a curriculum where the average length of the optimal path from the starting agent position is bounded by some value which gradually increases after a few training episodes. This makes it more likely to encounter the goal at early stages of training, allowing for easier

(a) (b) (c)

Figure 4.2: Average, min and max reward of all the models as they train on our curriculum. Note again that in the first two plots the the map size is 32×32. a and c demonstrate performances respectively on the VIN dataset and our generated 64×64 maps. b shows performance on evaluation maps constrained by the curriculum settings (segmented line), and without (continuous line).

conditioning over the goal feature. Across all our tests on this setup, both VProp and MVProp greatly outperformed our implementation of VIN. Figure 4.2 shows rewards obtained during training, averaged across 5 training runs seeded randomly. It's important to note that the original VIN architecture was mostly tested in a fully supervised setting (via imitation learning), where the best possible route was given to the network as target. In the appendix, however, [176] claim that VIN can perform in a RL setting, obtaining an 82.5% success rate, versus the 99.3% success rate of the supervised setting on a map of 16×16. The authors do not provide results for the larger 28×28 map dataset, nor do they provide learning curves and variance, however overall these results are consistent with the *best* performance we obtained from testing our implementation.

The final average performances of each model against the static-world experiments clearly demonstrate the strength of VProp and MVProp. In all the above experiments, both VProp and MVProp very quickly outperform the baseline. In particular MVProp very quickly learns a transition function over the MDP dynamics that is sharp enough to provide good values across even bigger sizes, hence obtaining near-optimal policies over all the sizes during the first thousand training episodes.

4.4.2 Tackling dynamic environments

To test the capability of our models to effectively learn non-static environments - that is, relatively more complex transition functions - we propose a set of experiments in which we allow our environment to spawn dynamic adversarial entities controlled by a set of fixed policies. These policies include a ϵ-*noop* strategy, which makes

the entity move in random direction with probability ϵ or do nothing, a ϵ-direction policy, which makes the entity move to a specific direction with probability ϵ or do nothing, and finally strictly adversarial policies that try to catch the agent before it can reach the goal. We use the first category of policies to augment our standard path-planning experiments, generating *enemies_only* environments where 20% of the space is occupied by agents with $\epsilon = 0.5$, and *mixed* environments with the same amount of entities, half consisting of fixed walls, and the remaining of agents with $\epsilon = 0.2$ The second type of policies is used to generate a deterministic but continuously changing environment which we call *avalanche*, in which the agent is tasked to reach the goal as quickly as possible while avoiding "falling" entities. Finally, we propose a third type of experiments where the latter fully adversarial policy is applied on 1 to 6 enemy entities depending on the size of the environment. These scenarios differ in difficulty, but all require the modules to both learn off extremely sparse positive rewards and deal with a more complex transition function, thus presenting a strong challenge for all the tested methods.

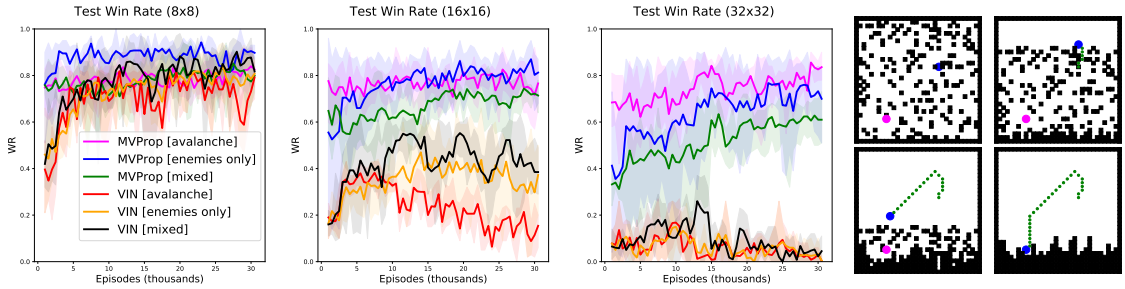
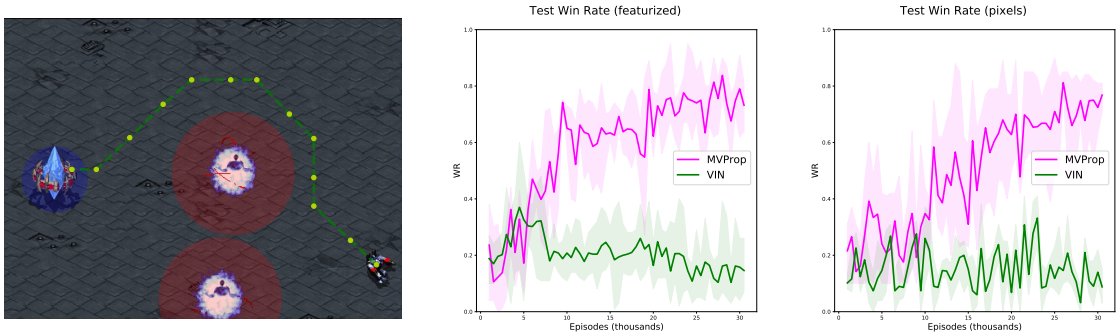


Figure 4.3: Average, min, and max, test win rate obtained on our dynamics experiments. Each agent was trained on the 8x8 instances of the scenario in a similar fashion to the static world experiments. Figure 4.3d shows an example of policy obtained after training on a avalanche testing configuration. Agent and goal are shown as circles for better visualization, however they still occupy a single cell.

As these new environments are not static, the agent needs to re-plan at every step, forcing us to train on 8x8 maps to reduce the time spent rolling-out the recurrent modules. This however allows us to train without curriculum, as the agent is already likely to successfully hit the goal in a smaller area with a stochastic policy. Figure 4.3 shows that MVProp learns to handle this new complexity in the dynamics, successfully generalising to 32x32 maps, much larger than the ones seen during training (Figure 4.3d), significantly outperforming the baseline.

4.4.3 StarCraft navigation

Finally, we evaluate VProp on a navigation task in *StarCraft: Brood War* where the navigation-related actions have low-level physical dynamics that affect the transition function. In StarCraft, it is common to want to plan a trajectory around enemy units, as these have auto-attack behaviours that will interfere or even destroy your own unit if found navigating too close to them. While planning trajectories of the size of a standard StarCraft map is outside of the scope of this work, this problem becomes already quite difficult when considering scenarios where enemies are close to the bottleneck areas, thus we can test our architecture on small maps to simulate instances of these scenarios.



(a) (b) (c)

Figure 4.4: StarCraft navigation results. Figure 4.4a shows a generated trajectory on a random scenario at late stages of training. The red and blue overlays (not shown to the agent) indicate the distance required to interact with each enemy entity.

We use TorchCraft [172] to setup the environment and extract the position and type of units randomly spawned in the scenario. The state space is larger than the one used in our previous experiments and positive rewards might be extremely sparse, thus we employ a mixed curriculum to sample the units and their positions, allowing the models to observe positive rewards more often at the early stages of training and speed up training (note that this is also a requirement for the VIN baseline to have a chance at the task [176]). As shown in Figure 4.4b, the MVProp is strong enough to plan around the low-level noise of the move actions, which enables the agent to navigate around the lethal enemies and reach the goal on most random instances after training. Compared to the VIN baseline, the better sample efficiency translates into the ability to more accurately learn a model of the state-action transition function, which ultimately allows VProp to learn planning modules for this non-trivial environment. We also evaluate on the scenario directly from pixels, by adding two convolutional layers following by a max-pooling operator to provide capacity to

learn the state features: as expected (Figure 4.4c) the architecture takes some more time to condition on the entities, but ultimately reaches similar final performances. In both cases, VIN struggles to condition correctly on the more complex transition function even with the curriculum providing early positive feedback.

4.5 Conclusions

Architectures that try to solve the large but structured space of navigation tasks have much to benefit from employing planners that can be learnt from data, however these need to be sample efficient to quickly adapt to local environment dynamics so that they can provide a flexible planning horizon without the need to collect new data. Our work shows that such planners can be successfully learnt via Reinforcement Learning when the dynamics of the task are taken into account, and that great generalization capabilities can be expected when these models are applied to 2D path-planning tasks. Furthermore, we have demonstrated that our methods can even generalize when the environment has dynamic, noisy, and adversarial elements, or with high-dimensional observation spaces, enabling them to be employed in relatively complex tasks. A major issue that still prevents these planners from being deployed on harder tasks is computational cost, since the depth increases with the length of the path that agents must solve, however architectures employing VI modules as low level planners have been successfully tackling complex interactive tasks (Section 4.1), thus we expect our methods to provide a way for such type of work to train end-to-end via reinforcement learning, even for pathfinding tasks found in different graph-like structures (for which we have at least the relevant convolutional operators). Finally, interesting venues where VProp and MVProp may be applied are mobile robotics and visual tracking [104, 20], where our work could be used to learn arbitrary propagation functions, and model a wide range of potential functions.

5

The NetHack Learning Environment

Recent advances in (Deep) Reinforcement Learning (RL) have been driven by the development of novel simulation environments, such as the Arcade Learning Environment (ALE) [15], StarCraft [172, 191], BabyAI [31], Obstacle Tower [84], Minecraft [81, 64, 77], and Procgen Benchmark [36]. These environments introduced new challenges for state-of-the-art methods and demonstrated failure modes of existing RL approaches. For example, *Montezuma's Revenge* highlighted that methods performing well on other ALE tasks were not able to successfully learn in this sparse-reward environment. This sparked a long line of research on novel methods for exploration [e.g., 14, 181, 129] and learning from demonstrations [e.g., 73, 149, 9]. However, this progress has limits: the current best approach on this environment, Go-Explore [47, 48], overfits to specific properties of ALE and Montezuma's Revenge. While Go-Explore is an impressive solution for Montezuma's Revenge, it exploits the determinism of environment transitions, allowing it to memorize sequences of actions that lead to previously visited states from which the agent can continue to explore.

We are interested in surpassing the limits of deterministic or repetitive settings and seek a simulation environment that is complex and modular enough to test various open research challenges such as exploration, planning, skill acquisition, memory, and transfer. However, since state-of-the-art RL approaches still require millions or even billions of samples, simulation environments need to be fast to allow RL agents to perform many interactions per second. Among attempts to surpass the limits of deterministic or repetitive settings, *procedurally generated environments* are a promising path towards testing systematic generalization of RL

methods [e.g., 85, 84, 145, 36]. Here, the game state is generated programmatically in every episode, making it extremely unlikely for an agent to visit the exact state more than once during its lifetime. Existing procedurally generated RL environments are either costly to run [e.g., 191, 81, 84] or are, as we argue, of limited complexity [e.g., 32, 37, 11].

To address these issues, we present the `NetHack Learning Environment` (`NLE`), a procedurally generated environment that strikes a balance between complexity and speed. It is a fully-featured *Gym* environment [22] around the popular open-source terminal-based single-player turn-based "dungeon-crawler" game, NetHack [91]. Aside from procedurally generated content, NetHack is an attractive research platform as it contains hundreds of enemy and object types, it has complex and stochastic environment dynamics, and there is a clearly defined goal (descend the dungeon, retrieve an amulet, and ascend). Furthermore, NetHack is difficult to master for human players, who often rely on external knowledge to learn about strategies and NetHack's complex dynamics and secrets.[1] Thus, in addition to a guide book [141, 142] released with NetHack itself, many extensive community-created documents exist, outlining various strategies for the game [e.g., 121, 50].

In summary, we make the following core contributions:

(i) we present `NLE`, a fast but complex and feature-rich Gym environment for RL research built around the popular terminal-based game, NetHack,

(ii) we release an initial suite of tasks in the environment and demonstrate that novel tasks can be added easily,

(iii) we introduce baseline models trained using IMPALA [49] and Random Network Distillation (RND) [24], a popular exploration bonus, resulting in agents that learn diverse policies for early stages of NetHack, and

(iv) we demonstrate the benefit of NetHack's symbolic observation space by presenting in-depth qualitative analyses of trained agents.

[1] "NetHack is largely based on discovering secrets and tricks during gameplay. It can take years for one to become well-versed in them, and even experienced players routinely discover new ones." [52]

5.1 NetHack: a Frontier for Reinforcement Learning Research

In traditional so-called *roguelike* games (e.g., Rogue, Hack, NetHack, and Dungeon Crawl Stone Soup) the player acts turn-by-turn in a procedurally generated grid-world environment, with game dynamics strongly focused on exploration, resource management, and continuous discovery of entities and game mechanics.These games are designed to provide a steep learning curve and a constant level of challenge and surprise to the player. They are generally extremely difficult to win even once, let alone to master, i.e., win regularly and multiple times in a row.

As advocated by [85, 84, 36], procedurally generated environments are a promising direction for testing systematic generalization of RL agents. We argue that such environments need to be both sufficiently complex and fast to run to serve as a challenging long-term research testbed. In Section 5.1.1, we illustrate that NetHack contains many desirable properties, making it an excellent candidate for driving long-term research in RL. We introduce `NLE` in Section 5.1.2, an initial suite of tasks in Section 5.1.3, an evaluation protocol for measuring progress towards *solving* NetHack in Section 5.1.4, as well as baseline models in Section 5.1.5.

5.1.1 NetHack

NetHack is one of the oldest and most popular roguelikes, originally released in 1987 as a successor to *Hack*, an open-source implementation of the original *Rogue* game. At the beginning of the game, the player takes the role of a hero who is placed into a dungeon and tasked with finding the *Amulet of Yendor* to offer it to an in-game deity. To do so, the player has to descend to the bottom of over 50 procedurally generated levels to retrieve the amulet and then subsequently escape the dungeon, unlocking five extremely challenging final levels (the four Elemental Planes and the Astral Plane).

Many aspects of the game are procedurally generated and follow stochastic dynamics. For example, the overall structure of the dungeon is somewhat linear, but the exact location of places of interest (e.g., the *Oracle*) and the structure of branching sub-dungeons (e.g., the *Gnomish Mines*) are determined randomly. The procedurally generated content of each level makes it highly unlikely that a player will ever experience the exact same situation more than once. This provides a fundamental challenge to learning systems and a degree of complexity that enables us to more effectively evaluate an agent's ability to generalize. It also disqualifies current state-of-the-art exploration methods such as Go-Explore [47, 48] that are based on

Figure 5.1: Annotated example of an agent at two different stages in NetHack (Left: a procedurally generated first level of the Dungeons of Doom, right: Gnomish Mines). A larger version of this figure is displayed in Figure C.6 in the appendix.

a goal-conditioned policy to navigate to previously visited states. Moreover, states in NetHack are composed of hundreds of possible symbols, resulting in an enormous combinatorial observation space.[2] It is an open question how to best project this symbolic space to a low-dimensional representation appropriate for methods like Go-Explore. For example, Ecoffet et al.'s heuristic of downsampling images of states to measure their similarity to be used as an exploration bonus will likely not work for large symbolic and procedurally generated environments. NetHack provides further variation by different hero roles (e.g., monk, valkyrie, wizard, tourist), races (human, elf, dwarf, gnome, orc) and random starting inventories (see Appendix C.1 for details). Consequently, NetHack poses unique challenges to the research community and requires novel ways to determine state similarity and, likely, entirely new exploration frameworks.

To provide a glimpse into the complexity of NetHack's environment dynamics, we closely follow the educational example given by "Mr Wendal" on YouTube.[3] At a specific point in the game, the hero has to get past *Medusa's Island* (see Figure 5.2 for an example). Medusa's Island is surrounded by water } that the agent has to cross. Water can rust and corrode the hero's metallic weapons) and armor [. Applying a can of grease (prevents rusting and corrosion. Furthermore, going into water will make a hero's inventory wet, erasing scrolls ? and spellbooks + that they carry. Applying a can of grease to a bag or sack (will make it a waterproof container for items. But the sea can also contain a kraken ; that can grab and drown the hero, leading to instant death. Applying a can of grease to a hero's

[2] Information about the over 450 items and 580 monster types, as well as environment dynamics involving these entities can be found in the NetHack Wiki [121] and to some extent in the NetHack Guidebook [142].

