# INVESTIGATING NOVEL REPRESENTATIONS AND DEEP REINFORCEMENT LEARNING FOR GENERAL GAME PLAYING

A TECHNICAL REPORT SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Supervisor

Dr. Ji Ruan

Prof. Ajit Narayanan

15 December 2022

By

Alvaro Gunawan

School of Engineering, Computer and Mathematical Sciences

# Abstract

General Game Playing (GGP) is a platform for developing general Artificial Intelligence algorithms for agents that can play any game given a description encoded in Game Description Language (GDL). The recent accomplishments of AlphaGo and AlphaZero have motivated new work in applying deep learning approaches to GGP. As GGP's general nature requires the inference of game rules prior to game playing, there are two main learning tasks: (i) inference of game rules in GDL and (ii) strategy optimisation to maximise utility. This requires an investigation into novel representations and deep reinforcement learning architectures that can be applied to the general nature of the games in GGP and take advantage of the logical nature of game rules and descriptions. Accordingly to the two main tasks mentioned before, we tackle each task separately with our contributions are as follows. Firstly, we present GGPDeepRL, an AlphaZero-style GGP agent that utilises deep reinforcement learning to learn to play general games. This agent adapts the deep reinforcement learning algorithm originally presented in AlphaZero to accommodate the general games of GGP as they can be multi-player, general-sum and simultaneous action. It is able to overcome the game-specific limitations of AlphaZero as it constructs its neural network based on the game description provided. Our results confirm the feasibility of applying deep reinforcement learning to GGP but note certain limitations with regards to the neural network architecture used. Secondly, we present a graph neural network based *neural reasoner*, utilising novel graph-based representations of the game rules to learn to

approximate the inference task. The neural reasoner uses *instantiated rule graphs* as input, a general, game-agnostic graph-based representation of game states described in GDL. Instantiated rule graphs allow for a single neural reasoner to be trained to infer over states across multiple games, as the representation is capable of capturing game state features while still being consistent across different games. We investigate the effect of different labelling functions for the instantiated rule graph as well as varying graph neural network architectures and parameters. The neural reasoner is capable of accurately inferring legal actions and subsequent states as well as transfer these learned inferences across different games. Our findings suggest that deep learning techniques can be applied to GGP effectively if the representations and architectures used are capable of taking advantage of general structural features present in games. These new approaches open the door to an entirely new class of GGP agents that would be capable of learning to play games in a truly general manner.

# Contents

**Abstract** **2**

**Attestation of Authorship** **9**

**Publications** **10**

**Acknowledgements** **11**

**1 Introduction** **13**

**2 Literature Review** **20**
2.1 Introduction . . . . . 20
2.2 Neural Networks . . . . . 23
2.2.1 NNs for Logic . . . . . 24
2.3 Reinforcement Learning for Games . . . . . 25
2.4 General Game Playing . . . . . 28
2.5 Conclusion . . . . . 33

**3 Background** **34**
3.1 Introduction . . . . . 34
3.2 General Game Playing . . . . . 35
3.3 Game Description Language . . . . . 36
3.3.1 Modularity . . . . . 41
3.3.2 Scrambling and renaming . . . . . 41
3.3.3 Grounding . . . . . 42
3.4 GGP Environment and Agent Methods . . . . . 42
3.4.1 Game Management Infrastructure . . . . . 43
3.4.2 Automated Reasoning . . . . . 44
3.4.3 Search Methods . . . . . 45
3.5 Deep Learning for Games . . . . . 52
3.5.1 Neural Networks . . . . . 52
3.5.2 Deep Reinforcement Learning . . . . . 59
3.6 Conclusion . . . . . 61

**4 Exploring a Learning Architecture for GGP** **62**
4.1 Introduction . . . . . 62
4.2 GGP Learning Agent Architecture . . . . . 64
4.2.1 Reasoner . . . . . 65
4.2.2 Neural Network . . . . . 66
4.2.3 Search algorithm . . . . . 68
4.2.4 Reinforcement Learning . . . . . 69
4.3 Experimental Studies and Results . . . . . 70
4.3.1 Two-player zero-sum games . . . . . 72
4.3.2 Single player games . . . . . 73
4.3.3 Multi player games . . . . . 74
4.3.4 Neural Network Depth . . . . . 77
4.3.5 Convolutional Neural Network Experiments . . . . . 78
4.3.6 Effects on Simulation-limited and Time-limited MCTS . . . . 80
4.4 Conclusion . . . . . 83

**5 A Graph Neural Network Reasoner for GDL** **86**
5.1 Introduction . . . . . 86
5.2 Overview of the Neural Reasoner . . . . . 89
5.3 Game State Representations . . . . . 92
5.3.1 Rule Graphs . . . . . 93
5.3.2 Instantiated Rule Graphs . . . . . 99
5.4 Machine Learning Framework . . . . . 101
5.4.1 Node-Vector Embeddings . . . . . 102
5.4.2 Graph Neural Network Architecture . . . . . 103
5.4.3 Dataset Generation . . . . . 105
5.4.4 Training Details . . . . . 109
5.5 Results and Discussion . . . . . 110
5.5.1 Training and Evaluation on Individual Games . . . . . 110
5.5.2 Transfer Learning and Mixed Training . . . . . 118
5.5.3 Discussion . . . . . 125
5.6 Conclusion . . . . . 126

**6 Conclusion** **129**

**References** **132**

# List of Tables


2.1 Winners of the annual AAAI GGP Competition . . . . . 30

4.1 Features of games used in experiments. Results are presented as win/loss/draws. . . . . 72
4.2 GGPDeepRL 's performance on two-player zero-sum games against random, NN and MCTS agents. . . . . 72
4.3 Comparison of average utility over 100 games on single player games. 73
4.4 Total goal value of various agents playing 20 games of Pacman3p . . . . . 75
4.5 Average utility of agents in connectfour3p over 100 games. . . . . 76
4.6 Results of playing 100 games of Babel . . . . . 76