[3] youtube.com/watch?v=SjuTyJlgLJ8

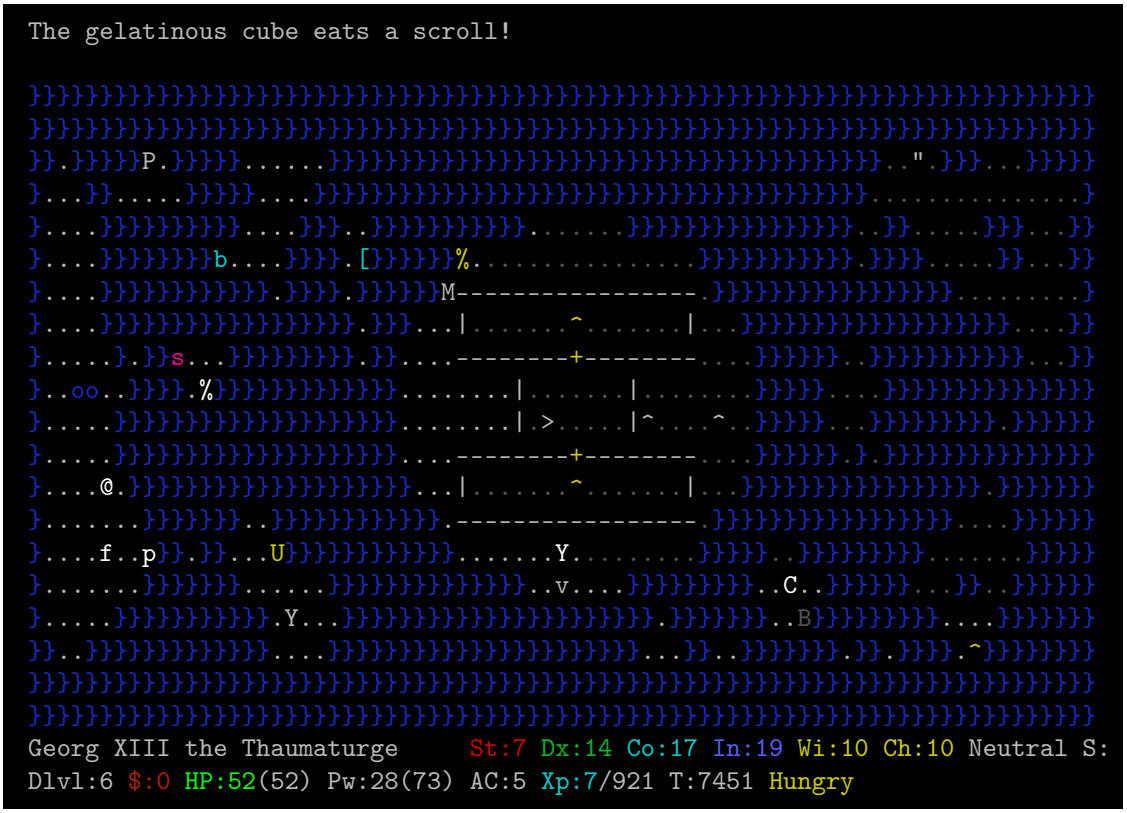


Figure 5.2: The hero (@) has to cross water (}) to get past Medusa (@, out of the hero's line of sight) down the staircase (>) to the next level.

armor prevents the kraken from grabbing the hero. However, a cursed can of grease will grease the hero's hands instead and they will drop their weapon and rings. One can use a towel (to wipe off grease. To reach Medusa @, the hero can alternatively use magic to freeze the water and turn it into walkable ice ·. Wearing snow boots [will help the hero not to slip. When Medusa is in the hero's line of sight, her gaze will petrify and instantly kill—the hero should use a towel to cover their eyes to fight Medusa, or even apply a mirror (to petrify her with her own gaze.

There are many other entities a hero must learn to face, many of which appear rarely even across multiple games, especially the most powerful monsters. These entities are often compositional, for example a monster might be a wolf d, which shares some characteristics with other in-game canines such as coyotes d or hell hounds d. To help a player learn, NetHack provides in-game messages describing many of the hero's interactions (see the top of Figure 5.1).[4] Learning to capture

[4]An example interaction after applying a figurine of an Archon: "*You set the figurine on the ground and it transforms. You get a bad feeling about this. The Archon hits! You are blinded by the Archon's radiance! You stagger... It hits! You die... But wait... Your medallion feels warm! You feel much better! The medallion crumbles to dust! You survived that attempt on your life.*"

these interesting and somewhat realistic albeit abstract dynamics poses challenges for multi-modal and language-conditioned RL [110].

NetHack is an extremely long game. Successful expert episodes usually last tens of thousands of turns, while average successful runs can easily last hundreds of thousands of turns, spawning multiple days of play-time. Compared to testbeds with long episode horizons such as StarCraft and Dota 2, NetHack's "episodes" are one or two orders of magnitude longer, and they wildly vary depending on the policy. Moreover, several official *conducts* exist in NetHack that make the game even more challenging, e.g., by not wearing any armor throughout the game (see Appendix C.1 for more).

Finally, in comparison to other classic roguelike games, NetHack's popularity has attracted a larger number of contributors to its community. Consequently, there exists a comprehensive game wiki [121] and many so-called spoilers [50] that provide advice to players. Due to the randomized nature of NetHack, this advice is general in nature (e.g., explaining the behavior of various entities) and not a step-by-step guide. These texts could be used for language-assisted RL along the lines of [209]. Lastly, there is also a large public repository of human replay data (over five million games) hosted on the NetHack Alt.org (NAO) servers, with hundreds of finished games per day on average [111]. This extensive dataset could spur research advances in imitation learning, inverse RL, and learning from demonstrations [1, 5].

5.1.2 The NetHack Learning Environment

The `NetHack Learning Environment` (`NLE`) is built on NetHack 3.6.6, the 36th public release of NetHack, which was released on March 8th, 2020 and is the latest available version of the game at the time of publication of this paper. `NLE` is designed to provide a common, turn-based (i.e., synchronous) RL interface around the standard terminal interface of NetHack. We use the game *as-is* as the backend for our `NLE` environment, leaving the game dynamics unchanged. We added to the source code more control over the random number generator for seeding the environment, as well as various modifications to expose the game's internal state to our Python frontend.

By default, the observation space consists of the elements *glyphs*, *chars*, *colors*, *specials*, *blstats*, *message*, *inv_glyphs*, *inv_strs*, *inv_letters*, as well as *inv_oclasses*. The elements *glyphs*, *chars*, *colors*, and *specials* are tensors representing the (batched) 2D symbolic observation of the dungeon; *blstats* is a vector of agent coordinates and other character attributes ("bottom-line stats", e.g., health points, strength, dexterity, hunger level; normally displayed in the bottom area of the GUI), *message*

is a tensor representing the current message shown to the player (normally displayed in the top area of the GUI), and the *inv_** elements are padded tensors representing the hero's inventory items. More details about the default observation space and possible extensions can be found in Appendix C.2.

The environment has 93 available actions, corresponding to all the actions a human player can take in NetHack. More precisely, the action space is composed of 77 command actions and 16 movement actions. The movement actions are split into eight "one-step" compass directions (i.e., the agent moves a single step in a given direction) and eight "move far" compass directions (i.e., the agent moves in the specified direction until it runs into some entity). The 77 command actions include *eating*, *opening*, *kicking*, *reading*, *praying* as well as many others. We refer the reader to Appendix C.3 as well as to the NetHack Guidebook [142] for the full table of actions and NetHack commands.

`NLE` comes with a Gym interface [22] and includes multiple pre-defined tasks with different reward functions and action spaces (see next section and Appendix C.5 for details). We designed the interface to be lightweight, achieving competitive speeds with Gym-based ALE (see Appendix C.4 for a rough comparison). Finally, `NLE` also includes a dashboard to analyze NetHack runs recorded as terminal `tty` recordings. This allows `NLE` users to analyze replays of the agent's behavior at an arbitrary speed and provides an interface to visualize action distributions and game events (see Appendix C.8 for details). `NLE` is available under an open source license at https://github.com/facebookresearch/nle.

5.1.3 Tasks

`NLE` aims to make it easy for researchers to probe the behavior of their agents by defining new tasks with only a few lines of code, enabled by NetHack's symbolic observation space as well as its rich entities and environment dynamics. To demonstrate that NetHack is a suitable testbed for advancing RL, we release a set of initial tasks for tractable subgoals in the game: navigating to a **staircase** down to the next level, navigating to a staircase while being accompanied by a **pet**, locating and **eat**ing edibles, collecting **gold**, maximizing in-game **score**, **scout**ing to discover unseen parts of the dungeon, and finding the **oracle**. These tasks are described in detail in Appendix C.5, and, as we demonstrate in our experiments, lead to unique challenges and diverse behaviors of trained agents.

5.1.4 Evaluation Protocol

We lay out a protocol and provide guidance for evaluating future work on `NLE` in a reproducible manner. The overall goal of `NLE` is to train agents that can solve NetHack. An episode in the full game of NetHack is considered solved if the agent retrieves the *Amulet of Yendor* and offers it to its co-aligned deity in the Astral Plane, thereby ascending to demigodhood. We declare `NLE` to be solved once agents can be trained to consecutively ascend (ten episodes without retry) to demigodhood on unseen seeds given a random role, race, alignment, and gender combination. Since the environment is procedurally generated and stochastic, evaluating on held-out unseen seeds ensures we test systematic generalization of agents. As of October 2020, NAO reports the longest *streak* of human ascensions on NetHack 3.6.x to be 61; the role, race, etc. are not necessarily randomized for these ascension streaks. Since we believe that this goal is out of reach for machine learning approaches in the foreseeable future, we recommend comparing models on the score task in the meantime. Using NetHack's in-game score as the measure for progress has caveats. For example, expert human players can solve NetHack while minimizing the score [see 121, "Score" entry, for details]. NAO reports ascension scores for NetHack 3.6.x ranging from the low hundreds of thousands to tens of millions. Although we believe training agents to maximize the in-game score is likely insufficient for solving the game, the in-game score is still a sensible proxy for incremental progress on `NLE` as it is a function of, among other things, the dungeon depth that the agent reached, the number of enemies it killed, the amount of gold it collected, as well as the knowledge it gathered about potions, scrolls, and wands.

When reporting results on `NLE`, we require future work to state the full character specification (e.g., `mon-hum-neu-mal`), all NetHack options that were used (e.g., whether or not *autopickup* was used), which actions were allowed (see Table C.1), which actions or action-sequences were hard-coded (e.g., engraving [see 121, "Elbereth" as an example]) and how many different seeds were used during training. We ask to report the average score obtained on 1000 episodes of randomly sampled and previously unseen seeds. We do not impose any restrictions during training, but at test time any save scumming (i.e., saving and loading previous checkpoints of the episode) or manipulation of the random number generator [e.g., 4] is forbidden.

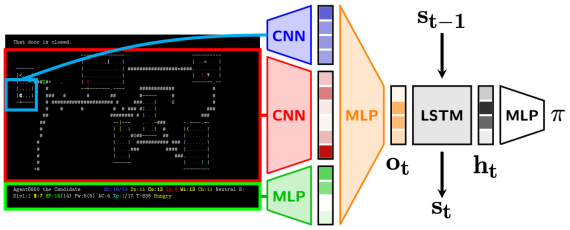


Figure 5.3: Overview of the core architecture of the baseline models released with `NLE`. A larger version of this figure is displayed in Figure C.7 in the appendix.

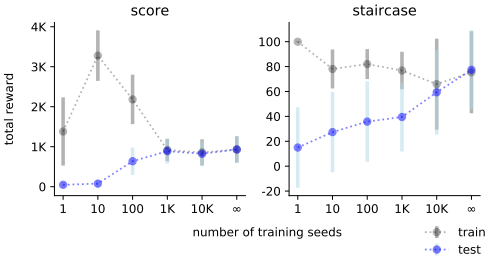


Figure 5.4: Training and test performance when training on restricted sets of seeds.

5.1.5 Baseline Models

For our baseline models, we encode the multi-modal observation o_t as follows. Let the observation o_t at time step t be a tuple (g_t, z_t) consisting of the 21×79 matrix of glyph identifiers and a 21-dimensional vector containing agent stats such as its (x, y)-coordinate, health points, experience level, and so on. We produce three dense representations based on the observation (see Figure 5.3). For every of the 5991 possible glyphs in NetHack (monsters, items, dungeon features, etc.), we learn a k-dimensional vector embedding. We apply a ConvNet (red) to all visible glyph embeddings as well as another ConvNet (blue) to the 9×9 crop of glyphs around the agent to create a dedicated egocentric representation for improved generalization [74, 205]. We found this egocentric representation to be an important component during preliminary experiments. Furthermore, we use an MLP to encode the hero's stats (green). These vectors are concatenated and processed by another MLP to produce a low-dimensional latent representation $\mathbf{o}_t$ of the observation. Finally, we employ a recurrent policy parameterized by an LSTM [75] to obtain the action distribution. For baseline results on the tasks above, we use a reduced action space that includes the movement, search, kick, and eat actions.

For the main experiments, we train the agent's policy for 1B steps in the environment using IMPALA [49] as implemented in TorchBeast [99]. Throughout training, we change NetHack's seed for procedurally generating the environment after every episode. To demonstrate NetHack's variability based on the character configuration, we train with four different agent characters: a neutral human male monk (`mon-hum-neu-mal`), a lawful dwarf female valkyrie (`val-dwa-law-fem`), a chaotic elf male wizard (`wiz-elf-cha-mal`), and a neutral human female tourist (`tou-hum-neu-fem`). More implementation details can be found in Appendix C.6.

In addition, we present results using Random Network Distillation (RND) [24], a popular exploration technique for Deep RL. As previously discussed, exploration

techniques which require returning to previously visited states such as Go-Explore are not suitable for use in NLE, but RND does not have this restriction. RND encourages agents to visit unfamiliar states by using the prediction error of a fixed random network as an intrinsic exploration reward, which has proven effective for hard exploration games such as Montezuma's Revenge [23]. The intrinsic reward obtained from RND can create "reward bridges" between states which provide sparse extrinsic environmental rewards, thereby enabling the agent to discover new sources of extrinsic reward that it otherwise would not have reached. We replace the baseline network's pixel-based feature extractor with the symbolic feature extractor described above for the baseline model, and use the best configuration of other RND hyperparameters documented by the authors (see Appendix C.7 for full details).

5.2 Experiments and Results

We present quantitative results on the suite of tasks included in NLE using a standard distributed Deep RL baseline and a popular exploration method, before additionally analyzing agent behavior qualitatively. For each model and character combination, we present results of the mean episode return over the last 100 episodes averaged for five runs in Figure 5.5. We discuss results for individual tasks below (see Table C.5 in the appendix for full details).

Staircase: Our agents learning to navigate the dungeon to the staircase > with a success rate of 77.26% for the monk, 50.42% for the tourist, 74.62% for the valkyrie, and 80.42% for the wizard. What surprised us is that agents learn to reliably kick in locked doors. This is a costly action to explore as the agent loses health points and might even die when accidentally kicking against walls. Similarly, the agent has to learn to reliably search for hidden passages and secret doors. Often, this involves using the search action many times in a row, sometimes even at many locations on the map (e.g., around all walls inside a room). Since NLE is procedurally generated, during training agents might encounter easier environment instances and use the acquired skills to accelerate learning on the harder ones [145, 36]. With a small probability, the staircase down might be generated near the agent's starting position. Using RND exploration, we observe substantial gains in the success rate for the monk (+13.58pp), tourist (+6.52pp) and valkyrie (+16.34pp) roles, while lower results for wizard roles (−12.96pp).

Pet: Finding the staircase while taking care of the hero's pet (e.g., the starting kitten f or little dog d) is a harder task as the pet might get killed or fall into a trap door, making it impossible for the agent to successfully complete the episode. Compared to the staircase task, the agent success rates are generally lower (62.02% for monk, 25.66% for tourist, 63.30% for valkyrie, and wizard 66.80%). Again, RND exploration provides consistent and substantial gains.

Eat: This tasks highlights the importance of testing with different character classes in NetHack. The monk and tourist start with a number edible items (e.g., food rations %, apples % and oranges %). A sub-optimal strategy is to consume all of these comestibles right at the start of the episode, potentially risking choking to death. In contrast, the other roles have to hunt for food, which our agents learn to do slowly over time for the valkyrie and wizard roles. By having more pressure to quickly learn a sustainable food strategy, the valkyrie learns to outlast other roles and survives the longest in the game (on average 1713 time steps). Interestingly, RND exploration leads to consistently worse results for this task.

Gold: Locating gold $ in NetHack provides a relatively sparse reward signal. Still, our agents learn to collect decent amounts during training and learn to descend to deeper dungeon levels in search for more. For example, monk agents reach dungeon level 4.2 on average for the CNN baseline and even 5.0 using RND exploration.

Score: As discussed in Section 5.1.4, we believe this task is the best candidate for comparing future methods regarding progress on NetHack. However, it is questionable whether a reward function based on NetHack's in-game score is sufficient for training agents to solve the game. Our agents average at a score of 748 for monk, 11 for tourist, 573 for valkyrie, and 314 for wizard, with RND exploration again providing substantial gains (e.g. increasing the average score to 780 for monk). The resulting agents explore much of the early stages of the game, reaching dungeon level 5.4 on average for the monk with the deepest descent to level 11 achieving a high score of 4260 while leveling up to experience level 7 (see Table C.6 in the appendix).

Scout: The scout task shows a trend that is similar to the score task. Interestingly, we observe a lower experience level and in-game score, but agents descend, on average, similarly deep into the dungeon (e.g. level 5.5 for monk). This is sensible, since a policy that avoids to fight monsters, thereby lowering the chances of premature death, will not increase the in-game score as fast or level up the character as

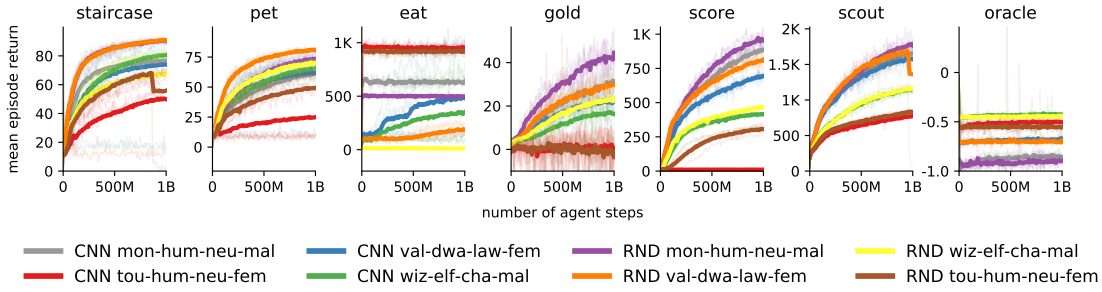


Figure 5.5: Mean return of the last 100 episodes averaged over five runs.

quickly, thus keeping the difficulty of spawned monsters low. We note that delaying to level up in order to avoid encountering stronger enemies early in the game is a known strategy human players adopt in NetHack [e.g. 121, "Why do I keep dying?" entry, January 2019 version].

Oracle: None of our agents find the Oracle (except for one lucky valkyrie episode). Locating the Oracle is a difficult exploration task. Even if the agent learns to make its way down the dungeon levels, it needs to search many, potentially branching, levels of the dungeon. Thus, we believe this task serves as a challenging benchmark for exploration methods in procedurally generated environments in the short term. Long term, many tasks harder than this (e.g., reaching *Minetown*, *Mines' End*, *Medusa's Island*, *The Castle*, *Vlad's Tower*, *Moloch's Sanctum* etc.) can be easily defined in `NLE` with very few lines of code.