5.1 Vector embedding values . . . . . 102
5.2 Graph metrics of game datasets . . . . . 107
5.3 Neural reasoner accuracy over 100 games. . . . . 111
5.4 Next fluent and legal action prediction for 100 games of standard and flattened games . . . . . 114
5.5 Next fluent and legal action prediction with varying labelling functions 115
5.6 Model accuracy over 100 games of $C4_{3p}$. . . . . 117
5.7 Neural reasoner accuracy over 100 games for `next` and `legal` prediction. . . . . 119
5.8 Neural reasoner accuracy in random and sequential mixed training. . . 121

# List of Figures


1.1 Timeline of developments in computer game playing . . . . . 13

3.1 A Tic-Tac-Toe state and its corresponding state in GDL. . . . . 37
3.2 Tic-Tac-Toe description in GDL . . . . . 39
3.3 State trasition of a Tic-Tac-Toe state in GDL. . . . . 40
3.4 Network Game Manager . . . . . 43
3.5 Minimax applied to an example game tree. . . . . 47
3.6 Monte Carlo Tree Search . . . . . 49
3.7 A fully-connected neural network . . . . . 54
3.8 A convolutional layer. . . . . 56
3.9 A message-passing GNN layer. . . . . 58

4.1 Architecture of GGPDeepRL agent. . . . . 65
4.2 Generating a fluent list from GDL state . . . . . 67
4.3 Fully connected neural network architecture generated from game description . . . . . 68
4.4 GGPZero Search Algorithm . . . . . 69
4.5 Average utility in hamilton with increasing training iterations over 100 games . . . . . 74
4.6 Average utility in knightstour with varying number of layers and training iterations over 100 games . . . . . 77
4.7 Winning rate and training time for GGPDeepRL with varying number of hidden layers. . . . . 78
4.8 Games won by CNN agent versus FCNN agent in 100 games of connectfour . . . . . 79
4.9 Training loss of FCNN and CNN on connectfour. . . . . 80
4.10 GGPDeepRL winning rate against MCTS with increasing time and simulation limits with varying amounts of training iterations on Breakthrough games. . . . . 81
4.11 Average number of simulations completed by agents . . . . . 82

5.1 Fragment of GDL description of Tic-Tac-Toe. . . . . 89
5.2 Overview of the Neural Reasoner . . . . . 91
5.3 Rule graph of the fragment in Figure 5.1 . . . . . 95

5.4 Standard definition of a rule in Tic-Tac-Toe and two instances of the grounded rule . . . . . 99
5.5 An example of an instantiated rule graph for state $S = \{\texttt{cell}(1,1,\texttt{b})\}$. With minimal labelling, only the nodes with a yellow shading would be labelled. With subtree-complete labelling, all the nodes with a blue shading would be labelled. The $\texttt{cell}(1,1,\texttt{b})$ expression nodes are labelled with both labelling functions. . . . . 101
5.6 Example vector embedding of two nodes of the instantiated rule graph from Figure 5.5 using subtree-complete labelling. . . . . 103
5.7 Architecture of Graph Neural Network used in the Neural Reasoner . 103
5.8 GAT Block design with bi-directional layers. . . . . 104
5.9 Training loss for parallelbuttonsandlights, tictactoe, connectfour and hanoi6disks. . . . . 112
5.10 Standard definition of rules in $\text{TTT}_L$ and flattened version of the same rules . . . . . 112
5.11 Rule graphs of unflattened and flattened fragment from Figure 5.10 . . 113
5.12 Comparison of minimal labelling and subtree-complete labelling during training for $\text{connectfour}_{3p}$ . . . . . 116
5.13 Comparison of random and sequential mixed training . . . . . 122
5.14 Principal component analysis of intermediate graph embedding generated by neural reasoner prior to output layers. . . . . 124

# Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.

Signature of candidate

# Publications

## Co-authored Work

All co-authors in the following table have approved these chapters for inclusion in this thesis.

| Chapter | Author % |
|---|---|
| Chapter 4: Exploring a Learning Architecture for General Game Playing.<br><br>Gunawan, A., Ruan, J., Thielscher, M., & Narayanan, A. (2020, November). Exploring a learning architecture for general game playing. In *Australasian Joint Conference on Artificial Intelligence* (pp. 294-306). Springer, Cham. | G = 80<br>R = 10<br>T = 5<br>N = 5 |
| Chapter 5: A Graph Neural Network Reasoner for Game Description Language.<br><br>Gunawan, A., Ruan, J., & Huang, X. (2022, August). A Graph Neural Network Reasoner for Game Description Language. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, (pp 443–452). doi:10.24963/kr.2022/46 | G = 80<br>R = 10<br>H = 10 |
| Alvaro Gunawan (G), Ji Ruan (R), Michael Thielscher (T), Ajit Narayanan (N), Xiaowei Huang (H) | |

We, the undersigned, hereby agree to the percentages of participation to the chapters identified above:

| Alvaro Gunawan | Ji Ruan | Michael Thielscher | Ajit Narayanan | Xiaowei Huang |
|---|---|---|---|---|

# Acknowledgements

First and foremost, I would like to thank my supervisors Dr. Ji Ruan and Prof. Ajit Narayanan for their guidance and assistance throughout the course of my entire academic journey. As lecturers in the very first undergraduate classes I attended, they instilled a passion and thirst for knowledge that became the initial spark for me to pursue further studies. Their support during my PhD studies has been invaluable, especially during the difficult times of the pandemic.