5.2.1 Generalization Analysis

Akin to [36], we evaluate agents trained on a limited set of seeds while still testing on 100 held-out seeds. We find that test performance increases monotonically with the size of the set of seeds that the agent is trained on. Figure 5.4 shows this effect for the score and staircase tasks. Training only on a limited number of seeds leads to high training performance, but poor generalization. The gap between training and test performance becomes narrow when training with at least 1000 seeds, indicating that at that point agents are exposed to sufficient variation during training to make memorization infeasible. We also investigate how model capacity affects performance by comparing agents with five different hidden sizes for the final layer (of the architecture described in Section 5.1.5). Figure C.2 in the appendix shows that increasing the model capacity improves results on the score but not on the staircase task, indicating that it is an important hyperparameter to consider, as also noted by [36].

5.2.2 Qualitative Analysis

We analyse the cause for death of our agents during training and present results in Figure C.4 in the appendix. We notice that starvation and traps become a less prominent cause of death over time, most likely because our agents, when starting to learn to descend dungeon levels and fight monsters, are more likely to die in combat before they starve or get killed by a trap. In the score and scout tasks, our agents quickly learn to avoid eating rotten corpses, but food poisoning becomes again prominent towards the end of training.

We can see that gnome lords G, gnome kings G, chameleons :, and even mind flayers h become a more prominent cause of death over time, which can be explained with our agents leveling up and descending deeper into the dungeon. Chameleons are a particularly interesting entity in NetHack as they regularly change their form to a random animal or monster, thereby adversarially confusing our agent with rarely seen symbols for which it has not yet learned a meaningful representation (similar to unknown words in natural language processing). We release a set of high-score recordings of our agents (see Appendix C.10 on how to view them via a browser or terminal).

5.3 Related Work

Progress in RL has historically been achieved both by algorithmic innovations as well as development of novel environments to train and evaluate agents. Below, we review recent RL environments and delineate their strengths and weaknesses as testbeds for current methods and future research.

Recent Game-Based Environments: Retro video games have been a major catalyst for Deep RL research. ALE [15] provides a unified interface to Atari 2600 games, which enables testing of RL algorithms on high-dimensional visual observations quickly and cheaply, resulting in numerous Deep RL publications over the years [7]. The *Gym Retro* environment [122] expands the list of classic games, but focuses on evaluating visual generalization and transfer learning on a single game, *Sonic The Hedgehog*.

Both *StarCraft: BroodWar* and *StarCraft II* have been successfully employed as RL environments [172, 191] for research on, for example, planning [120], multi-agent systems [55, 150], imitation learning [55], and model-free reinforcement learning [190]. However, the complexity of these games creates a high entry barrier both in terms

of computational resources required as well as intricate baseline models that require a high degree of domain knowledge to be extended.

3D games have proven to be useful testbeds for tasks such as navigation and embodied reasoning. *Vizdoom* [90] modifies the classic first-person shooter game *Doom* to construct an API for visual control; *DeepMind Lab* [11] presents a game engine based on *Quake III Arena* to allow for the creation of tasks based on the dynamics of the original game; *Project Malmo* [81], MineRL [64] and *CraftAssist* [77] provide visual and symbolic interfaces to the popular *Minecraft* game. While Minecraft is also procedurally generated and has complex environment dynamics that an agent needs to learn about, it is much more computationally demanding than NetHack (see Table C.4 in the appendix). As a consequence, the focus has been on learning from demonstrations [64].

More recent work has produced game-like environments with procedurally generated elements, such as the *Procgen Benchmark* [36], *MazeExplorer* [65], and the *Obstacle Tower* environment [84]. However, we argue that, compared to NetHack or Minecraft, these environments do not provide the depth likely necessary to serve as long-term RL testbeds due to limited number of entities and environment interactions that agents have to learn to master. In contrast, NetHack agents have to acquire knowledge about complex environment dynamics of hundreds of entities (dungeon features, items, monsters etc.) to do well in a game that humans often take years of practice to solve.

In conclusion, none of the current benchmarks combine a fast simulator with a procedurally generated environment, a hard exploration problem, a wide variety of complex environment dynamics, and numerous types of static and interactive entities. The unique combination of challenges present in NetHack makes `NLE` well-suited for driving research towards more general and robust RL algorithms.

Roguelikes as Reinforcement Learning Testbeds: We are not the first to argue for roguelike games to be used as testbeds for RL. Asperti et al. [8] present an interface to Rogue, the very first roguelike game and one of the simplest roguelikes in terms of game dynamics and difficulty. They show that policies trained with model-free RL algorithms can successfully learn rudimentary navigation. Similarly, Kanagawa and Kaneko [89] present an environment inspired by Rogue that provides a parameterizable generation of Rogue levels. Like us, Dannenhauer et al. [42] argue that roguelike games could be a useful RL testbed. They discuss the roguelike game *Dungeon Crawl Stone Soup*, but their position paper provides neither an RL environment nor experiments to validate their claims.

Most similar to our work is *gym_nethack* [27, 28], which offers a Gym environment based on NetHack 3.6.0. We commend the authors for introducing NetHack as an RL environment, and to the best of our knowledge they were the first to suggest the idea. However, there are several design choices that limit the impact and longevity of their version as a research testbed. First, they heavily modified NetHack to enable agent interaction. In the process, *gym_nethack* disables various crucial game mechanics to simplify the game, its environment dynamics, and the resulting optimal policies. This includes removing obstacles like boulders, traps, and locked doors as well as all item identification mechanics, making items much easier to employ and the overall environment much closer to its simpler predecessor, Rogue. Additionally, these modifications tie the environment to a particular version of the game. This is not ideal as (i) players tend to use new versions of the game as they are released, hence, publicly available human data becomes progressively incompatible, thereby limiting the amount of data that can be used for learning from demonstrations; (ii) older versions of NetHack tend to include well-documented exploits which may be discovered by agents (see Appendix C.9 for exploits used in programmatic bots). In contrast, `NLE` is designed to make the interaction with NetHack as close as possible to the one experienced by humans playing the full game. `NLE` is the only environment exposing the entire game in all its complexity, allowing for larger-scale experimentation to push the boundaries of RL research.

5.4 Conclusion and Future Work

The `NetHack Learning Environment` is a fast, complex, procedurally generated environment for advancing research in RL. We demonstrate that current state-of-the-art model-free RL serves as a sensible baseline, and we provide an in-depth analysis of learned agent behaviors.

NetHack provides interesting challenges for exploration methods given the extremely large number of possible states and wide variety of environment dynamics to discover. Previously proposed formulations of intrinsic motivation based on seeking novelty [14, 129, 24] or maximizing surprise [134, 23, 139] are likely insufficient to make progress on NetHack given that an agent will constantly find itself in novel states or observe unexpected environment dynamics. NetHack poses further challenges since, in order to win, an agent needs to acquire a wide range of skills such as collecting resources, fighting monsters, eating, manipulating objects, casting spells, or taking care of their pet, to name just a few. The multilevel

dependencies present in NetHack could inspire progress in hierarchical RL and long-term planning [44, 86, 132, 189]. Transfer to unseen game characters, environment dynamics, or level layouts can be evaluated [182]. Furthermore, its richness and constant challenge make NetHack an interesting benchmark for lifelong learning [107, 131, 147, 112]. In addition, the extensive documentation about NetHack can enable research on using prior (natural language) knowledge for learning, which could lead to improvements in generalization and sample efficiency [21, 110, 209, 79]. Lastly, NetHack can also drive research on learning from demonstrations [1, 5] since a large collection of replay data is available. In sum, we argue that the `NetHack Learning Environment` strikes an excellent balance between complexity and speed while encompassing a variety of challenges for the research community.

For future versions of the environment, we plan to support NetHack 3.7 once it is released, as it will further increase the variability of observations via *Themed Rooms*. This version will also introduce scripting in the Lua language, which we will leverage to enable users to create their custom sandbox tasks, directly tapping into NetHack and its rich universe of entities and their complex interactions to define custom RL tasks.

5.5 Broader Impact

To bridge the gap between the constrained world of video and board games, and the open and unpredictable real world, there is a need for environments and tasks which challenge the limits of current Reinforcement Learning (RL) approaches. Some excellent challenges have been put forth over the years, demanding increases in the complexity of policies needed to solve a problem or scale needed to deal with increasingly photorealistic, complex environments. In contrast, our work seeks to be extremely fast to run while still testing the generalization and exploration abilities of agents in an environment which is rich, procedurally generated, and in which reward is sparse. The impact of solving these problems with minimal environment-specific heuristics lies in the development of RL algorithms which produce sample efficient, robust, and general policies capable of more readily dealing with the uncertain and changing dynamics of "real world" environments. We do not solve these problems here, but rather provide the challenge and the testbed against such improvements can be produced and evaluated.

Auxiliary to this, and in line with growing concerns that progress in Deep RL is more the result of industrial labs having privileged access to the resources required to run environments and agents on a massive scale, the environment presented

here is computationally cheap to run and to collect data in. This democratizes access for researchers in more resource-constrained labs, while not sacrificing the difficulty and richness of the environment. We hope that as a result of this, and of the more general need to develop sample-efficient agents with fewer data, the environmental impact of research using our environment will be reduced compared to more visually sophisticated ones.

6
Conclusion

In this dissertation we investigated a variety of sequential decision making problems and their solutions, focusing on topics ranging from different kinds of multi-agent learning to straight-up policy optimisation. The one common link to all this work is the willlingness to benchmark state of the art algorithms on what most RL researchers would consider to be *hard* environments.

To start with, we discovered that Multi-Agent RL has fundamental issues that need to be resolved when attempting to transition classic algorithms to the world of Deep Learning. We did so by conjuring up a complex multi-agent problem based on the hardest environment we could find, *StarCraft unit micromanagement* and by relentlessly trying to make agents learn and work on its various variants. We learnt that value-based DRL that employs experience replay buffers is subject to theoretically show-stopping non-stationarity, which can be alleviated by doing importance sampling on the experiences, and augmenting them with knowledge about when they were produced. We developed actor-critic methods that closely follow the old-school idea of *centralised learning and decentralised execution*, providing a partial solution to the problem of multi-agent credit assignment in a cooperative setting. We then took all the knowledge we learnt during this process, and created a benchmark to let the community contribute to this problem. At the time of writing this dissertation, these papers have accrued over one thousand citations, countless of direct follow-up work on multi-agent RL (most of which benchmarked on SMAC!), and have pushed the community to new directions. I am incredibly proud of this work.

Armed with hope and optimism, we took the micromanagement problem, adapted it to focus on a single unit, and we set out to figure out how to push the state of the art in differentiable planning models by reformulating VIN and producing VProp and MVProp. These models have struggled to remain relevant in the face of a community focused on creating the next best high-cost, academia-unfriendly, forest-burning policy learning system, but they lay down the foundations for what is next to come after we all collectively look in front of us and notice the sheer madness lies ahead. I highly suspect these low-level differentiable planners will become more and more common as end-to-end robotics picks up steam and architectures struggle to scale beyond a few skill spaces.

And, finally, using the gained experience gathered with SMAC and the rest of the StarCraft work, we searched and found a new environment that could provide the next exciting target for the community and ourselves: NetHack. I strongly believe the NetHack Learning Environment to be a great (almost as-good-as-it-gets right now) intersection between an extremely familiar problem – the classic gridworld – and wildly challenging learning and planning problem to solve, the kind that

makes researchers want to get out of bed and get their tensor accelerators firing up at full steam. I very much believe that NetHack makes the first of the next generation of RL grand-challenges that will help us develop more interpretable, structured, hierarchical agents that can deal with lifelong learning assumptions in very harsh adversarial settings. There are however all problems that still remain absolutely open to further investigation:

- Our multi-agent settings were both fairly unstructured and relatively abstract compared to real-life interactive settings. With the community making first steps to figure out ways of making inter-agent communication work, there exist huge opportunities ahead in terms of what a truly *human-robot-interaction-ready* multi-agent RL framework should look like. There exist nascent steps to create benchmarks and propose early frameworks to achieve basic real-life properties of the multi-agent problem that are extremely interesting and exciting, but nothing has managed to raise above the pack yet in a significant manner. Not to mention that the way we train our methods don't really scale past a handful of agents, so we will need to find ways to deal with this issue, or incorporate more sophisticated models of other agents.

- Our work on Value Propagation Networks strongly calls for a hierarchical controller to just use them and finally solve basic skill learning in robotic settings. There have been attempts to do this in a limited manner [34, 30, 201], but we are only at the very beginning of this process. Exciting real-life robotic challenges are being developed as we write this dissertation, and they are only waiting to be tackled in a general and effective way.

- A mental point that I made early in my doctoral life but that I properly ended up appreciating only in the later phases is that that Reinforcement Learning has a culture that makes it particularly challenging to benchmark and evalute algorithms with fair methodology. The field is moving fast and loose, with thousands of papers being published every year, the majority of which are not clearly comparable with each other, and most certainly not reproducible in any reasonable manner [72, 60]. Community efforts in the NLP and computer vision communities have managed to reign their own chaos, and are beginning to have well structured qualitative and quantitative benchmarking. This is something that the RL community still needs to focus on, and it is imperative that we do so as soon as possible if we want to avoid wasting thousands of people-years on research that neither matters nor is

at all scientific. Creating and curating useful benchmarks like NetHack is a great way of getting started, but we should also better understand the dynamics of RL research and directly improve it through meta-research efforts. Fortunately some initial calls for action are beginning to appear, e.g. [2, 136, 59], and the community is gradually becoming more aware of the issue. Time will tell.

Appendices

A

Deep Multi-Agent Reinforcement Learning

A.1 The StarCraft Multi-Agent Challenge

A.1.1 Experimental Setup

A.1.1.1 Architecture and Training

The architecture of all agent networks is a DRQN[69] with a recurrent layer comprised of a GRU with a 64-dimensional hidden state, with a fully-connected layer before and after. Exploration is performed during training using independent ϵ-greedy action selection, where each agent a performs ϵ-greedy action selection over its own Q_a. Throughout the training, we anneal ϵ linearly from 1.0 to 0.05 over $50k$ time steps and keep it constant for the rest of the learning. We set $\gamma = 0.99$ for all experiments. The replay buffer contains the most recent 5000 episodes. We sample batches of 32 episodes uniformly from the replay buffer, and train on fully unrolled episodes, performing a single gradient descent step after every episode. **Note:** This differs from the earlier an earlier beta release of SMAC, which trained once after every 8 episodes. The target networks are updated after every 200 training episodes.

To speed up the learning, we share the parameters of the agent networks across all agents. Because of this, a one-hot encoding of the `agent_id` is concatenated onto each agent's observations. All neural networks are trained using RMSprop[1] with learning rate 5×10^{-4}.

The mixing network consists of a single hidden layer of 32 units, utilising an ELU non-linearity. The hypernetworks consist of a feedforward network with a

[1]We set $\alpha = 0.99$ and do not use weight decay or momentum.

single hidden layer of 64 units with a ReLU non-linearity[2]. The output of the hypernetwork is passed through an absolute function (to acheive non-negativity) and then resized into a matrix of appropriate size.

The architecture of the COMA critic is a feedforward fully-connected neural network with the first 2 layers having 128 units, followed by a final layer of $|U|$ units. We set $\lambda = 0.8$. We utilise the same ϵ-floor scheme as in [54] for the agents' policies, linearlly annealing ϵ from 0.5 to 0.01 over 100k timesteps. For COMA we roll-out 8 episodes and train on those episodes. The critic is first updated, performing a gradient descent step for each timestep in the episode, starting with the final timestep. Then the agent policies are updated by a single gradient descent step on the data from all 8 episodes.

The architecture of the centralised Q for QTran is similar to the one used in [162]. The agent's hidden states (64 units) are concatenated with their chosen action ($|U|$ units) and passed through a feedforward network with a single hidden layer and a ReLU non-linearity to produce an agent-action embedding ($64 + |U|$ units). The network is shared across all agents. The embeddings are summed across all agents. The concatentation of the state and the sum of the embeddings is then passed into the Q network. The Q network consists of 2 hidden layers with ReLU non-linearities with 64 units each. The V network takes the state as input and consists of 2 hidden layers with ReLU non-linearities with 64 units each. We set $\lambda_{opt} = 1$ and $\lambda_{nopt\_min} = 0.1$.

We also compared a COMA style architecture in which the input to Q is the state and the joint-actions encoded as one-hot vectors. For both architectural variants we also tested having 3 hidden layers. For both network sizes and architectural variants we performed a hyperparameter search over $\lambda_{opt} = 1$ and $\lambda_{nopt\_min} \in \{0.1, 1, 10\}$ on all 3 of the maps we tested QTran on and picked the best performer out of all configs.

A.1.1.2 Reward and Observation

All experiments use the default shaped rewards throughout all scenarios. At each timestep, agents receive positive rewards, equal to the hit-point damage dealt, and bonuses of 10 and 200 points for killing each enemy unit and winning the scenario, respectively. The rewards are scaled so that the maximum cumulative reward achievable in each scenario is around 20.

The agent observations used in the experiments include all features from Section 3.4.2, except for the he last actions of the allied units (within the sight range), terrain height and walkability.

[2]This differs from the architecture used in [140] and in an earlier beta release of SMAC, in which a single linear layer was used for the hypernetworks.

A.1.2 Table of Results

Table A.1 shows the final median performance (maximum median across the testing intervals within the last $250k$ of training) of the algorithms tested. The mean test win %, across 1000 episodes, for the heuristic-based ai is also shown.

Table A.1: The Test Win Rate % of IQL, COMA, VDN, QMIX and the heuristic-based algorithm.