I would like to share my appreciation to my colleagues in the Centre for Artificial Intelligence Research, who were always open to discussing ideas and providing feedback: Peng Xia, Darrel Sadanand, Wenwang Pang, Jeff Feng and Tatsuki Hashimoto. Getting to know their personal interests and perspectives has been a joy, as well sharing the struggle that all postgraduate students must endure.

I would like to offer special thanks to Prof. Michael Thielscher and Prof. Xiaowei Huang for their contribution to our publications. Your expertise was an inspiration and your collaboration helped take what we were able to accomplish to greater heights. Many thanks to Bumjun Kim and the rest of the technicians at AUT, our experiments would not have been possible without your assistance. Your promptness and understanding whenever we needed technical help saved us countless times.

I would like to extend sincere gratitude to the Scholarship Office, the Vice Chancellor and AUT University for the Vice Chancellor's Doctoral Scholarship as it has provided the financial support necessary to complete this project.

I would like to thank my friends and family for their support throughout the years.

To my friends, thank you for your empathy when I was struggling with challenges during my research and for ensuring that I still could find time outside of my studies for enjoyment. To my family, thank you for your patience and support while I embarked on this journey. This is an opportunity that I have cherished dearly and would not have been possible if not for your belief in me.

Finally, I would like to give thanks to my partner Clairine. Your love and encouragement gave me the drive to push as far as possible along this journey. Your patience and care gave me the reassurance to keep going when hurdles seemed impossible to overcome. This thesis is as much as your work as it is mine, as it would have been impossible to complete without you by my side.

# Chapter 1

# Introduction

Game playing has been a staple of Artificial Intelligence research from the earliest days of computing up to the most recent advances in machine learning. Games provide a complex yet clearly defined environment for developing intelligent agents and the algorithms required to find optimal strategies to play them. Figure 1.1 shows a timeline of some key developments in computer game playing.

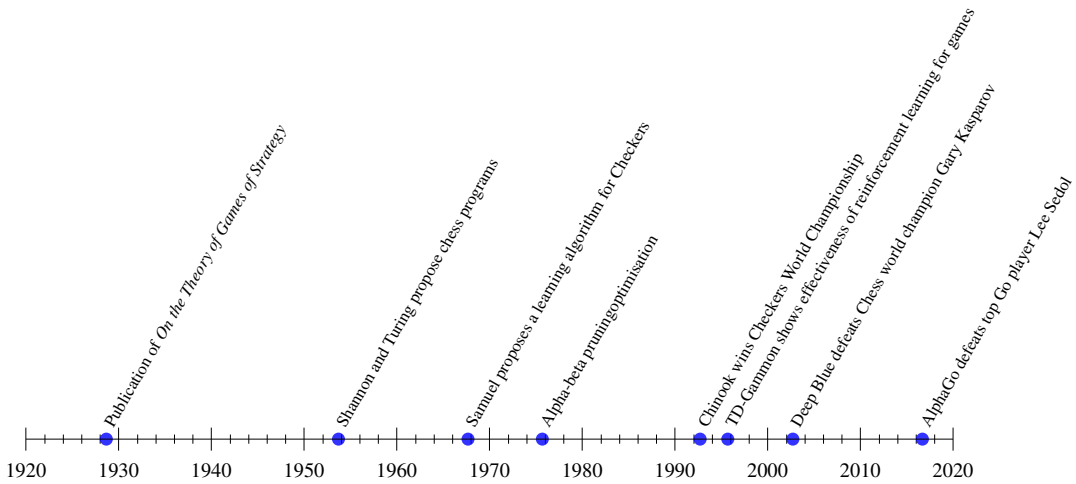



Figure 1.1: Timeline of developments in computer game playing

While the study of games stretch back far historically, the earliest foundations of the theoretical analysis and formalisation of games began in 1920 with the introduction of

the mathematical field of game theory (Neumann, 1928). Early developments until the 1950s in this vein were limited mostly to theoretical analysis due to the limitations in hardware (Shannon, 1950; Turing, 1953). With the rise in computing technology from the 1960s onwards, newly introduced algorithms such as Alpha-beta optimised minimax (Knuth & Moore, 1975) opened the door to several key developments in computer game playing. Some of the first reinforcement learning algorithms for game playing were proposed (Samuel, 1967; Tesauro, 1995), paving the way for learning-based game playing agents. This culminated in a series of world-firsts starting in the 1990s, with Chinook winning the Checkers World Championship (Schaeffer et al., 1992), Deep Blue defeating Chess world champion Gary Kasparov (Campbell, Hoane Jr & Hsu, 2002) and most recently AlphaGo defeating top Go professional player Lee Sedol (D. Silver et al., 2016).

General Game Playing (GGP) extends the challenge of standard computer game playing by requiring the agents to be able to play any game, given only a description of the game's rules. As the games are only provided in the form of a description of the rules, the agents must first infer the dynamics of a game with automated reasoning before it is able to find optimal strategies to play the game effectively. As a result, GGP encourages the development of agents with much more general reasoning abilities and strategy optimisation techniques compared to game-specific agents. GGP agents therefore are tasked with two main problems: (i) the inference of game rules using some reasoning techniques and (ii) strategy optimisation to maximise their outcomes in the games they play.

There has been recent interest in applying more advanced learning techniques to general game playing. For game playing, there has been a breadth of work in applying reinforcement learning techniques specifically to games (Tesauro, 1995; Gelly, Wang, Munos & Teytaud, 2006; Björnsson & Finnsson, 2009; Mnih et al., 2015). Games are a natural domain for reinforcement learning and many of the early works in reinforcement

learning focused on game playing (Samuel, 1967; Michie & Chambers, 1968). Deep learning based methods however have had limited applications in game playing for GGP (Emslie, 2017; Goldwaser & Thielscher, 2020), as the general nature has been a hurdle for neural network based approaches as each different game can be seen as a different domain.