	IQL	COMA	VDN	QMIX	Heuristic
2s_vs_1sc	100	98	100	100	0
2s3z	75	43	97	99	90
3s5z	10	1	84	97	42
1c3s5z	21	31	91	97	81
10m_vs_11m	34	7	97	97	12
2c_vs_64zg	7	0	21	58	0
bane_vs_bane	99	64	94	85	43
5m_vs_6m	49	1	70	70	0
3s_vs_5z	45	0	91	87	0
3s5z_vs_3s6z	0	0	2	2	0
6h_vs_8z	0	0	0	3	0
27m_vs_30m	0	0	0	49	0
MMM2	0	0	1	69	0
corridor	0	0	0	1	0

A.1.3 Evaluation Methodology

We propose the following methodology for evaluating MARL methods using SMAC.

To ensure the fairness of the challenge and comparability of results, performances should be evaluated under standardised conditions. One should not undertake any changes to the environment used for evaluating the policies. This includes the observation and state spaces, action space, the game mechanics, and settings of the environment (e.g., frame-skipping rate). One should not modify the StarCraft II map files in any way or change the difficulty of the game AI. Episode limits of each scenario should also remain unchanged.

SMAC restricts the execution of the trained models to be decentralised, i.e., during testing each agent must base its policy solely on its own action-observation

history and cannot use the global state or the observations of other agents. It is, however, acceptable to train the decentralised policies in centralised fashion. Specifically, agents can exchange individual observations, model parameters and gradients during training as well as make use of the global state.

A.1.3.1 Evaluation Metrics

Our main evaluation metric is the mean win percentage of evaluation episodes as a function of environment steps observed, over the course of training. Such progress can be estimated by periodically running a fixed number of evaluation episodes (in practice, 32) with any exploratory behaviours disabled. Each experiment is repeated using a number of independent training runs and the resulting plots include the median performance as well as the 25-75% percentiles. We use five independent runs for this purpose in order to strike a balance between statistical significance and the computational requirements. We recommend using the median instead of the mean in order to avoid the effect of any outliers. We report the number of independent runs, as well as environment steps used in training. Each independent run takes between 8 to 16 hours, depending on the exact scenario, using Nvidia Geforce GTX 1080 Ti graphics cards.

It can prove helpful to other researchers to include the computational resources used, and the wall clock time for running each experiment. SMAC provides functionality for saving StarCraft II replays, which can be viewed using a freely available client. The resulting videos can be used to comment on interesting behaviours observed.

A.1.3.2 Scenarios

Perhaps the simplest scenarios are **symmetric** battle scenarios, where the two armies are composed of the same units. Such challenges are particularly interesting when some of the units are extremely effective against others (this is known as *countering*), for example, by dealing bonus damage to a particular armour type. In such a setting, allied agents must deduce this property of the game and design an intelligent strategy to protect teammates vulnerable to certain enemy attacks.

SMAC also includes more challenging scenarios, for example, in which the enemy army outnumbers the allied army by one or more units. In such **asymmetric** scenarios it is essential to consider the health of enemy units in order to effectively target the desired opponent.

Lastly, SMAC offers a set of interesting **micro-trick** challenges that require a higher-level of cooperation and a specific micro trick to defeat the enemy. An

example of such scenario is the `corridor` scenario (Figure 3.8c). Here, six friendly Zealots face 24 enemy Zerglings, which requires agents to make effective use of the terrain features. Specifically, agents should collectively wall off the choke point (the narrow region of the map) to block enemy attacks from different directions. The `3s_vs_5z` scenario features three allied Stalkers against five enemy Zealots (Figure 3.8b). Since Zealots counter Stalkers, the only winning strategy for the allied units is to kite the enemy around the map and kill them one after another. Some of the micro-trick challenges are inspired by *StarCraft Master* challenge missions released by Blizzard [185].

A.1.3.3 Environment Setting

SMAC makes use of the StarCraft II Learning Environment (*SC2LE*) [192] to communicate with the StarCraft II engine. SC2LE provides full control of the game by making it possible to send commands and receive observations from the game. However, SMAC is conceptually different from the RL environment of SC2LE. The goal of SC2LE is to learn to play the full game of StarCraft II. This is a competitive task where a centralised RL agent receives RGB pixels as input and performs both macro and micro with the player-level control similar to human players. SMAC, on the other hand, represents a set of cooperative multi-agent micro challenges where each learning agent controls a single military unit.

SMAC uses the *raw API* of SC2LE. Raw API observations do not have any graphical component and include information about the units on the map such as health, location coordinates, etc. The raw API also allows sending action commands to individual units using their unit IDs. This setting differs from how humans play the actual game, but is convenient for designing decentralised multi-agent learning tasks.

Furthermore, to encourage agents to explore interesting micro strategies themselves, we limit the influence of the StarCraft AI on our agents. In the game of StarCraft II, whenever an idle unit is under attack, it automatically starts a reply attack towards the attacking enemy units without being explicitly ordered. We disable such automatic replies towards the enemy attacks or enemy units that are located closely by creating new units that are the exact copies of existing ones with two attributes modified: *Combat: Default Acquire Level* is set to *Passive* (default *Offensive*) and *Behaviour: Response* is set to *No Response* (default *Acquire*). These fields are only modified for allied units; enemy units are unchanged.

The sight and shooting range values might differ from the built-in *sight* or *range* attribute of some StarCraft II units. Our goal is not to master the original full StarCraft II game, but rather to benchmark MARL methods for decentralised control.

The game AI is set to level 7, *very difficult*. Our experiments, however, suggest that this setting does significantly impact the unit micromanagement of the built-in heuristics.

B

Value Propagation Networks

B.1 More general graph structures

VIN and VProp are, in their most general formulation, applicable to any graph-structured input. They ultimately belong to the general class of graph convolutional neural networks, and several variations of the value iteration modules specifically tailored to non-regular graph structures have been proposed (see e.g., Niu et al. [123]). The three equations for VProp (Sections 4.3.2, 4.3.1) are applicable to any graph structure, as long as there is a one-to-one mapping between nodes in the neighborhood of the current position and actions (which is usually the case in navigation problems). Our work focuses on the simplest possible parametrization that is relevant in many navigation and pathfinding scenarios, while for more general graph structures, even assuming a deterministic model, the term $p_{i,j}v^{(k-1)}_{i',j'}$ should be extended into $p_{i,j,i',j'}v^{(k-1)}_{i',j'}$ to account for the edge weight between i, j and i', j'. For regular graph structures such as 2D grids, this can be included in the embedding function Φ, which outputs not just one parameter for each position but as many parameters as the neighborhood size. Other possibilities include using an attention mechanism for graph-convolutional neural networks in place of p (see e.g., Tamar et al. [176, section 4.4]). In such case, our method differs from the original VIN purely by the parametrization of the reward and the focus on the deterministic models, which we believe are relevant in navigation problems.

B.2 Agent setup

All the models and agent code was implemented in PyTorch [133], and will be made available upon acceptance together with the environments. Most of the agents tested shared learning hyperparameters fitted to the VIN baseline to make comparison as fair as possible, and were validated using more than 5 random seeds across all our experiments. The second term in the ϕ^{t_1} update is supposed to play the role of TRPO-like regularization as implemented in [193], where they use probabilities of an average model instead of the previous probabilities. We implemented an n-step memory replay buffer, observing that keeping only the last 50000 transitions (to avoid trying to fit predictions of a bad model) worked well on our tasks. In all our experiments we used RMSProp rather than plain SGD, with relative weights $\lambda = \eta = 100.0\eta'$. We also used a learning rate to 0.001 and mini-batch size of 128, with learning updates set at a frequency of 32 steps. We tested reasonable ranges for all these hyperparameters, but observed no relative significant changes when cross-validated over multiple seeds for most of them.

B.3 MazeBase setup

The agents were tasked to navigate the maze as fast as possible, as total cost increased with time since *noop* actions were not allowed. Episodes terminated whenever the agent would take any illegal action such as hitting a wall, or when the maximum number of steps (set to roughly three times the length of the shortest path) would be reached. All entites were set with discrete collision boundaries corresponding to the featurised observation. The environments were also constrained so that only one entity could be present in any given cell at time t. Unless specified otherwise, attempting to walk into walls would yield a reward of -1, any valid movement would provide a reward of $-0.01 \times f(a_t, s_t)$, where $f(a, s)$ is the cost of moving in the direction specified by action a in state s, and reaching the goal would give a positive reward of 1 to the agent. In our experiments we define f as the L2 distance between the agent position at state s_t, and the one at s_{t+1}, to adjust for the real cost of moving diagonally.

For all static experiments, the ratio of un-walkable blocks over total space was fixed to 30%, and the blocks were sampled uniformly within the space, unless specified otherwise. This setting provided environments of decent difficulty, with chunky obstacles as well as harder, more narrow paths. Dynamic environments used different ratios of blocks (and other entities) vs walkable surface: *avalanche*

maps	VIN	VProp	MVProp	VIN	VProp	MVProp
	win rate			distance to optimal path		
v16x16	63.6% ± 13.2%	94.4% ± 5.6%	100%	0.2 ± 0.2	0.2 ± 0.2	0.0 ± .0
32×32	15.6% ± 5.3%	68.8% ± 27.2%	100%	0.8 ± 0.3	0.4 ± 0.3	0.0 ± .0
64×64	4.0% ± 4.1%	53.2% ± 31.8%	100%	1.5 ± 0.4	0.5 ± 0.4	0.0 ± .0

Table B.1: Average performance at the end of training of all tested models on the static grid-worlds with 90% confidence value, across 5 different training runs (with random seeding). v16x16 correspond to the maps sampled from VIN's 16x16 grid test dataset, while the rest of the maps are sampled uniformly from our generator using the same parameters employed at training time. The distance to the optimal path is averaged only for successful episodes.

environments filled between 20% and 30% of the surface with "falling" entities, *enemies only* spawned 10% (rounded to the bigger integer) of the surface with adversarial agents running A*, while *mixed* environments employed the same amount of adversarial agents with the addition of static and stochastic entities (with and ϵ-greedy stochastic policy) making up for 10% of the surface area, for a total of roughly 20% occupied blocks.

Our VIN models were based off the architecture employed by Tamar et al. [176] in their grid-world experiments, which we used to also build VProp and MVProp models. The only significant difference between the models (beyond the number of maps used in the recurrency) consisted in VProp and MVProp using 8 input filters and unpadded convolutions. Note that we tested our VIN models using the same setup and saw no significant difference in these tests either.

B.4 StarCraft setup

As TorchCraft by default runs at a very high framerate, we had to set the amount of skipped frames to 15, to allow us to be able to see relevant changes in the environment after each step; this is roughly double compared to other work done on the same platform [188, 55]. The rest of the environment parameters were kept to the default values. All entities were spawned in a similar fashion to the grid-world experiments, with an additional minimal constraint based on each entity's pixel size. At training time we employed a fixed max-distance curriculum to gradually increase the distance between spawned goal and agent, however we also prevented enemies from spawning during the first 500 episodes so as to allow the stochastic policy to quickly condition on the goal. Without this particular change in the curriculum

setting we found learning to be generally more unstable, since the stochastic policy would need to naturally luck out. This could have been fixed by utilising a better exploration strategy, but we considered it outside the scope of this work.

We used a fixed 8x downsampling operator to reduce the size of the raw observation of 480x360 pixels to 60x45. Based on the experiment, this downsampled observation was then either fully featurised in a grid-world fashion, or transformed into greyscale (as it is typically done in other deep reinforcment learning experimental settings). We increased the capacity of the models by adding 2 additional convolutional layers with 32 filters and 7x7 and 5x5 kernels, and max-pooling layers with stride and extent equal to 2 to further reduce the dimensionality before the recurrent step.

C

The NetHack Learning Environment

C.1 Further Details on NetHack

Character options The player may choose (or pick randomly) the character from thirteen roles (archaeologist, barbarian, cave(wo)man, healer, knight, priest(ess), ranger, rogue, samurai, tourist, valkyrie, and wizard), five races (human, elf, dwarf, gnome, and orc), three moral alignments (neutral, lawful, chaotic), and two genders (male or female). Each choice determines some of the character's features, as well as how the character interacts with other entities (e.g., some species of monsters may not be hostile depending on the character race; priests of a particular deity may only help religiously aligned characters).

The hero's interaction with several game entities involves pre-defined stochastic dynamics (usually defined by virtual dice tosses), and the game is designed to heavily punish careless exploration policies.[1] This makes NetHack an ideal environment for evaluating exploration methods such as curiosity-driven learning [134, 23] or safe reinforcement learning [58].

Learning and planning in NetHack involves dealing with partial observability. The game, by default, employs *Fog of War* to hide information based on a simple 2D light model (see for example the difference between white . and gray . room tiles in Figure 5.1 or Figure C.6), requiring the player not only to discover the topology of the level (including searching for hidden doors and passages), but to also condition their policy on a world that might change, e.g., due to monsters spawning and interacting outside of the visible range.

[1] Occasionally dying because of simple, avoidable mistakes is so common in the game that the online community has defined an acronym for it: *Yet Another Stupid Death* (YASD).

On top of the standard ASCII interface, NetHack supports many official and unofficial graphical user interfaces. Figure C.1 shows a screenshot of Lu Wang's BrowserHack[2] as an example.

Figure C.1: Screenshot of BrowserHack showing NetHack with a graphical user interface.

Conducts While winning NetHack by retrieving and ascending with the Amulet of Yendor is already immensely challenging, experienced NetHack players like to challenge themselves even more by imposing additional restrictions on their play. The game tracks some of these challenges with the `#conduct` command [142]. These official challenges include eating only vegan or vegetarian food, or not eating at all, or playing the game in "pacifist" mode without killing a single monster. While very experienced players often try to adhere to several challenges at once, even moderately experienced players often limit their use of certain polymorph spells (e.g., "polypiling"—changing the form of several objects at once in the hope of getting better ones) or they try to beat the game while *minimizing* the in-game

[2]Playable online at https://coolwanglu.github.io/BrowserHack/

score. We believe this established set of conducts will supply the RL community with a steady stream of extended challenges once the standard `NetHack Learning Environment` is solved by future methods.

C.2 Observation Space

The Gym environment is implemented by wrapping a more low-level `NetHack` Python object into a Python class responsible for the featurization, reward schedule and end-of-episode dynamics. While the low-level `NetHack` object gives access to a large number of NetHack game internals, the Gym wrapper exposes by default only a part of this data as numerical observation arrays, namely the observation tensors *glyphs*, *chars*, *colors*, *specials*, *blstats*, *message*, *inv_glyphs*, *inv_strs*, *inv_letters*, and *inv_oclasses*.

Glyphs, Chars, Colors, Specials: NetHack supports non-ASCII graphical user interfaces, dubbed window-ports (see Figure C.1 for an example). To support displaying different monsters, objects and floor types in the NetHack dungeon map as different tiles, NetHack internally defines *glyphs* as ids in the range $0, \ldots,$ `MAX_GLYPH`, where `MAX_GLYPH` $= 5991$ in our build[3]. The *glyph* observation is an integer array of shape $(21, 79)$ of these game glyph ids.[4] In NetHack's standard terminal-based user interface, these glyphs are mapped into ASCII characters of different colors which we return as the *chars*, *colors*, and *specials* observations, both all which are of shape $(21, 79)$; *chars* are ASCII bytes in the range $0, \ldots, 127$ wheras *colors* are in range $0, \ldots, 15$. For additional highlighting (e.g., flipping background and foreground colors for the hero's pet), NetHack also computes xor'ed values which we return as the *specials* tensor.

Blstats: "Bottom line statistics", a integer vector of length 25, containing the (x, y) coordinate of the hero and the following 23 character stats that typically appear in the bottom line of the ASCII interface: `strength_percentage`, `strength`, `dexterity`, `constitution`, `intelligence`, `wisdom`, `charisma`, `score`, `hitpoints`,

[3]The exact number of monsters in NetHack depends on compile-time options as well as the target operating system. For instance, the mail daemon is only available on Unix-like operating systems, where it delivers email in the form of a NetHack scroll if the system is configured to host a Unix mailbox.

[4]NetHack's set of glyph ids is not necessarily well suited for machine learning. For example, more than half of all glyph ids are of type "swallow", most of which are guaranteed not to show up in any actual game of NetHack. We provide additional tooling to determine the type of a given glyph id to process this observation further.

`max_hitpoints`, `depth`, `gold`, `energy`, `max_energy`, `armor_class`, `monster_level`, `experience_level`, `experience_points`, `time`, `hunger_state`, `carrying_capacity`, `dungeon_number`, and `level_number`.

Message: A padded byte vector of length 256 representing the current message shown to the player, normally displayed in the top area of the GUI. We support different padding strategies and alphabet sizes, but by default we choose an alphabet size of 96, where the last character is used for padding.

Inventory: In NetHack's default ASCII user interface, the hero's inventory can be opened and closed during the game. Other user interfaces display a permanent inventory at all times. `NLE` follows that strategy. The inventory observations consist of the following four arrays: *inv_glyphs*: an integer vector of length 55 of glyph ids, padded with `MAX_GLYPH`; *inv_strs*: A padded byte array of shape $(55, 80)$ describing the inventory items; *inv_letters*: A padded byte vector of length 55 with the corresponding ASCII character symbol; *inv_oclasses*: An integer vector of shape 55 with ids describing the type of inventory objects, padded with `MAXOCLASSES` $= 18$.