On the contrary, applying learning techniques to the inference task is a more challenging task. While there have been attempts in the field of neurosymbolic artificial intelligence, the highly structured nature of formal logic tends to be a difficult paradigm for neural network based approaches. Firstly, the representations used for logical structures typically do not translate well to the vector-based representations that most neural networks use as input. This means some way of representing logical structures in a neural network appropriate manner is required. Secondly, inference tasks are typically formulated as a series of entailments, such as if $\phi \vDash \psi$, where $\phi$ may be some facts and rules (in the case of GGP, facts about the current state and rules of the game) and $\psi$ are the consequential facts that $\phi$ entails (in GGP, this may be the legal actions or if the state is terminal). Neural networks (and learning approaches in general) are typically stochastic in nature and thus this task must instead be formulated to consider the probabilities of the consequences instead. This can be done by instead estimating the probability that $\psi$ is true given $\phi$.

Recent advances in machine learning have introduced new methods that have shown major improvements in game playing. Deep reinforcement learning techniques have shown to be incredibly effective in AlphaGo, AlphaGoZero and AlphaZero (D. Silver et al., 2016; D. Silver, Schrittwieser et al., 2017; D. Silver, Hubert et al., 2017), with AlphaGo defeating top professional players in Go. AlphaGoZero extended the work of AlphaGo by learning from zero experience and AlphaZero extended this further by applying these techniques to Chess and Shogi as well. However, their approach

requires game-specific neural network architectures to be used and are limited to two-player, zero-sum, turn-taking games. These previous attempts at applying a deep reinforcement learning approach to game playing rely on having specialised hand-crafted neural networks designed and trained for specific games. For our implementation, several modifications must be made to the deep reinforcement learning algorithm to accommodate the more general nature of GGP. The neural network is automatically generated based on the game description provided to the agent, with the width of the input space set to be appropriate to the game and the number of outputs set according to the number of players and the action spaces. Furthermore, to accommodate the multi-player, general-sum games that can be encountered in GGP, the MCTS algorithm is modified to accommodate these games.

The inference of game rules in GGP would typically use automated reasoning techniques such as resolution and its derivatives. Developments in neural network design have introduced architectures that are able to infer over novel representations better suited for more structured data. For this reason, we present *instantiated rule graphs*, a novel graph-based representation of states that combine the graph structure present in game descriptions and the information present in the state. We have implemented a graph neural network based neural reasoner, taking advantage of relational inductive biases present in the logical structure of the game descriptions. This neural reasoner is able to learn to infer games in GGP either individually or with multiple games in a mixed dataset. It is also able to transfer learned inferences across similar games without additional training.

While it might be tempting to combine both implementations directly, separating the tasks allows for more exploration into each individual component. As the game-playing methods consist of many inter-connected components, it would be difficult to diagnose with the added complexity of a neural reasoner. We instead wish to present these two new methods as base components to further develop more complete learning

architectures for GGP and to foster the development of agents that are able to transfer learned general knowledge across various domains.

Our main contributions are in two folds: Firstly, we present GGPDeepRL, an AlphaZero-style GGP agent that utilises deep reinforcement learning to learn to play general games. This agent adapts the deep reinforcement learning algorithm originally presented in AlphaZero to accommodate the general games of GGP as they can be multi-player, general-sum and simultaneous action. It is able to overcome the game-specific limitations of AlphaZero as it constructs its neural network based on the game description provided. We explore this learning architecture and examine its behaviour with varying parameters and evaluate it on a variety of games with a wide array of game-theoretic features. We investigate the effect of neural network depth on overall performance, compare it with a convolutional neural network version and analyse the effect of simulation and time limited MCTS. Our results confirm the feasibility of applying deep reinforcement learning to GGP but note certain limitations with regards to the neural network architecture used.

Secondly, we present a graph neural network based *neural reasoner*, utilising novel graph-based representations of the game rules to learn to approximate the inference task. The neural reasoner uses *instantiated rule graphs* as input, a general, game-agnostic graph-based representation of game states described in GDL. Instantiated rule graphs allow for a single neural reasoner to be trained to infer over states across multiple games, as the representation is capable of capturing game state features while still being consistent across different games. We provide a method to generate training samples for the neural reasoner using games described in GDL as instantiated rule graphs. We investigate the effect of different labelling functions for the instantiated rule graph as well as varying graph neural network architectures and parameters on reasoning accuracy. We also examine the issues caused by long-range dependencies present in the graph representation. The neural reasoner is capable of accurately inferring legal

actions and subsequent states as well as transfer these learned inferences across different games.

We first provide a review of the relevant literature on deep learning and game playing. We then introduce the key background information that acts as the base of our work. The subsequent chapters will present and discuss the experiments on GGPDeepRL and the neural reasoner respectively. The chapters of this thesis are structured as follows:

Chapter 2 will provide a historical perspective and a comprehensive review of the literature on the intersection of learning methods and game playing. Section 2.2 provides a brief history of the developments of neural networks and their applications across various domains. We provide a more detailed discussion on the application of neural networks specifically in the logical domain. Section 2.3 reviews the history of reinforcement learning with a specific focus on how it has impacted the development of computer game playing. Finally, Section 2.4 provides an overview of the field of General Game Playing, highlighting the development of various techniques used by tournament-winning agents. We also review the various methods in which games are represented for broader general game playing settings.

Chapter 3 introduces the preliminary concepts that form the background of our work. Section 3.2 presents an overview of General Game Playing and its problem specifications. Section 3.3 introduces Game Description Language, the logical programming language used to represent the rules of the games used in General Game Playing. Section 3.4 provides technical details on the General Game Playing environment as well as the various game playing methods used by GGP agents to generate effective strategies. Finally, Section 3.5 describes the deep learning techniques such as deep neural networks and deep reinforcement learning used in game playing.

Chapter 4 explores an implementation of an AlphaZero-style deep reinforcement learning agent for GGP. The agent is evaluated on several games and with varying architecture parameters to investigate its capabilities for GGP. Section 4.2 describes

the architecture of our self-play reinforcement learning agent, GGPDeepRL . In the extensive Section 4.3 we present the results of our experiments, showing the agent's capabilities in a variety of games as well as an in-depth examination in the agent's architecture. Finally, Section 4.3 concludes the chapter by discussing the key observations from the results and the limitations of the approach.

Chapter 5 presents the graph neural network based reasoner for GDL. We introduce a novel graph-based game state representation used by the neural reasoner and the architecture used to learn the inference on these representations. Section 5.2 gives an overview of the different components of our neural reasoner. Section 5.3 describes a general representation for game states in GGP, and in particular we propose *instantiated rule graphs*. Section 5.4 introduces our proposed GNN architecture that is able to handle the instantiated rule graphs and outputs the predicted legal actions and next fluents. Section 5.5 presents and discusses the results of our neural reasoner across a variety of games, a mixture of games and transferring across different games. Finally, Section 5.6 concludes by summarising the results of our experiments and discusses possible future directions for research.

Finally, Chapter 6 will conclude with a discussion on the limitations of our approaches and possible future work.

# Chapter 2

# Literature Review

## 2.1 Introduction

The pursuit of developing intelligent agents capable of playing games spans the history of artificial intelligence research, game theory, computer science and mathematics. Game theory was first established as a separate field of study with von Neumann's publication of *On the Theory of Games of Strategy* (Neumann, 1928), focused on the formalisation and analysis of games in which multiple players must interact with each other. Players in these kinds of games typically find that their payoffs (or utilities) are not only tied to their own actions, but the joint actions of all players in the game. Finding the exact behaviour of these types of games was the driving force behind work in game theory, with game theoretical analysis having direct applications to fields such as mathematical economics. As part of this analysis, von Neumann also introduced some of the earliest game playing algorithms by proving the minimax theorem.

One of the most important aspects of game theory is the analysis of the player's behaviours given the utilities and actions available to them in a particular game. Game theoretical analysis assumes that agents are rational and that they would take actions or use strategies that maximise their own payoff. This type of analysis of games - searching

for strategies in which each player maximises their utilities - led to the development of solution concepts. Most famously is the Nash equilibrium

Game theoretical analysis initially focused on the *normal form* representation of games, in which a matrix represented the possible interactions of the players. This matrix representation allows for simpler calculations of solution concepts as they effectively model a "one-shot" game, in which each player selects a single strategy and the utilities are the resulting intersection of all the chosen strategies. In contrast, most of the commonly played games such as Chess and Go exhibit sequential behaviour with multiple states and decision points for each player. For these types of games, an *extensive form* representation is more natural, in which the sequence of states and actions are explicitly defined. The most commonly used formulation of extensive form games was first introduced by Kuhn (Kuhn, 1950) as an extension on von Neumann's work and has informed the development of *game tree* representations that are utilised by game playing algorithms today. Most extensive form representations of games have an equivalent normal form representation and as such the solution concepts developed for normal form games can also be applied to extensive form games.

With the rise of computing technology in the 1940s and 1950s, the possibility of developing algorithms for computers to play complex games was becoming a reality. Shannon (Shannon, 1950) and Turing (Turing, 1953) both independently described programs that would allow a computer to play Chess using game-theoretic notions such as the minimax principle. The hardware available at the time limited the capabilities of Chess-playing computers, but still encouraged an active field of research.

As the field of computer science began to flourish, Chess among other games were popular test beds for the development of various algorithms. This eventually lead to further improvements to game-playing algorithms such as alpha-beta pruning (Knuth & Moore, 1975). Together with the rapidly increasing capabilities of computer hardware, computer game playing was able to reach a level in which it could compete with top

players of their respective games. Beginning in the 1990s with Chinook winning the Checkers World Championship (Schaeffer et al., 1992) and Deep Blue defeating Chess world champion Gary Kasparov (Campbell et al., 2002). To this day, many of the best Chess engines such as Stockfish (The Stockfish developers (see AUTHORS file), n.d.) still utilise some form of minimax search, taking advantage of advances in hardware, search efficiency and estimation techniques over the years.

While minimax search achieved many successes in games such as Chess, such approaches had difficulties in other games such as Go. In these cases, minimax search suffered from two main problems: Firstly, the evaluation functions for these games can not be as clearly defined or lack enough expert information to define them. For General Game Playing, it is even more difficult as the agents must be able to evaluate the states of arbitrary games with minimal information as to which games are being played. Secondly, even with the optimisations of alpha-beta pruning, the symmetric expansion of the game tree to a specified depth means the agents are less scalable to games with very large action and state spaces. Inspired by Monte-Carlo methods, the use of stochastic sampling for game playing was first proposed as an alternative way to estimate the values of game states (Abramson, 1987) without requiring a static evaluation function. This was then further extended by asymmetrically expanding the game tree based on a selection strategy that balances exploration and exploitation, giving birth the the class of search methods known as Monte-Carlo Tree Search (Coulom, 2006). Monte-Carlo Tree Search based agents were the first to achieve master-level performance on smaller versions of Go (Lee et al., 2009), which historically was not possible with minimax-based agents. However, pure Monte-Carlo Tree Search based agents were still unable to challenge to professional Go players in the full 19×19 version of Go. This milestone was finally achieved with AlphaGo's victory over top professional Go players (D. Silver et al., 2016), utilising a combination of Monte-Carlo Tree Search and deep neural networks, learning through *deep reinforcement learning*. The methods

proposed in AlphaGo also found success in Chess and Shogi (D. Silver, Schrittwieser et al., 2017), proving that this approach is general to other complex two-player zero-sum games.