The low-level `NetHack` Python object has some additional methods to query and modify NetHack's game state, e.g. the current RNG seeds. We refer to the source code to describe these.[5]

C.3 Action Space

The game of NetHack uses ASCII inputs, i.e., individual keyboard presses including modifiers like Ctrl and Meta. `NLE` pre-defines 98 actions, 16 of which are compass directions and 82 of which are command actions. Table C.1 gives a list of command actions, including their ASCII value and the corresponding key binding in NetHack, while Table C.3 lists the 16 compass directions. For a detailed description of these actions, as well as other NetHack commands, we refer the reader to the NetHack guide book [142]. Not all actions are sensible for standard RL training on `NLE`. E.g., the VERSION or QUIT actions are unlikely to be useful for direct input from the agent. `NLE` defines a list of `USEFUL_ACTIONS` that includes a subset of 76 actions; however, what is useful depends on the circumstances. In addition, even though an action like SAVE is unlikely to be useful in most game situations it corresponds to the letter `S`, which may be assigned to an inventory item or some other in-game menu entry such that it does become a useful action in that context.

[5] See, e.g., the `nethack.py` as well as `pynethack.cc` files in the `NLE` repository.

By default, `NLE` will auto-apply the MORE action in situations where the game waits for input to display more messages.

Table C.1: Command actions.[6]

Name	Value	Key	Description
EXTCMD	35	`#`	perform an extended command
EXTLIST	191	`M-?`	list all extended commands
ADJUST	225	`M-a`	adjust inventory letters
ANNOTATE	193	`M-A`	name current level
APPLY	97	`a`	apply (use) a tool (pick-axe, key, lamp...)
ATTRIBUTES	24	`C-x`	show your attributes
AUTOPICKUP	64	`@`	toggle the pickup option on/off
CALL	67	`C`	call (name) something
CAST	90	`Z`	zap (cast) a spell
CHAT	227	`M-c`	talk to someone
CLOSE	99	`c`	close a door
CONDUCT	195	`M-C`	list voluntary challenges you have maintained
DIP	228	`M-d`	dip an object into something
DOWN	62	`>`	go down (e.g., a staircase)
DROP	100	`d`	drop an item
DROPTYPE	68	`D`	drop specific item types
EAT	101	`e`	eat something
ESC	27	`C-[`	escape from the current query/action
ENGRAVE	69	`E`	engrave writing on the floor
ENHANCE	229	`M-e`	advance or check weapon and spell skills
FIRE	102	`f`	fire ammunition from quiver
FIGHT	70	`F`	Prefix: force fight even if you don't see a monster
FORCE	230	`M-f`	force a lock
GLANCE	59	`;`	show what type of thing a map symbol corresponds to
HELP	63	`?`	give a help message
HISTORY	86	`V`	show long version and game history
INVENTORY	105	`i`	show your inventory
INVENTTYPE	73	`I`	inventory specific item types
INVOKE	233	`M-i`	invoke an object's special powers
JUMP	234	`M-j`	jump to another location
KICK	4	`C-d`	kick something
KNOWN	92	`\`	show what object types have been discovered
KNOWNCLASS	96	`` ` ``	show discovered types for one class of objects
LOOK	58	`:`	look at what is here
LOOT	236	`M-l`	loot a box on the floor
MONSTER	237	`M-m`	use monster's special ability
MORE	13	`C-m`	read the next message
MOVE	109	`m`	Prefix: move without picking up objects/fighting

[6]The descriptions are mostly taken from the `cmd.c` file in the NetHack source code.

MOVEFAR	77	`M`	Prefix: run without picking up objects/fighting
OFFER	239	`M-o`	offer a sacrifice to the gods
OPEN	111	`o`	open a door
OPTIONS	79	`O`	show option settings, possibly change them
OVERVIEW	15	`C-o`	show a summary of the explored dungeon
PAY	112	`p`	pay your shopping bill
PICKUP	44	`,`	pick up things at the current location
PRAY	240	`M-p`	pray to the gods for help
PREVMSG	16	`C-p`	view recent game messages
PUTON	80	`P`	put on an accessory (ring, amulet, etc)
QUAFF	113	`q`	quaff (drink) something
QUIT	241	`M-q`	exit without saving current game
QUIVER	81	`Q`	select ammunition for quiver
READ	114	`r`	read a scroll or spellbook
REDRAW	18	`C-r`	redraw screen
REMOVE	82	`R`	remove an accessory (ring, amulet, etc)
RIDE	210	`M-R`	mount or dismount a saddled steed
RUB	242	`M-r`	rub a lamp or a stone
RUSH	103	`g`	Prefix: rush until something interesting is seen
SAVE	83	`S`	save the game and exit
SEARCH	115	`s`	search for traps and secret doors
SEEALL	42	`*`	show all equipment in use
SEETRAP	94	`^`	show the type of adjacent trap
SIT	243	`M-s`	sit down
SWAP	120	`x`	swap wielded and secondary weapons
TAKEOFF	84	`T`	take off one piece of armor
TAKEOFFALL	65	`A`	remove all armor
TELEPORT	20	`C-t`	teleport around the level
THROW	116	`t`	throw something
TIP	212	`M-T`	empty a container
TRAVEL	95	`_`	travel to a specific location on the map
TURN	244	`M-t`	turn undead away
TWOWEAPON	88	`X`	toggle two-weapon combat
UNTRAP	245	`M-u`	untrap something
UP	60	`<`	go up (e.g., a staircase)
VERSION	246	`M-v`	list compile time options
VERSIONSHORT	118	`v`	show version
WAIT / SELF	46	`.`	rest one move while doing nothing / apply to self
WEAR	87	`W`	wear a piece of armor
WHATDOES	38	`&`	tell what a command does
WHATIS	47	`/`	show what type of thing a symbol corresponds to
WIELD	119	`w`	wield (put in use) a weapon
WIPE	247	`M-w`	wipe off your face
ZAP	112	`z`	zap a wand

Table C.3: Compass direction actions.

	one-step		move far	
Direction	Value	Key	Value	Key
North	107	`k`	75	`K`
East	108	`l`	76	`L`
South	106	`j`	74	`J`
West	104	`h`	72	`H`
North East	117	`u`	85	`U`
South East	110	`n`	78	`N`
South West	98	`b`	66	`B`
North West	121	`y`	89	`Y`

C.4 Environment Speed Comparison

Table C.4 shows a comparison between popular Gym environments and `NLE`. All environments were controlled with a uniformly random policy using reset on terminal states. The tests were conducted on a MacBook Pro equipped with an Intel Core i7 2.9 GHz, 16GB of RAM, MacOS Mojave, Python 3.7, Conda 4.7.12, and latest available packages as of May 2020. *ObstacleTowerEnv* was instantiated with `(retro=False, real_time=False)`. Note that this data does not necessarily reflect performance of these environments with better—or worse—policies, as each environment has computational dynamics that depend on its state. However, we expect the difference in terms of magnitude to remain mostly unchanged across these environments.

Table C.4: Comparison between `NLE` and popular environments when using their respective Python Gym interface. SPS stands for "environment steps per second". All environments but `ObstacleTowerEnv` were run via `gym` with standard settings (and headless when possible), for 60 seconds.

Environment	SPS	steps	episodes
`NLE` (score)	14.4K	868.75K	477
CartPole-v1	76.88K	4612.65K	207390
ALE (MontezumaRevengeNoFrameskip-v4)	0.90K	53.91K	611
Retro (Airstriker-Genesis)	1.31K	78.56K	52
ProcGen (procgen-coinrun-v0)	13.13K	787.98K	1283
ObstacleTowerEnv	0.06K	3.61K	6
MineRLNavigateDense-v0	0.06K	3.39K	0

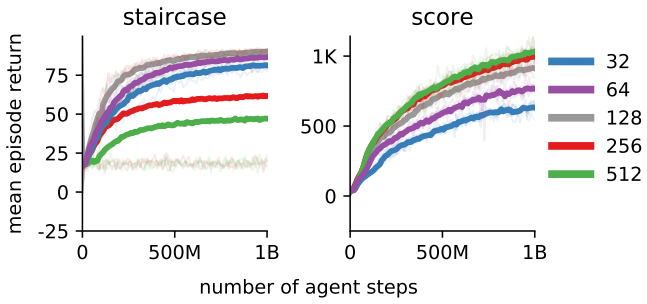


Figure C.2: Mean episode return of the last 100 episodes for models with different hidden sizes averaged over five runs.

C.5 Task Details

For all tasks described below, we add a penalty of -0.001 to the reward function if the agent's action did not advance the in-game timer, which, for example, happens when the agent tries to move against a wall or navigates menus. For all tasks, except the *Gold* task, we disable NetHack's *autopick* option [142]. Furthermore, we disable so-called *bones files* that would otherwise lead to agents occasionally discovering the remains and ghosts of previous agents, considerably increasing the variance across episodes.

Staircase The agent has to find the staircase down `>` to the next dungeon level. This task is already challenging, as there is often no direct path to the staircase. Instead, the agent has to learn to reliably open doors `+`, kick-in locked doors, search for hidden doors and passages `#`, avoid traps `^`, or move boulders `0` that obstruct a passage. The agent receives a reward of 100 once it reaches the staircase down and the the episode terminates after 1000 agent steps.

Pet Many successful strategies for NetHack rely on taking good care of the hero's pet (e.g., the little dog `d` or kitten `f` that the hero starts with). Pets are controlled by the game, but their behavior is influenced by the agent's actions. In this task, the agent only receives a positive reward of 100 when it reaches the staircase while the pet is next to the agent.

Eat To survive in NetHack, players have to make sure their character does not starve to death. There are many edible objects in the game, for example food rations `%`, tins, and monster corpses. In this task, the agent receives the increase of nutrition as determined by the in-game "Hunger" status as reward [see 121, "Nutrition" entry for details]. A steady source of nutrition are monster corpses,

but for that the agent has to learn to locate and to kill monsters while avoiding to consume rotten corpses, poisonous monster corpses such as Kobolds k or acidic monster corpses such as Acid Blobs b.

Gold Throughout the game, the player can collect gold $ to, for example, trade for useful items with shopkeepers. The agent receives the amount of gold it collects as reward. This incentivizes the agent to explore dungeon maps fully and to descend dungeon levels to discover new sources of gold. There are many advanced strategies for obtaining large amounts of gold such as finding, identifying and selling gems; stealing from or killing shopkeepers; or hunting for vaults or leprechaun halls. To make this task easier for the agent, we enable NetHack's *autopickup* option for gold.

Scout An important part of the game is exploring dungeon levels. Here, we reward the agent (+1) for uncovering previously unknown tiles in the dungeon, for example by entering a new room or following a newly discovered passage. Like the previous task, this incentivizes the agent to explore dungeon levels and to descend.

Score In this task, the agent receives the increase of the in-game score between two time steps as reward. The in-game score is governed by a complex calculation, but in early stages of the game it is dominated by killing monsters and the number of dungeon levels that the agent descends [see 121, "Score" entry for details].

Oracle While levels are procedurally generated, there are a number of landmarks that appear in every game of NetHack. One such landmark is the Oracle 0, which is randomly placed between levels five and nine of the dungeon. Reliably finding the Oracle is difficult, as it requires the agent to go down multiple staircases and often to exhaustively explore each level. In this task, the agent receives a reward of 1000 if it manages to reach the Oracle.

C.6 Baseline CNN Details

As embedding dimension of the glyphs we use 32 and for the hidden dimension for the observation $\mathbf{o}_t$ and the output of the LSTM $\mathbf{h}_t$, we use 128. For encoding the full map of glyphs as well as the 9×9 crop, we use a 5-layer ConvNet architecture with filter size 3×3, padding 1 and stride 1. The input channel of the first layer of the ConvNet is the embedding size of the glyphs (32). Subsequent layers have an input and output channel dimension of 16. We employ a gradient norm clipping of 40 and clip rewards using $r_c = \tanh(r/100)$. We use RMSProp with a learning rate of 0.0002 without momentum and with $\varepsilon_{\text{RMSProp}} = 0.000001$. Our entropy cost is set to 0.0001.

C.7 Random Network Distillation Details

For RND hyperparameters we mostly follow the recommendations by the authors [24]:

- we initialize the weights according to the original paper, using an orthogonal distribution with a gain of $\sqrt{2}$
- we use a two-headed value function rather than merely summing the intrinsic and extrinsic reward
- we use a discounting factor of 0.999 for the extrinsic reward and 0.99 for the intrinsic reward
- we use non-episodic intrinsic reward and episodic extrinsic reward
- we use reward normalization for the intrinsic reward, dividing it by a running estimate of its standard deviation

We modify a few of the parameters for use in our setting:

- we use exactly the same feature extraction architecture as the baseline model instead of the pixel-based convolutional feature extractor
- we do not use observation normalization, again due to the symbolic nature of our observation space
- before normalizing, we divide the intrinsic reward by ten so that it has less weight than the extrinsic reward
- we clip intrinsic rewards in the same way that we clip extrinsic rewards, i.e., using $r_c = \tanh(r/100)$, so that the intrinsic and extrinsic rewards are on a similar scale

We downscale the forward modeling loss by a factor of 0.01 to slow down the rate at which the model becomes familiar with a given state, since the intrinsic reward often collapsed quickly despite the reward normalization. We determined these settings during a set of small-scale experiments.

We also tried using subsets of the full feature set (only the embedding of the full display of glyphs, or only the embedding of the crop of glyphs around the agent) as well as the exact architecture used by the original authors, but with the pixel input replaced by a random 8-dimensional embedding of the symbolic observation space. However, we did not observe this improved results.

We tried using intrinsic reward only as the authors did in the original RND paper, but we found that agents trained in this way made no significant progress through the dungeon, even on a single fixed seed. This indicates that this form of intrinsic reward is not sufficient to make progress on NetHack. As noted in Section 5.2, the intrinsic reward did help in some tasks for some characters when combined with the extrinsic reward. Crucially, RND exploration is not sufficient for agents to learn to find the Oracle, which leaves this as a difficult challenge for future exploration techniques.

C.8 Dashboard

We release a web dashboard built with NodeJS (see Figure C.5) to visualize experiment runs and statistics for `NLE`, including replaying episodes that were recorded as `tty` files.

C.9 NetHack Bots

Since the early stages of the development of NetHack, players have tried to build bots to play and solve the game. Notable examples are *TAEB*, *BotHack*, and *Saiph* [175, 121]. These bot frameworks largely rely on search heuristics and common planning methods, without generally making use of any statistical learning methods. An exception is *SWAGGINZZZ* [4] which uses lookups, exhaustive simulation and manipulation of the random number generator.

Successful bots have made use of exploits that are no longer present in recent versions of NetHack. For example, *BotHack* employs the "pudding farming" strategy [see 121, "Pudding farming" entry] to level up and to create items for the character by spawning and killing a large number of black puddings P. This enabled the bot to become quite strong, which rendered late-game fights considerably easier. This strategy was disabled by the NetHack DevTeam with a patch that is incorporated into versions of NetHack above 3.6.0. Likewise, the random number generator manipulations employed in *SWAGGINZZZ* are no longer possible.

We believe that it is very unlikely that in the future we will see a hand-crafted bot solving NetHack in the way we defined it in Section 5.1.4. In fact, the creator of *SWAGGINZZZ* remarked that "[e]ven with RNG manipulation, writing a bot that 99% ascends NetHack is **extremely** complicated. So much stuff can go wrong, and there is no shortage of corner cases" [4].

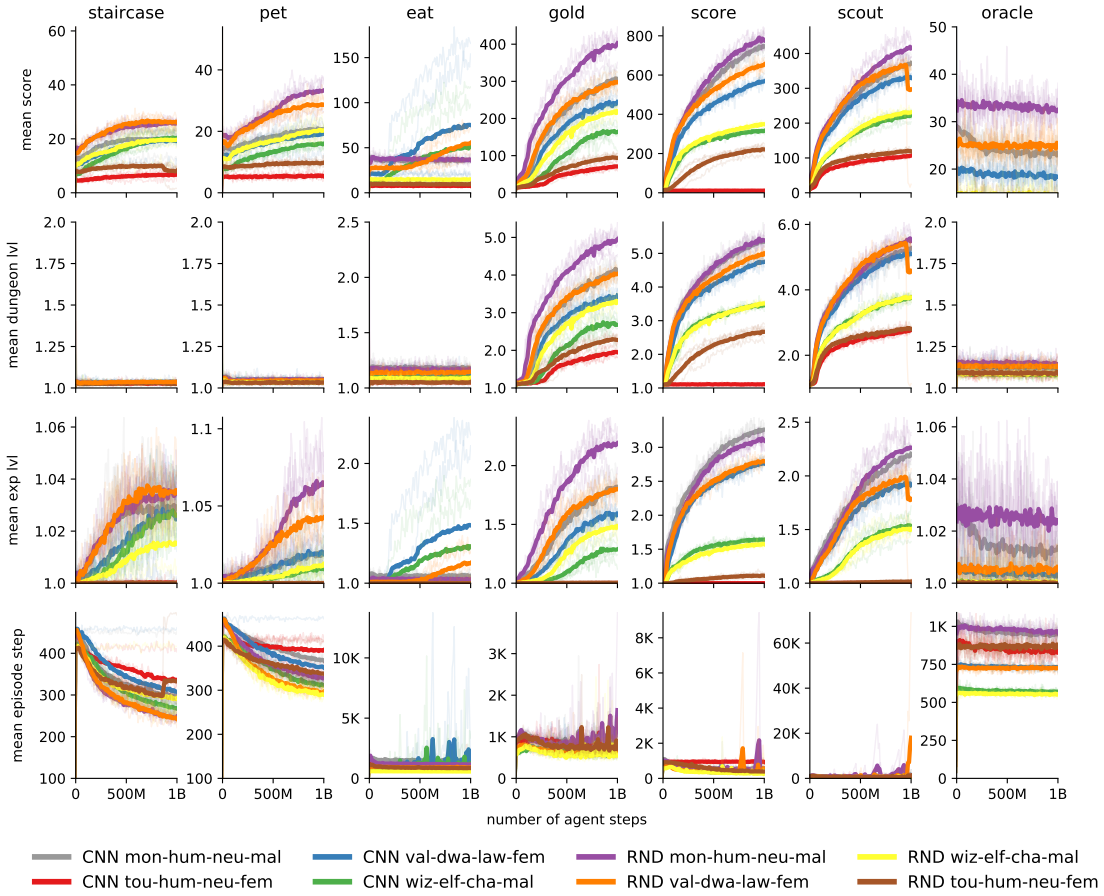


Figure C.3: Mean score, dungeon level reached, experience level achieved, and steps performed in the environment in the last 100 episodes averaged over five runs.

C.10 Viewing Agent Videos

We have uploaded some agent recordings to https://asciinema.org/~nle. These can be either watched on the Asciinema portal, or on a terminal by running `asciinema play -s 0.2` *`url`* (`asciinema` itself is available as a pip package at https://pypi.org/project/asciinema). The `-s` flag regulates the speed of the recordings, which can also be modified on the web interface by pressing `>` (faster) or `<` (slower).

Table C.5: Metrics averaged over last 1000 episodes for each task.

Task	Model	Character	Score	Time	Exp Lvl	Dungeon Lvl	Win
staircase	CNN	mon-hum-neu-mal	20	252	1.0	1.0	77.26
		tou-hum-neu-fem	6	288	1.0	1.0	50.42
		val-dwa-law-fem	19	329	1.0	1.0	74.62
		wiz-elf-cha-mal	20	253	1.0	1.0	80.42
	RND	mon-hum-neu-mal	26	199	1.0	1.0	90.84
		tou-hum-neu-fem	8	203	1.0	1.0	56.94
		val-dwa-law-fem	25	242	1.0	1.0	90.96
		wiz-elf-cha-mal	20	317	1.0	1.0	67.46
pet	CNN	mon-hum-neu-mal	20	297	1.0	1.1	62.02
		tou-hum-neu-fem	6	407	1.0	1.0	25.66
		val-dwa-law-fem	18	379	1.0	1.0	63.30
		wiz-elf-cha-mal	16	273	1.0	1.0	66.80
	RND	mon-hum-neu-mal	33	319	1.1	1.0	74.38
		tou-hum-neu-fem	10	336	1.0	1.0	49.38
		val-dwa-law-fem	28	311	1.0	1.0	81.56
		wiz-elf-cha-mal	20	278	1.0	1.0	70.48
eat	CNN	mon-hum-neu-mal	36	1254	1.1	1.2	–
		tou-hum-neu-fem	7	423	1.0	1.0	–
		val-dwa-law-fem	75	1713	1.5	1.1	–
		wiz-elf-cha-mal	50	1181	1.3	1.1	–
	RND	mon-hum-neu-mal	36	1102	1.0	1.2	–
		tou-hum-neu-fem	9	404	1.0	1.0	–
		val-dwa-law-fem	55	1421	1.2	1.1	–
		wiz-elf-cha-mal	14	808	1.0	1.1	–
gold	CNN	mon-hum-neu-mal	307	947	1.8	4.2	–
		tou-hum-neu-fem	71	788	1.0	2.0	–
		val-dwa-law-fem	245	1032	1.6	3.5	–
		wiz-elf-cha-mal	162	780	1.3	2.7	–
	RND	mon-hum-neu-mal	403	1006	2.2	5.0	–
		tou-hum-neu-fem	92	816	1.0	2.2	–
		val-dwa-law-fem	298	998	1.8	4.0	–
		wiz-elf-cha-mal	217	789	1.5	3.3	–
score	CNN	mon-hum-neu-mal	748	932	3.2	5.4	–
		tou-hum-neu-fem	11	795	1.0	1.1	–
		val-dwa-law-fem	573	908	2.8	4.8	–
		wiz-elf-cha-mal	314	615	1.6	3.5	–
	RND	mon-hum-neu-mal	780	863	3.1	5.4	–
		tou-hum-neu-fem	219	490	1.1	2.6	–
		val-dwa-law-fem	647	857	2.8	5.0	–
		wiz-elf-cha-mal	352	585	1.6	3.5	–
scout	CNN	mon-hum-neu-mal	372	838	2.2	5.3	–
		tou-hum-neu-fem	105	580	1.0	2.7	–
		val-dwa-law-fem	331	852	1.9	5.1	–
		wiz-elf-cha-mal	222	735	1.5	3.8	–
	RND	mon-hum-neu-mal	416	924	2.3	5.5	–
		tou-hum-neu-fem	119	599	1.0	2.8	–
		val-dwa-law-fem	304	1021	1.8	4.6	–
		wiz-elf-cha-mal	231	719	1.5	3.8	–
oracle	CNN	mon-hum-neu-mal	24	876	1.0	1.1	0.00
		tou-hum-neu-fem	9	674	1.0	1.1	0.00
		val-dwa-law-fem	18	1323	1.0	1.1	0.02
		wiz-elf-cha-mal	10	742	1.0	1.1	0.00
	RND	mon-hum-neu-mal	32	967	1.0	1.1	0.00
		tou-hum-neu-fem	13	811	1.0	1.1	0.00
		val-dwa-law-fem	26	1353	1.0	1.1	0.00
		wiz-elf-cha-mal	14	791	1.0	1.1	0.00

Table C.6: Top five of the last 1000 episodes in the score task.

Model	Character	Killer Name	Score	Exp Lvl	Dungeon Lvl
CNN	mon-hum-neu-mal	warg	4408	7	9
		forest centaur	4260	7	11
		hill orc	2880	6	8
		gnome lord	2848	6	9
		crocodile	2806	6	8
	tou-hum-neu-fem	jackal	200	1	3
		hobgoblin	200	1	5
		hobbit	200	1	3
		giant rat	190	1	4
		large kobold	174	1	4
	val-dwa-law-fem	gnome lord	2176	5	12
		ape	1948	6	7
		gremlin	1924	5	11
		gnome king	1916	5	11
		vampire	1864	4	10
	wiz-elf-cha-mal	dingo	1104	3	9
		giant ant	1008	3	8
		gnome mummy	988	3	8
		coyote	988	3	9
		kicking a wall	972	3	8
RND	mon-hum-neu-mal	rothe	3664	5	7
		rotted dwarf corpse	3206	5	7
		leocrotta	2771	5	11
		winter wolf cub	2724	6	9
		starvation	2718	6	6
	tou-hum-neu-fem	grid bug	1432	1	7
		sewer rat	1253	1	4
		bolt of cold	1248	1	3
		goblin	1125	1	4
		goblin	1078	1	4
	val-dwa-law-fem	bugbear	2186	6	9
		starvation	2150	5	10
		ogre	2095	5	9
		rothe	2084	6	8
		Uruk-hai called Haiaigrisai of Aruka	2036	5	6
	wiz-elf-cha-mal	cave spider	1662	2	7
		iguana	1332	2	5
		starvation	1329	1	5
		starvation	1311	1	5
		gnome lord	1298	5	9

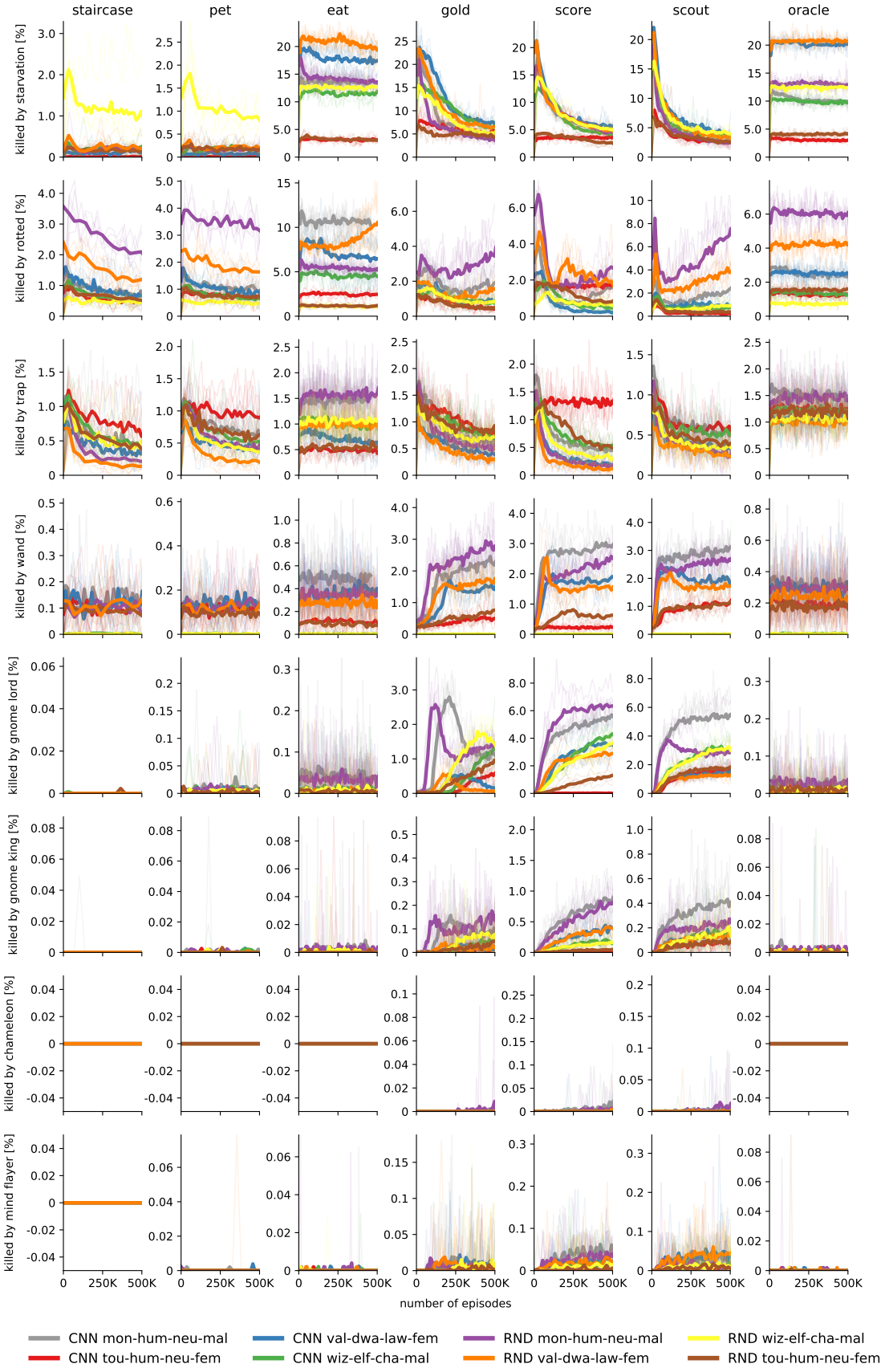


Figure C.4: Analysis of different causes of death during training, averaged over the last 1000 episodes and over five runs.

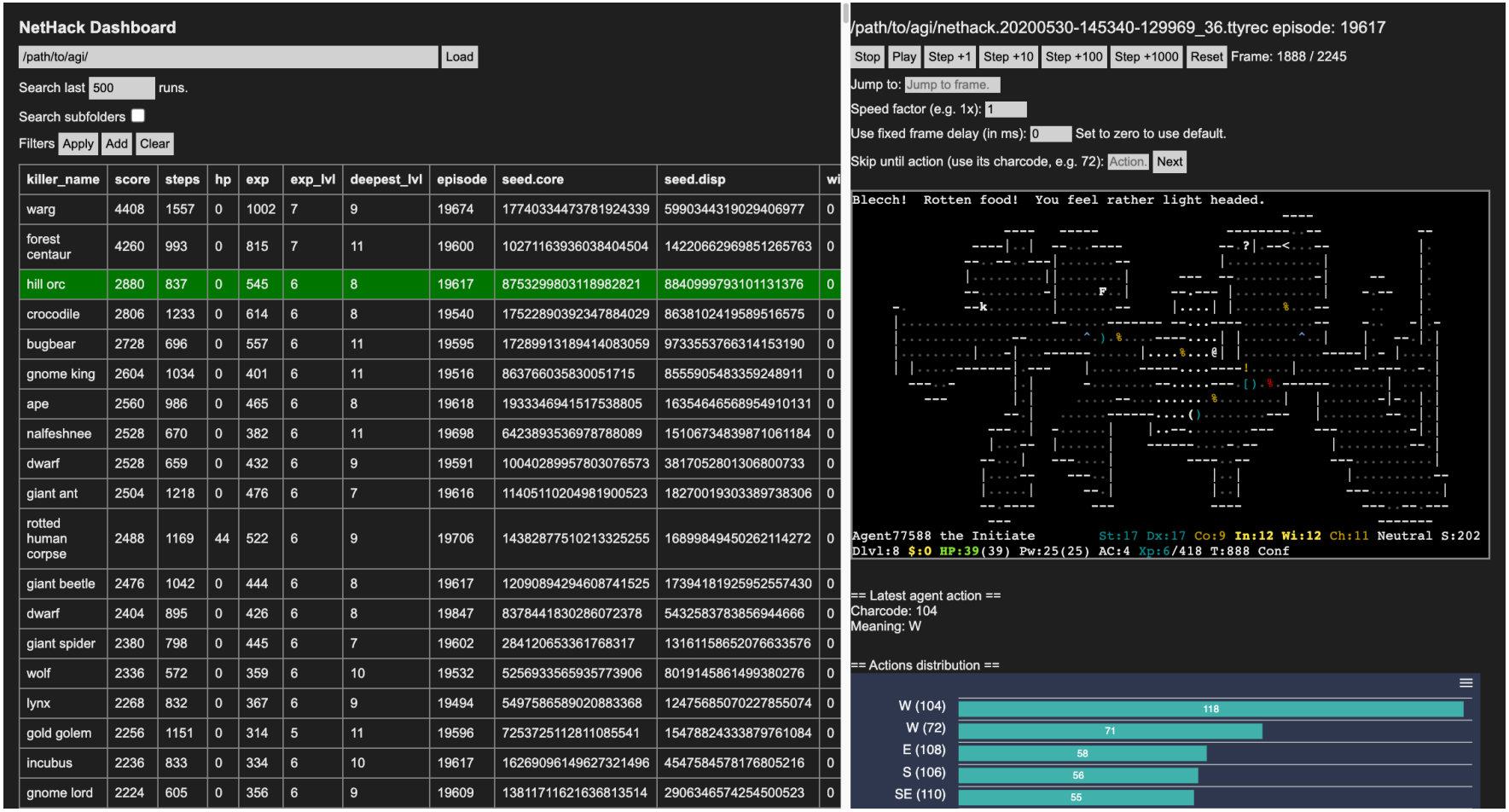


Figure C.5: Screenshot of the web dashboard included in the `NetHack Learning Environment`.

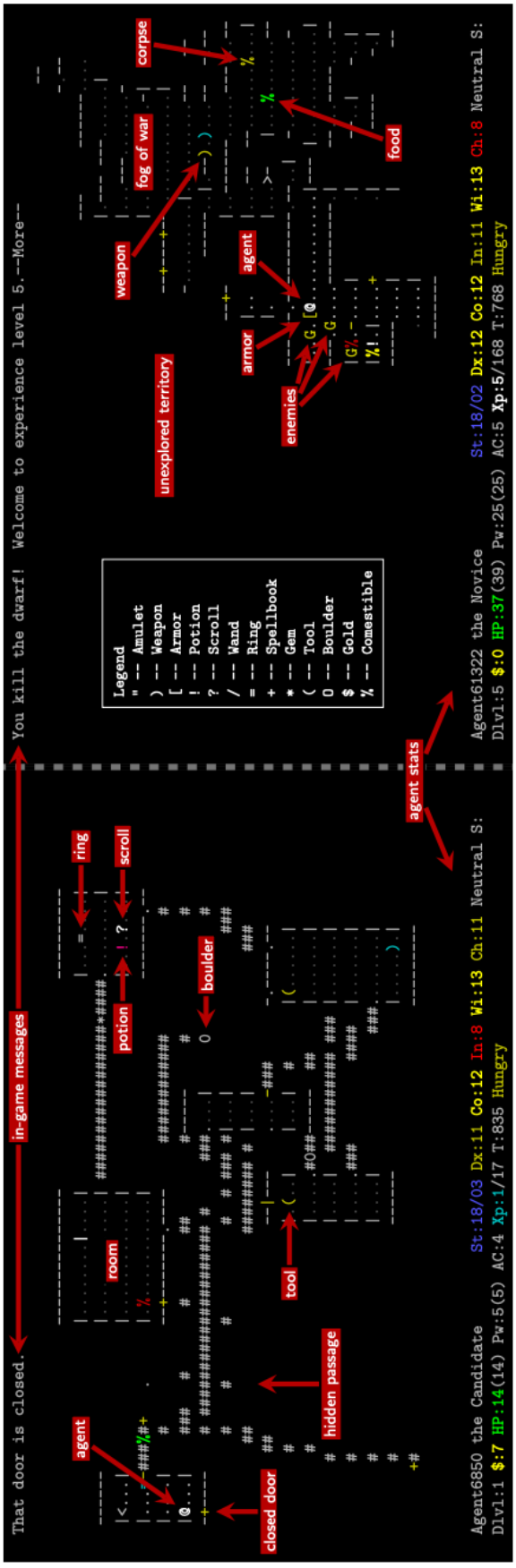


Figure C.6: Annotated example of an agent at two different stages in NetHack (Left: a procedurally generated first level of the Dungeons of Doom, right: Gnomish Mines).