In this chapter, we will review the key developments that have lead to the techniques used in game playing agents today. Section 2.2 covers the development of neural networks and their use in deep learning. Section 2.3 reviews the history of reinforcement learning and the various algorithms developed for use in game playing. Finally, Section 2.4 discusses the field of General Game Playing and how it extends earlier work on computer game playing.

## 2.2 Neural Networks

Neural networks have become ubiquitous in machine learning and beyond, with the many types of networks and architectures used in almost every field and industry. They began as biological and mathematical models of the nervous system in animals (McCulloch & Pitts, 1943; Rosenblatt, 1958), with early models either forgoing any sort of learning method or implementing a very simple algorithm to update the weights. The development of algorithms for computing gradients using a backpropagation procedure (Werbos, 1974) allowed for the efficient, end-to-end training of a neural network and opened the door to more complex and *deeper* neural networks. These architectures are known as feed-forward fully-connected neural networks and are still in use today, either as simpler architectures on their own or as a component in a larger neural network architecture.

During this time, the convolutional neural network (LeCun et al., 1989) was first introduced and showed the potential of deep neural networks for many different supervised learning tasks. Research into applying these deep neural network architectures for different applications began to crystallise into the field we know today as *deep learning*.

However, work in this field began to slow in the 1990s as neural networks at the time suffered from poor performance on more complex tasks. The algorithms used by these networks would lead them to poor local minima during training and the networks did not have the capacity to escape them. As graphics processing units (GPUs) became increasingly popular both commercially and in research (Raina, Madhavan & Ng, 2009), neural networks could be made larger and trained faster. Both the increase in size and speed eventually allowed neural networks to overcome the limitations faced before.

Today, deep learning has become the predominant machine learning technique used across many fields and has achieved state-of-the-art results across many domains (LeCun, Bengio & Hinton, 2015; Schmidhuber, 2015). CNNs in particular have recently achieved notable performance in many tasks, such as image classification (Krizhevsky, Sutskever & Hinton, 2017) and Atari game playing (Mnih et al., 2015). Neural networks with convolution-style operators have become the de-facto architectures for solving most tasks in which the inputs can be effectively represented by a matrix or grid-like representation. More recently, attention-based architectures (Vaswani et al., 2017) such as transformer networks have shown remarkable accuracy in many NLP tasks (Brown et al., 2020). Attention mechanisms have shown to be very effective in sparse representations, a normally difficult domain for other neural network architectures.

### 2.2.1 NNs for Logic

Work in the 90s and early 2000s on the modelling of logic programming using neural networks (A. S. Garcez Artur S d'Avila & Zaverucha, 1999; Abdullah, 1992; A. S. d. Garcez, Broda, Gabbay et al., 2002) laid the foundations of neuro-symbolic computing - the synthesis of logic and neural networks. Early work in the field rely on directly translating the logical statements to the neural network structure itself.

Recently, there has been a resurgence in applying neural network based methods

to logical domains, following their successes in computer vision and natural language processing. This more recent work focuses on using the logical statements themselves as input to the neural networks, allowing for more general neural network architectures. However, the structured nature of logic tends to be a difficulty for neural networks, which rely heavily on statistical features. Evans et al. introduced a dataset of logical entailment examples to train and evaluate neural network architectures on (Evans, Saxton, Amos, Kohli & Grefenstette, 2018), as well as their model PossibleWorldNet which was able to outperform several benchmarks. Rawson and Reger (Rawson & Reger, 2020) were able to further extend this by taking advantage of the relational inductive bias present in the logical dataset. Their work presented a graph neural network-based architecture and a graph based encoding for the logical statements, which combined was able to outperform PossibleWorldNet in several of the categories of the dataset. Beyond the two mentioned earlier, many other researchers have applied graph neural networks to the logical domain (Thost & Chen, 2021; Abdelaziz et al., 2021; Glorot et al., 2019; Paliwal, Loos, Rabe, Bansal & Szegedy, 2020; Crouse et al., 2019; Olšák, Kaliszyk & Urban, 2019).

Some recent work such as (T. Silver et al., 2020; Lin, Wang, Undersander & Rai, 2022) has applied graph neural networks to planning problems. However, most work in this field typically use graphical representations of the objects in the environment and their relationships instead of directly modelling a logical description. As such, the GNNs are tasked to learn the importance and probabilities of the objects themselves, rather than learning to infer the logical structure of the environment itself.

## 2.3 Reinforcement Learning for Games

In contrast to the supervised learning that has dominated most of machine learning research, reinforcement learning takes a different approach that is better suited to the

interactive environments of games. Reinforcement learning has had a long history, eventually culminating in the early 1980s into what we understand as reinforcement learning today. According to Sutton and Barto (Sutton & Barto, 2018), there were three main threads of research that were predecessors to modern reinforcement learning. The first thread focused on "learning by trial-and-error", which was initially based on the psychology of learning found in animals. The second thread tackled the problem of optimal control, which was typically solved using techniques such as dynamic programming. The third and final thread was the development of temporal-difference learning, a key concept that has inspired many of the modern reinforcement learning techniques that are used today. Together, these three threads laid the groundwork for the field of reinforcement learning and various algorithms used in artificial intelligence research today.