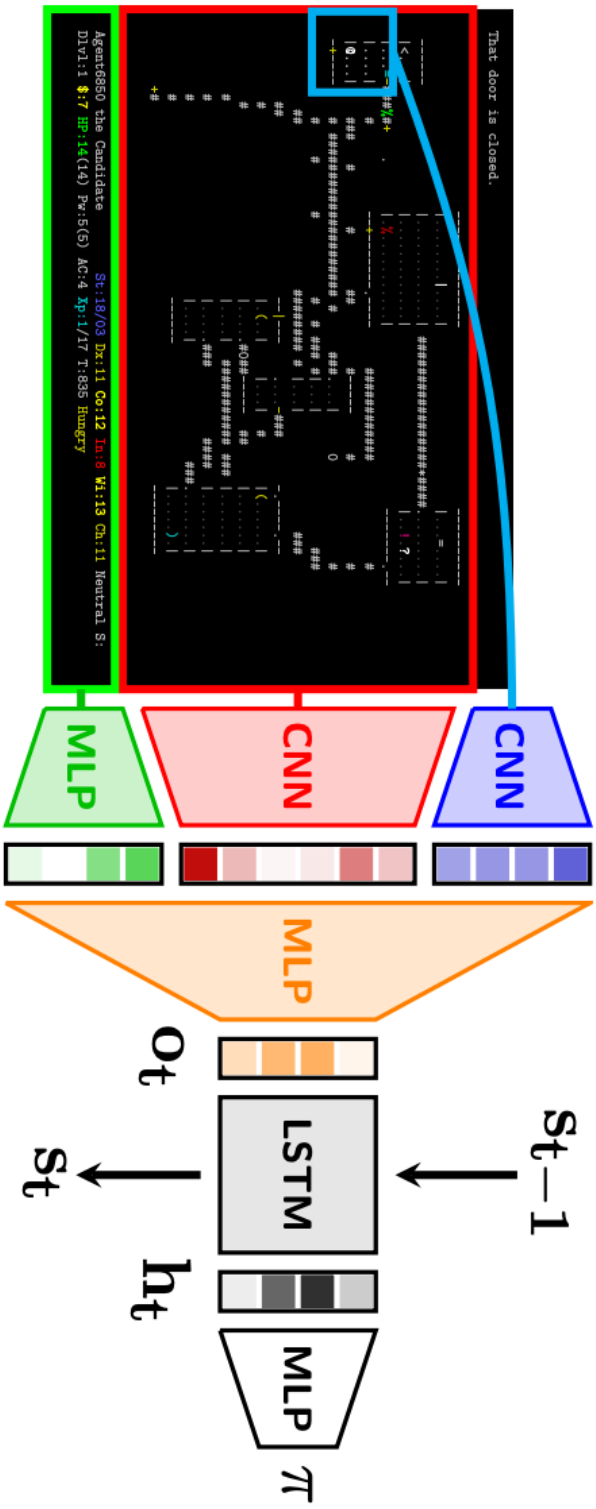


Figure C.7: Overview of the core architecture of the baseline models released with NLE.

References

[1] Pieter Abbeel and Andrew Y. Ng. "Apprenticeship learning via inverse reinforcement learning". In: *ICML*. 2004.

[2] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. "Deep reinforcement learning at the edge of the statistical precipice". In: *arXiv preprint arXiv:2108.13264* (2021).

[3] David W Aha, Matthew Molineaux, and Marc Ponsen. "Learning to win: Case-based plan selection in a real-time strategy game". In: *International Conference on Case-Based Reasoning*. Springer. 2005, pp. 5–20.

[4] Aransentin Breggan Hampe Pellsson. *SWAGGINZZZ*. https://pellsson.github.io/. Accessed: 2020-05-30. 2019.

[5] Brenna Argall, Sonia Chernova, Manuela M. Veloso, and Brett Browning. "A survey of robot learning from demonstration". In: *Robotics and Autonomous Systems* 57 (2009), pp. 469–483.

[6] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. "A brief survey of deep reinforcement learning". In: *arXiv preprint arXiv:1708.05866* (2017).

[7] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. "Deep reinforcement learning: A brief survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38.

[8] Andrea Asperti, Daniele Cortesi, Carlo De Pieri, Gianmaria Pedrini, and Francesco Sovrano. "Crawling in Rogue's dungeons with deep reinforcement techniques". In: *IEEE Transactions on Games* (2019).

[9] Yusuf Aytar, Tobias Pfaff, David Budden, Tom Le Paine, Ziyu Wang, and Nando de Freitas. "Playing hard exploration games by watching YouTube". In: *NeurIPS*. 2018.

[10] Andrea Banino, Caswell Barry, Benigno Uria, Charles Blundell, Timothy Lillicrap, Piotr Mirowski, Alexander Pritzel, Martin J Chadwick, Thomas Degris, Joseph Modayil, et al. "Vector-based navigation using grid-like representations in artificial agents". In: *Nature* (2018), p. 1.

[11] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. "DeepMind Lab". In: *CoRR* abs/1612.03801 (2016). arXiv: 1612.03801.

[12] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. "Deepmind lab". In: *arXiv preprint arXiv:1612.03801* (2016).

[13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[14] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. "Unifying count-based exploration and intrinsic motivation". In: *NeurIPS*. 2016.

[15] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* (2013).

[16] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680* (2019).

[17] Daniel S. Bernstein, Shlomo Zilberstein, and Neil Immerman. "The Complexity of Decentralized Control of Markov Decision Processes". In: *arXiv:1301.3836 [cs]* (Jan. 2013). arXiv: 1301.3836. URL: http://arxiv.org/abs/1301.3836 (visited on 12/29/2017).

[18] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II*. 4th. Athena Scientific, 2012.

[19] Shehroze Bhatti, Alban Desmaison, Ondrej Miksik, Nantas Nardelli, N. Siddharth, and Philip H. S. Torr. *Playing Doom with SLAM-Augmented Deep Reinforcement Learning*. Dec. 1, 2016. arXiv: 1612.00380 [cs, stat]. URL: http://arxiv.org/abs/1612.00380 (visited on 06/18/2020).

[20] Alejandro Bordallo, Fabio Previtali, Nantas Nardelli, and Subramanian Ramamoorthy. "Counterfactual Reasoning about Intent for Interactive Navigation in Dynamic Environments". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 2943–2950. arXiv: 1610.08424. URL: http://arxiv.org/abs/1610.08424.

[21] S. R. K. Branavan, David Silver, and Regina Barzilay. "Learning to Win by Reading Manuals in a Monte-Carlo Framework". In: *ACL*. 2011.

[22] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. "OpenAI Gym". In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540.

[23] Yuri Burda, Harrison Edwards, Deepak Pathak, Amos J. Storkey, Trevor Darrell, and Alexei A. Efros. "Large-Scale Study of Curiosity-Driven Learning". In: *ICLR*. 2019.

[24] Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. "Exploration by random network distillation". In: *ICLR*. 2019.

[25] Michael Buro. "Call for AI research in RTS games". In: *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*. AAAI press. 2004, pp. 139–142.

[26] Lucian Busoniu, Robert Babuska, and Bart De Schutter. "A comprehensive survey of multiagent reinforcement learning". In: *IEEE Transactions on Systems, Man, And Cybernetics-Part C: Applications and Reviews, 38 (2), 2008* (2008).

[27] Jonathan Campbell and Clark Verbrugge. "Learning Combat in NetHack". In: *AIIDE*. 2017.

[28] Jonathan Campbell and Clark Verbrugge. "Exploration in NetHack With Secret Discovery". In: *IEEE Transactions on Games* (2018).

[29] Yu-Han Chang, Tracey Ho, and Leslie Pack Kaelbling. "All learning is Local: Multi-agent Learning in Global Reward Games." In: *NIPS*. 2003, pp. 807–814.

[30] Devendra Singh Chaplot, Deepak Pathak, and Jitendra Malik. "Differentiable spatial planning using transformers". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 1484–1495.

[31] Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. "BabyAI: A Platform to Study the Sample Efficiency of Grounded Language Learning". In: *ICLR*. 2018.

[32] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. *Minimalistic Gridworld Environment for OpenAI Gym*. https://github.com/maximecb/gym-minigrid. 2018.

[33] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259* (2014).

[34] Sammy Christen, Lukas Jendele, Emre Aksan, and Otmar Hilliges. "Learning functionally decomposed hierarchies for continuous control tasks with path planning". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3623–3630.

[35] Kamil Ciosek and Shimon Whiteson. "OFFER: Off-Environment Reinforcement Learning". In: (2017).

[36] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. "Leveraging Procedural Generation to Benchmark Reinforcement Learning". In: *arXiv preprint arXiv:1912.01588* (2019).

[37] Karl Cobbe, Oleg Klimov, Christopher Hesse, Taehoon Kim, and John Schulman. "Quantifying Generalization in Reinforcement Learning". In: *ICML*. 2019.

[38] Mitchell K Colby, William J Curran, and Kagan Tumer. "Approximating Difference Evaluations with Local Information." In: *AAMAS*. 2015, pp. 1659–1660.

[39] Vincent Conitzer and Tuomas Sandholm. "AWESOME: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents". In: *Machine Learning* 67.1-2 (2007), pp. 23–43.

[40] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. http://pybullet.org. 2016–2021.

[41] Bruno C Da Silva, Eduardo W Basso, Ana LC Bazzan, and Paulo M Engel. "Dealing with non-stationary environments using context detection". In: *Proceedings of the 23rd international conference on Machine learning*. ACM. 2006, pp. 217–224.

[42] Dustin Dannenhauer, Michael W Floyd, Jonathan Decker, and David W Aha. "Dungeon Crawl Stone Soup as an Evaluation Domain for Artificial Intelligence". In: *Workshop on Games and Simulations for Artificial Intelligence, AAAI* (2019).

[43] Abhishek Das, Satwik Kottur, José MF Moura, Stefan Lee, and Dhruv Batra. "Learning Cooperative Visual Dialog Agents with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1703.06585* (2017).

[44] Peter Dayan and Geoffrey E. Hinton. "Feudal Reinforcement Learning". In: *NeurIPS*. 1992.

[45] DeepMind. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. 2019. URL: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ (visited on 02/05/2019).

[46] Marc Peter Deisenroth, Gerhard Neumann, Jan Peters, et al. "A survey on policy search for robotics". In: *Foundations and Trends® in Robotics* 2.1–2 (2013), pp. 1–142.

[47] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. "Go-Explore: A New Approach for Hard-exploration Problems". In: *arXiv preprint arXiv:1901.10995* (2019).

[48] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. "First return then explore". In: *CoRR* abs/2004.12919 (2020). arXiv: 2004.12919.

[49] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *ICML*. 2018.

[50] Eva Myers. *List of Nethack spoilers*. https://sites.google.com/view/evasroguelikegamessite/list-of-nethack-spoilers. Accessed: 2020-06-03. 2020.

[51] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. "TreeQN and ATreeC: Differentiable Tree Planning for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1710.11417* (2017).

[52] Juan Manuel Sanchez Fernandez. *Reinforcement Learning for roguelike type games (eliteMod v0.9)*. https://kcir.pwr.edu.pl/~witold/aiarr/2009_projekty/elitmod/. Accessed: 2020-01-19. 2009.

[53] Jakob Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. "Learning to communicate with deep multi-agent reinforcement learning". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2137–2145.

[54] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. "Counterfactual Multi-Agent Policy Gradients". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[55] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip H. S. 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*. 2017, pp. 1146–1155. arXiv: 1702.08887. URL: http://arxiv.org/abs/1702.08887 (visited on 06/18/2020).

[56] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip H. S. 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*. 2017, pp. 1146–1155.

[57] C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. "Brax-A Differentiable Physics Engine for Large Scale Rigid Body Simulation". In: (2021).

[58] Javier Garcıa and Fernando Fernández. "A comprehensive survey on safe reinforcement learning". In: *Journal of Machine Learning Research* (2015).

[59] Elizabeth Gibney. "This AI researcher is trying to ward off a reproducibility crisis". In: *Nature* 577.7788 (2020), p. 14.

[60] Sven Gronauer, Martin Gottwald, and Klaus Diepold. "The Successful Ingredients of Policy Gradient Algorithms". In: (2020).

[61] Edward Groshev, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. "Learning generalized reactive policies using deep neural networks". In: *arXiv preprint arXiv:1708.07280* (2017).

[62] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. "Cooperative Multi-Agent Control Using Deep Reinforcement Learning". In: (2017).

[63] Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. "Cognitive mapping and planning for visual navigation". In: *arXiv preprint arXiv:1702.03920* (2017).

[64] William H. Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, et al. "The MineRL Competition on Sample Efficient Reinforcement Learning using Human Priors". In: *NeurIPS Competition Track* (2019).

[65] Luke Harries, Sebastian Lee, Jaroslaw Rzepecki, Katja Hofmann, and Sam Devlin. "MazeExplorer: A Customisable 3D Benchmark for Assessing Generalisation in Reinforcement Learning". In: *IEEE Conference on Games*. 2019.

[66] Matthew Hausknecht, Prannoy Mupparaju, Sandeep Subramanian, S Kalyanakrishnan, and P Stone. "Half field offense: an environment for multiagent learning and ad hoc teamwork". In: *AAMAS Adaptive Learning Agents (ALA) Workshop*. 2016.

[67] Matthew Hausknecht, Prannoy Mupparaju, Sandeep Subramanian, S Kalyanakrishnan, and P Stone. "Half field offense: an environment for multiagent learning and ad hoc teamwork". In: *AAMAS Adaptive Learning Agents (ALA) Workshop*. 2016.

[68] Matthew Hausknecht and Peter Stone. “Deep recurrent q-learning for partially observable mdps”. In: *CoRR, abs/1507.06527* (2015).

[69] Matthew Hausknecht and Peter Stone. “Deep Recurrent Q-Learning for Partially Observable MDPs”. In: *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents*. 2015.

[70] 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*. 2016, pp. 1804–1813.

[71] Johannes Heinrich and David Silver. “Deep Reinforcement Learning from Self-Play in Imperfect-Information Games”. In: *CoRR* abs/1603.01121 (2016). arXiv: 1603.01121. URL: http://arxiv.org/abs/1603.01121.

[72] 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.

[73] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. “Deep Q-learning From Demonstrations”. In: *AAAI*. 2017.

[74] Felix Hill, Andrew K. Lampinen, Rosalia Schneider, Stephen Clark, Matthew Botvinick, James L. McClelland, and Adam Santoro. “Emergent Systematic Generalization in a Situated Agent”. In: *ICLR*. 2020.

[75] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[76] Maximilian Igl, Andrew Gambardella, Jinke He, Nantas Nardelli, N. Siddharth, Wendelin Boehmer, and Shimon Whiteson. “Multitask Soft Option Learning”. In: *Conference on Uncertainty in Artificial Intelligence*. Conference on Uncertainty in Artificial Intelligence. PMLR, Aug. 27, 2020, pp. 969–978. arXiv: 1904.01033. URL: http://proceedings.mlr.press/v124/igl20a.html (visited on 11/10/2020).

[77] Yacine Jernite, Kavya Srinet, Jonathan Gray, and Arthur Szlam. “CraftAssist Instruction Parsing: Semantic Parsing for a Minecraft Assistant”. In: *CoRR* abs/1905.01978 (2019). arXiv: 1905.01978.

[78] Minqi Jiang, Jelena Luketina, Nantas Nardelli, Pasquale Minervini, Philip H. S. Torr, Shimon Whiteson, and Tim Rocktäschel. “WordCraft: An Environment for Benchmarking Commonsense Agent”. In: *Language in Reinforcement Learning, ICML*. July 2020. URL: https://arxiv.org/abs/2007.09185.

[79] Yiding Jiang, Shixiang Gu, Kevin Murphy, and Chelsea Finn. “Language as an Abstraction for Hierarchical Deep Reinforcement Learning”. In: *NeurIPS*. 2019.

[80] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. “The Malmo Platform for Artificial Intelligence Experimentation.” In: *IJCAI*. Citeseer. 2016, pp. 4246–4247.

[81] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. “The Malmo Platform for Artificial Intelligence Experimentation”. In: *IJCAI*. 2016.

[82] Emilio Jorge, Mikael Kågebäck, and Emil Gustavsson. "Learning to Play Guess Who? and Inventing a Grounded Language as a Consequence". In: *arXiv preprint arXiv:1611.03218* (2016).

[83] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, et al. "Unity: A general platform for intelligent agents". In: *arXiv preprint arXiv:1809.02627* (2018).

[84] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. "Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning". In: *IJCAI*. 2019.

[85] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. "Illuminating generalization in deep reinforcement learning through procedural level generation". In: *arXiv preprint arXiv:1806.10729* (2018).

[86] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* (1996).

[87] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[88] Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. "Half field offense in RoboCup soccer: A multiagent reinforcement learning case study". In: *Robot Soccer World Cup*. Springer. 2006, pp. 72–85.

[89] Yuji Kanagawa and Tomoyuki Kaneko. "Rogue-gym: A new challenge for generalization in reinforcement learning". In: *IEEE Conference on Games*. 2019.

[90] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. "Vizdoom: A doom-based ai research platform for visual reinforcement learning". In: *IEEE Conference on Computational Intelligence and Games*. 2016.

[91] Kenneth Lorber. *NetHack Home Page*. https://nethack.org. Accessed: 2020-05-30. 2020.

[92] Arbaaz Khan, Clark Zhang, Nikolay Atanasov, Konstantinos Karydis, Vijay Kumar, and Daniel D Lee. "Memory Augmented Control Networks". In: *arXiv preprint arXiv:1709.05706* (2017).

[93] 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*. ACM. 1997, pp. 340–347.

[94] Jelle R Kok and Nikos Vlassis. "Collaborative multiagent reinforcement learning by payoff propagation". In: *Journal of Machine Learning Research* 7.Sep (2006), pp. 1789–1828.

[95] Vijay R Konda and John N Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems*. 2000, pp. 1008–1014.

[96] Landon Kraemer and Bikramjit Banerjee. "Multi-agent reinforcement learning as a rehearsal for decentralized planning". In: *Neurocomputing* 190 (2016), pp. 82–94.

[97] Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zając, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, et al. "Google research football: A novel reinforcement learning environment". In: *arXiv preprint arXiv:1907.11180* (2019).