The following highlights several of the key developments throughout the history of reinforcement learning research. A key concept in reinforcement learning is the trade-off between exploration and exploitation, where an agent must choose between exploring more of the action space to learn additional information about the environment, or to exploit already learned information to maximise its current estimated rewards. One of the earliest specifications of this problem was presented by Bellman (Bellman, 1956) and is today commonly known as the "multi-armed bandit" problem. A solution to such problems based on dynamic programming was proposed and introduced the *Bellman equation* (Bellman, 1966) as a way to define a function for solving optimal control problems. To model discrete stochastic versions of these optimal control problems, Bellman also introduced the Markov decision process (MDP) (Bellman, 1957). These concepts introduced for optimal control problems has had wide-reaching impact on reinforcement learning tasks and approaches to this day, with Bellman equations being a core element in many reinforcement learning algorithms and MDPs being the base of many approaches of modelling interactive environments.

What we consider to be modern reinforcement learning began to take shape with the introduction of a class of Monte Carlo methods (first collectively described in 1998 (Sutton & Barto, 2018)) which uses the agent's raw experience without the need to fully solve a complete model of the environment. Combining these concepts with those from dynamic programming, *temporal-difference* (TD) learning was developed (Sutton, 1988). An off-policy version of this approach called *Q-learning* was also introduced (Watkins, 1989) shortly after. These TD approaches were used to great effect with the development of an effective Backgammon playing agent, TD-gammon (Tesauro, 1995). This was arguably the first competitive game-playing agent that utilised reinforcement learning as a core element. An important precursor to TD-Gammon was Samuel's checkers player (Samuel, 1967), as it introduced several key principles that are used in game-playing TD algorithms.

Both Monte Carlo and TD based methods are typically classified as tabular methods as they directly store the learned estimations for each state in a table-like structure. A more powerful and general way of storing such learned estimations would be to use a parametrised functional form instead - framing the reinforcement learning problem as a function approximation task instead (Bertsekas & Tsitsiklis, 1996; Bertsekas, 2012; Sugiyama, 2015). These approaches have grown in popularity recently with the rise of artificial neural networks as nonlinear function approximators.

The previously mentioned approaches attempt to learn action values which it then bases a policy on. Policy gradient methods instead directly learn a policy (Sutton, McAllester, Singh & Mansour, 1999), such as the actor-critic methods (Degris, White & Sutton, 2012) commonly used today.

A notable development in reinforcement learning was the introduction of Monte-Carlo Tree Search (MCTS) (Coulom, 2006; Kocsis & Szepesvári, 2006), which applied the principles used in Monte Carlo reinforcement learning methods directly into a game tree search algorithm. MCTS uses simulated trajectories by executing randomised

playouts until terminal states to efficiently accumulate value estimates in a stochastic manner. Additionally, by using an upper confidence bound selection strategy, MCTS can direct these simulations towards more relevant areas of the game tree. MCTS-based agents were able to achieve strong performance against classical Go programs (Gelly et al., 2006; Coulom, 2006, 2007) and was used extensively by tournament-winning GGP agents (Genesereth & Björnsson, 2013).

Some of the most notable recent work applying reinforcement learning based approaches to games is that of AlphaGo, AlphaGo Zero and AlphaZero (D. Silver et al., 2016; D. Silver, Schrittwieser et al., 2017; D. Silver, Hubert et al., 2017), which were able to achieve super-human performance on Go, Chess and Shogi. The approach presented by these agents was named *deep reinforcement learning*, as the agents would utilise deep neural networks as a key element of their reinforcement learning algorithm. The deep neural networks were used for policy and value estimation and were trained through self-play, a process in which the agent plays against a version of itself to explore the state space using the strategies it had already learned. Although the architecture presented in AlphaZero was general across the three games, the networks were specific for each game and had to be trained separately due to the varying state representations.

## 2.4 General Game Playing

General Game Playing (Genesereth & Thielscher, 2014) was first proposed to foster the development of algorithms that can be applied generally across different games (Genesereth, Love & Pell, 2005), with the goal of achieving artificial intelligence with more general capabilities. The specification for Game Description Language (Love, Hinrichs, Haley, Schkufza & Genesereth, 2008), the formal language used to encode games in GGP, was introduced as a standard representation for game descriptions. The adoption of GDL as a standardised syntax for game descriptions led to the development

of multiple GGP agents and culminated in several GGP competitions run over the years (Genesereth & Björnsson, 2013).

As can be seen in Table 2.1, the approaches used by the winning players have changed throughout the years. Notably, the search methods used parallel the developments that occurred in computer game playing and reinforcement learning more broadly, as MCTS became the dominant algorithm used by most players. With MCTS requiring efficient reasoning for the playout and simulation phases, players using the more efficient propositional network reasoners were more successful. One notable exception was Woodstock, which used a drastically different approach compared to the other agents, using techniques from stochastic constraint programming. Another important aspect to note is that the players have not taken advantage of the recent developments in deep learning, a strong motivator for the investigations that we have conducted.

More recently, there has been several new works in GGP outside the initial Stanford specification. Extensions to GDL introduce additional capability: GDL-II (Thielscher, 2010) adds the capability of representing games that include randomness and imperfect information, GDL-III (Thielscher, 2017) extends this further by adding additional operators to represent epistemic features in games. Various strategies have been developed to reason about both GDL-II and GDL-III games as they have additional features on top of the standard propositional logic of GDL (Huang, Ruan & Thielscher, 2013; Schiffel & Thielscher, 2014; Ruan & Thielscher, 2012, 2014; Engesser, Mattmüller, Nebel & Thielscher, 2021; Ruan & Thielscher, 2011). Other works also extend the existing definition GDL by endowing it with additional operators and semantics to model additional features of game playing (Zhang & Thielscher, 2015; Haufe, Schiffel & Thielscher, 2012; Jiang, Zhang, Perrussel & Zhang, 2016).