[98] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. *TorchBeast: A PyTorch Platform for Distributed RL*. Oct. 8, 2019. arXiv: 1910.03552 [cs, stat]. URL: http://arxiv.org/abs/1910.03552 (visited on 06/18/2020).

[99] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. "TorchBeast: A PyTorch Platform for Distributed RL". In: *arXiv* abs/1910.03552 (2019).

[100] Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. "The NetHack Learning Environment". In: *Advances in Neural Information Processing Systems*. Vol. 33. Dec. 2020. arXiv: 2006.13760. URL: http://arxiv.org/abs/2006.13760 (visited on 06/25/2020).

[101] 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*. Citeseer. 2000.

[102] Angeliki Lazaridou, Alexander Peysakhovich, and Marco Baroni. "Multi-agent cooperation and the emergence of (natural) language". In: *arXiv preprint arXiv:1612.07182* (2016).

[103] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[104] Namhoon Lee, Wongun Choi, Paul Vernaza, Christopher B Choy, Philip HS Torr, and Manmohan Chandraker. "DESIRE: Distant Future Prediction in Dynamic Scenes with Interacting Agents". In: *arXiv preprint arXiv:1704.04394* (2017).

[105] Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. "Multi-agent Reinforcement Learning in Sequential Social Dilemmas". In: *arXiv preprint arXiv:1702.03037* (2017).

[106] Joel Z. Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. "Multi-agent Reinforcement Learning in Sequential Social Dilemmas". In: (Feb. 2017). arXiv: 1702.03037. URL: http://arxiv.org/abs/1702.03037.

[107] David Lopez-Paz and Marc'Aurelio Ranzato. "Gradient episodic memory for continual learning". In: *NeurIPS*. 2017.

[108] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. "Multi-agent actor-critic for mixed cooperative-competitive environments". In: *arXiv preprint arXiv:1706.02275* (2017).

[109] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments". In: (June 2017). arXiv: 1706.02275. URL: http://arxiv.org/abs/1706.02275.

[110] Jelena Luketina, Nantas Nardelli, Gregory Farquhar, Jakob Foerster, Jacob Andreas, Edward Grefenstette, Shimon Whiteson, and Tim Rocktäschel. "A Survey of Reinforcement Learning Informed by Natural Language". In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press, 2019, pp. 6309–6317. arXiv: 1906.03926. URL: https://arxiv.org/abs/1906.03926.

[111] M. Drew Streib. *Public NetHack server at alt.org (NAO)*. https://alt.org/nethack/. Accessed: 2020-05-30. 2020.

[112] Daniel J. Mankowitz, Augustin Zídek, André Barreto, Dan Horgan, Matteo Hessel, John Quan, Junhyuk Oh, Hado van Hasselt, David Silver, and Tom Schaul. "Unicorn: Continual Learning with a Universal, Off-policy Agent". In: *arXiv* abs/1802.08294 (2018).

[113] Maja J Mataric. "Using communication to reduce locality in distributed multiagent learning". In: *Journal of experimental & theoretical artificial intelligence* 10.3 (1998), pp. 357–369.

[114] Laetitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. "Independent reinforcement learners in cooperative Markov games: a survey regarding coordination problems". In: *The Knowledge Engineering Review* 27.01 (2012), pp. 1–31.

[115] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andy Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, et al. "Learning to navigate in complex environments". In: *arXiv preprint arXiv:1611.03673* (2016).

[116] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. "Human-level control through deep reinforcement learning". In: 518.7540 (Feb. 2015), pp. 529–533. URL: http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html?foxtrotcallback=true (visited on 09/07/2017).

[117] 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". In: *Nature* 518.7540 (2015), pp. 529–533.

[118] Igor Mordatch and Pieter Abbeel. "Emergence of Grounded Compositional Language in Multi-Agent Populations". In: *arXiv preprint arXiv:1703.04908* (2017).

[119] Alex Muryy, N. Siddharth, Nantas Nardelli, Andrew Glennerster, and Philip H. S. Torr. "Lessons from Reinforcement Learning for Biological Representations of Space". In: *Vision Research* 174 (Sept. 1, 2020), pp. 79–93. arXiv: 1912.06615. URL: http://www.sciencedirect.com/science/article/pii/S0042698920300997 (visited on 11/10/2020).

[120] Nantas Nardelli, Gabriel Synnaeve, Zeming Lin, Pushmeet Kohli, Philip H.S. Torr, and Nicolas Usunier. "Value Propagation Networks". In: *ICLR*. 2019.

[121] NetHack Wiki. *NetHackWiki*. https://nethackwiki.com/. Accessed: 2020-02-01. 2020.

[122] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. "Gotta learn fast: A new benchmark for generalization in rl". In: *arXiv preprint arXiv:1804.03720* (2018).

[123] Sufeng Niu, Siheng Chen, Hanyu Guo, Colin Targonski, Melissa C Smith, and Jelena Kovačević. "Generalized Value Iteration Networks: Life Beyond Lattices". In: *arXiv preprint arXiv:1706.02416* (2017).

[124] Junhyuk Oh, Satinder Singh, and Honglak Lee. "Value Prediction Network". In: *arXiv preprint arXiv:1707.03497* (2017).

[125] Frans A. Oliehoek and Christopher Amato. *A Concise Introduction to Decentralized POMDPs*. SpringerBriefs in Intelligent Systems. Springer, 2016. URL: http://www.springer.com/us/book/9783319289274.

[126] Frans A. Oliehoek, Matthijs T. J. Spaan, and Nikos Vlassis. "Optimal and Approximate Q-value Functions for Decentralized POMDPs". In: *JAIR* 32 (2008), pp. 289–353.

[127] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. "A survey of real-time strategy game AI research and competition in StarCraft". In: *IEEE Transactions on Computational Intelligence and AI in games* 5.4 (2013), pp. 293–311.

[128] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvári, Satinder Singh, Benjamin Van Roy, Richard Sutton, David Silver, and Hado van Hasselt. "Behaviour Suite for Reinforcement Learning". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=rygf-kSYwH.

[129] Georg Ostrovski, Marc G Bellemare, Aäron van den Oord, and Rémi Munos. "Count-based exploration with neural density models". In: *ICML*. 2017.

[130] Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu, and Tong Lu. "On reinforcement learning for full-length game of starcraft". In: *arXiv preprint arXiv:1809.09095* (2018).

[131] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. "Continual lifelong learning with neural networks: A review". In: *Neural Networks* (2019).

[132] Ronald Parr and Stuart J. Russell. "Reinforcement Learning with Hierarchies of Machines". In: *NeurIPS*. 1997.

[133] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch". In: (2017).

[134] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. "Curiosity-driven Exploration by Self-supervised Prediction". In: *ICML*. 2017.

[135] 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". In: *arXiv preprint arXiv:1703.10069* (2017).

[136] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. "Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program)". In: *Journal of Machine Learning Research* 22 (2021).

[137] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. *Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research*. 2018. eprint: arXiv:1802.09464.

[138] Scott Proper and Kagan Tumer. "Modeling difference rewards for multiagent learning." In: *AAMAS*. 2012, pp. 1397–1398.

[139] Roberta Raileanu and Tim Rocktäschel. "RIDE: Rewarding Impact-Driven Exploration for Procedurally-Generated Environments". In: *ICLR*. 2020.

[140] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. "QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning*. 2018, pp. 4295–4304.

[141] Eric S. Raymond. *A Guide to the Mazes of Menace*. 1987.

[142] Eric S. Raymond, Mike Stephenson, et al. *A Guide to the Mazes of Menace: Guidebook for NetHack*. NetHack DevTeam. 2020. URL: http://www.nethack.org/download/3.6.5/nethack-365-Guidebook.pdf.

[143] Eike Rehder, Maximilian Naumann, Niels Ole Salscheider, and Christoph Stiller. "Cooperative Motion Planning for Non-Holonomic Agents with Value Iteration Networks". In: *arXiv preprint arXiv:1709.05273* (2017).

[144] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. "Pommerman: A Multi-Agent Playground". In: *arXiv preprint arXiv:1809.07124* (2018).

[145] Sebastian Risi and Julian Togelius. "Procedural Content Generation: From Automatically Generating Game Levels to Increasing Generality in Machine Learning". In: *CoRR* abs/1911.13071 (2019). arXiv: 1911.13071.

[146] CP Robert and G Casella. "Monte Carlo Statistical Methods Springer". In: *New York* (2004).

[147] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. "Experience replay for continual learning". In: *NeurIPS*. 2019.

[148] Stuart Russell, Peter Norvig, and Artificial Intelligence. "A modern approach". In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995), p. 27.

[149] Tim Salimans and Richard Chen. "Learning Montezuma's Revenge from a Single Demonstration". In: *CoRR* abs/1812.03381 (2018). arXiv: 1812.03381.

[150] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philip H. S. Torr, Jakob Foerster, and Shimon Whiteson. "The StarCraft Multi-Agent Challenge". In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 2186–2188. arXiv: 1902.04043. URL: http://arxiv.org/abs/1902.04043 (visited on 06/18/2020).

[151] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. "Habitat: A Platform for Embodied AI Research". In: *ArXiv* abs/1904.01201 (2019).

[152] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized Experience Replay". In: *CoRR* abs/1511.05952 (2015).

[153] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.

[154] Christian Schroeder, Bradley Gram-Hansen, Nantas Nardelli, Andrew Gambardella, Rob Zinkov, Puneet Dokania, N. Siddharth, Ana Belen Espinosa-Gonzalez, Ara Darzi, Philip H. S. Torr, and Atılım Güneş Baydin. "Simulation-Based Inference for Global Health Decisions". In: *ML for Global Health, ICML*. Apr. 2020. arXiv: 2005.07062. URL: http://arxiv.org/abs/2005.07062 (visited on 06/18/2020).

[155] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *CoRR* abs/1506.02438 (2015). URL: http://arxiv.org/abs/1506.02438.

[156] Iulian Vlad Serban, Chinnadhurai Sankar, Michael Pieper, Joelle Pineau, and Yoshua Bengio. "The Bottleneck Simulator: A Model-based Deep Reinforcement Learning Approach". In: *arXiv preprint arXiv:1807.04723* (2018).

[157] Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. New York: Cambridge University Press, 2009.

[158] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, et al. "The predictron: End-to-end learning and planning". In: *arXiv preprint arXiv:1612.08810* (2016).

[159] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[160] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), p. 354.

[161] Viswanath Sivakumar, Tim Rocktäschel, Alexander H. Miller, Heinrich Küttler, Nantas Nardelli, Mike Rabbat, Joelle Pineau, and Sebastian Riedel. *MVFST-RL: An Asynchronous RL Framework for Congestion Control with Delayed Actions*. Oct. 30, 2019. arXiv: 1910.04054 [cs, stat]. URL: http://arxiv.org/abs/1910.04054 (visited on 06/18/2020).

[162] Kyunghwan Son, Daewoo Kim, Wan Ju Kang, David Earl Hostallero, and Yung Yi. "QTRAN: Learning to Factorize with Transformation for Cooperative Multi-Agent Reinforcement Learning". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 5887–5896.

[163] Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. "Universal Planning Networks: Learning Generalizable Representations for Visuomotor Control". In: *International Conference on Machine Learning*. 2018, pp. 4739–4748.

[164] Peter Stone, Gregory Kuhlmann, Matthew E Taylor, and Yaxin Liu. "Keepaway soccer: From machine learning testbed to benchmark". In: *Robot Soccer World Cup*. Springer. 2005, pp. 93–105.

[165] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. "Learning Multiagent Communication with Backpropagation". In: *arXiv preprint arXiv:1605.07736* (2016).

[166] Sainbayar Sukhbaatar, Arthur Szlam, Gabriel Synnaeve, Soumith Chintala, and Rob Fergus. "Mazebase: A sandbox for learning from games". In: *arXiv preprint arXiv:1511.07401* (2015).

[167] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Ji Liu, Yongsheng Liu, Han Liu, et al. "Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game". In: *arXiv preprint arXiv:1809.07193* (2018).

[168] 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". In: *arXiv:1706.05296 [cs]* (June 2017). arXiv: 1706.05296. URL: http://arxiv.org/abs/1706.05296 (visited on 12/19/2017).

[169] Richard S Sutton. "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1 (1988), pp. 9–44.

[170] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[171] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." In: *NIPS*. Vol. 99. 1999, pp. 1057–1063.

[172] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. *TorchCraft: A Library for Machine Learning Research on Real-Time Strategy Games*. Nov. 3, 2016. arXiv: 1611.00625 [cs]. URL: http://arxiv.org/abs/1611.00625 (visited on 06/18/2020).

[173] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. "TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games". In: (Nov. 2016). arXiv: 1611.00625. URL: http://arxiv.org/abs/1611.00625.

[174] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. "TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games". In: *arXiv preprint arXiv:1611.00625* (2016).

[175] TAEB. *TAEB Documentation: Other Bots*. https://taeb.github.io/bots.html. Accessed: 2020-01-19. 2015.

[176] Aviv Tamar, Sergey Levine, Pieter Abbeel, YI WU, and Garrett Thomas. "Value iteration networks". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2146–2154.

[177] 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". In: *arXiv preprint arXiv:1511.08779* (2015).

[178] 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". In: (Nov. 2015). arXiv: 1511.08779. URL: http://arxiv.org/abs/1511.08779.

[179] Ming Tan. "Multi-agent reinforcement learning: Independent vs. cooperative agents". In: *Proceedings of the tenth international conference on machine learning*. 1993, pp. 330–337.

[180] Ming Tan. "Multi-agent reinforcement learning: Independent vs. cooperative agents". In: *Proceedings of the tenth international conference on machine learning*. 1993, pp. 330–337.

[181] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. "# Exploration: A study of count-based exploration for deep reinforcement learning". In: *NeurIPS*. 2017.

[182] Matthew E. Taylor and Peter Stone. "Transfer Learning for Reinforcement Learning Domains: A Survey". In: *Journal Machine Learning Research* (2009).

[183] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[184] Gerald Tesauro. "Extending Q-learning to general adaptive multi-agent systems". In: *Advances in neural information processing systems*. 2003, None.

[185] Blizzard Entertainment. *New Blizzard Custom Game: StarCraft Master*. http://us.battle.net/sc2/en/blog/4544189/new-blizzard-custom-game-starcraft-master-3-1-2012. Accessed: 2018-11-16. 2012.

[186] Emanuel Todorov, Tom Erez, and Yuval Tassa. "Mujoco: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033.

[187] Kagan Tumer and Adrian Agogino. "Distributed agent-based air traffic flow management". In: *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. ACM. 2007, p. 255.

[188] Nicolas Usunier, Gabriel Synnaeve, Zeming Lin, and Soumith Chintala. "Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks". In: *arXiv preprint arXiv:1609.02993* (2016).

[189] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Manfred Otto Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. "FeUdal Networks for Hierarchical Reinforcement Learning". In: *ICML*. 2017.

[190] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* (2019).

[191] 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". In: *arXiv preprint arXiv:1708.04782* (2017).

[192] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. "StarCraft II: A New Challenge for Reinforcement Learning". In: *arXiv:1708.04782 [cs]* (Aug. 2017). arXiv: 1708.04782. URL: http://arxiv.org/abs/1708.04782 (visited on 02/04/2018).

[193] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. "Sample efficient actor-critic with experience replay". In: *arXiv preprint arXiv:1611.01224* (2016).

[194] Christopher Watkins. "Learning from delayed rewards". PhD thesis. University of Cambridge England, 1989.

[195] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[196] Lex Weaver and Nigel Tao. "The optimal reward baseline for gradient-based reinforcement learning". In: *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 2001, pp. 538–545.

[197] Théophane Weber, Sébastien Racanière, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. "Imagination-Augmented Agents for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1707.06203* (2017).

[198] Danny Weyns, Alexander Helleboogh, and Tom Holvoet. "The packet-world: A test bed for investigating situated multi-agent systems". In: *Software Agent-Based Applications, Platforms and Development Kits*. Springer, 2005, pp. 383–408.

[199] Martha White. "Unifying task specification in reinforcement learning". In: *arXiv preprint arXiv:1609.01995* (2016).

[200] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[201] Jan Wöhlke, Felix Schmitt, and Herke van Hoof. "Hierarchies of Planning and Reinforcement Learning for Robot Navigation". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 10682–10688.

[202] David H Wolpert and Kagan Tumer. "Optimal payoff functions for members of collectives". In: *Modeling complexity in economic and social systems*. World Scientific, 2002, pp. 355–369.

[203] Erfu Yang and Dongbing Gu. *Multiagent reinforcement learning for multi-robot systems: A survey*. Tech. rep. tech. rep, 2004.

[204] Yaodong Yang, Rui Luo, Minne Li, Ming Zhou, Weinan Zhang, and Jun Wang. "Mean Field Multi-Agent Reinforcement Learning". In: *arXiv preprint arXiv:1802.05438* (2018).

[205] Chang Ye, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. "Rotation, Translation, and Cropping for Zero-Shot Generalization". In: *CoRR* abs/2001.09908 (2020).

[206] E. Zawadzki, A. Lipson, and K. Leyton-Brown. "Empirically Evaluating Multiagent Learning Algorithms". In: *arXiv preprint 1401.8074* (2014).

[207] Jingwei Zhang, Lei Tai, Joschka Boedecker, Wolfram Burgard, and Ming Liu. "Neural SLAM". In: *arXiv preprint arXiv:1706.09520* (2017).

[208] Lianmin Zheng, Jiacheng Yang, Han Cai, Weinan Zhang, Jun Wang, and Yong Yu. "MAgent: A Many-Agent Reinforcement Learning Platform for Artificial Collective Intelligence". In: *arXiv preprint arXiv:1712.00600* (2017).

[209] Victor Zhong, Tim Rocktäschel, and Edward Grefenstette. "RTFM: Generalising to New Environment Dynamics via Reading". In: *ICLR*. 2020.