Beyond the direct extensions to GDL, other specifications have been developed to facilitate other facets of general game playing. Inspired by the work on learning to

| Year | Player | Search method | Reasoner | Heuristics and feature detection |
|---|---|---|---|---|
| 2005 | Cluneplayer (Clune, 2007) | Iterative-deepening minimax with alpha-beta pruning, transpotion tables and aspiration windows | No information provided | Abstract models with stable features for compound lottery heuristic evaluation function |
| 2006 | Fluxplayer (Schiffel & Thielscher, 2007) | Iterative-deepening minimax with alpha-beta pruning, transposition tables, history heuristics | Prolog-based Flux system | Fuzzy logic base heuristic function with semantic structure detection |
| 2007, 2008 & 2012 | Cadiaplayer (Björnsson & Finnsson, 2009) | Monte-Carlo Tree Search | Prolog-based reasoner | Opponent modelling and playout optimisation |
| 2009 & 2010 | Ary (Méhat & Cazenave, 2010) | Monte-Carlo Tree Search | Prolog-based reasoner | Nested Monte-Carlo and transposition tables |
| 2011 & 2013 | TurboTurtle (Schreiber & Landau, 2013) | Monte-Carlo Tree Search | Propositional networks | No information provided |
| 2014 | Sancho (Draper & Rose, 2016) | Monte-Carlo Tree Search on generalised game graph | Propositional networks | Bounded node structure, active trimming, setup-time heuristic discovery and static analysis for game factorisation |
| 2015 | Galvanise (Emslie, 2017) | Monte-Carlo Tree Search | Propositional networks | No information provided |
| 2016 | Woodstock (Koriche, Lagrue, Piette & Tabary, 2017) | Stochastic constraint network with Maintaining-Arc Consistency and stochastic sampling | N/A | Constraint-based symmetry detection |

Table 2.1: Winners of the annual AAAI GGP Competition

play Atari 2600 games through the Arcade Learning Environment (Bellemare, Naddaf, Veness & Bowling, 2013), General Video Game Playing (GVGP) (Levine et al., 2013) was proposed as an extension to GGP, GVGP requires agents to play an arcade-style video game it has not seen before. To facilitate GVGP, Video Game Description Language (VGDL) (Ebner et al., 2013; Schaul, 2013) was developed as a high-level description language for representing arcade-style 2D video games.

Due to the relative inefficiency of reasoning with GDL, several alternative representations have been proposed. Regular Boardgames (RBG) (Kowalski, Mika, Sutowicz & Szykuła, 2019) is an alternative to GDL that is based on the theory of regular languages. RBG is more efficient than GDL when representing games with board-like states, but is limited only to alternating-move games. Ludii (Piette et al., 2019) encodes general games using their component *ludemes*, an atomic, conceptual unit of game-related information that allows for the representation of key game concepts. Like RBG, Ludii shows improved efficiency over reasoning with GDL and is competitive with RBG in most games. Unlike in GDL where key structural aspects of games such as the board or game pieces must be defined explicitly from scratch, games are constructed using the higher-level game concepts represented by the ludemes.

Building upon the work of GGP, an inverted version of the inference task was proposed: given traces of a given game, can a system learn and infer the rules that could produce the traces? This challenge was named *inductive general game playing* (IGGP) (Cropper, Evans & Law, 2020) and recent investigations have found that existing inductive logic programming approaches struggle in this new domain. While IGGP is not a focus of our work, there is a possibility to extend our approaches to this domain, especially the general state representations utilising graph structures.

Research in applying reinforcement learning based approaches to GGP began with the framework RL-GGP (Benacloch-Ayuso, 2012) that integrates a GGP server and a standard framework for reinforcement learning RL-Glue. QM-learning (Wang,

Emmerich & Plaat, 2018) combines Q-learning with MCTS, and integrates it within GGP. The performance of the agent is evaluated on three small board games. However, this method is still effectively only an on-line search method, using Q-learning to optimise MCTS. The recent GGP agent (Emslie, 2017) implements an AlphaZero-style learning component that exhibits strong performance in several games, but requiring hand-crafted convolutional neural networks for these games. Like AlphaZero, this means that the agent requires human intervention before it is able to play a new game.

Generalised AlphaZero (Goldwaser & Thielscher, 2020) implements a Deep Reinforcement Learning framework for GGP in which a fully-connected neural network is generated for any given game description. This allows Generalsied AlphaZero to learn to play any game itself based only on game rules without human-crafted neural networks. The trained agents show strong performance is several two-player, turn-taking, zero-sum games but limited performance on cooperative games. This work is the closest in approach to our work as we also utilise a fully-connected neural network for our agent, but differs in type of representation used. Generalised AlphaZero uses a list of components in the propnet of the game, while our approach uses a fluent list. Furthermore, we perform additional experiments on a larger collection of games to further verify our implementation. Additional experimentation with Generalised AlphaZero showed the possibility of transferring learned knowledge by copying weights of a prior network into a new network for a different game (McEwan & Thielscher, 2022).

As mentioned earlier, many of the top GGP agents utilised some sort of heuristic that makes use of general features present in the game. This is usually done by extracting some kind of feature information from the game description itself. The logical representation of GDL lends itself to certain structures that can be found by transforming the syntax into some other representation. One of these alternative representations is the *rule graph*, which transforms the description into a graph consisting of a syntax tree and label nodes and edges to retain information on predicate and variable identifiers.

Rule graphs were first proposed by Kuhlmann and Stone (Kuhlmann & Stone, 2007), where they have investigated their use for transfer learning in GGP. In their work rule graphs are used to identify similar games, to which they transfer known value mappings for Q-learning. Schiffel (Schiffel, 2010) further extends this work and uses rule graphs for symmetry detection within games. As these make use of standard rule graphs, they do not apply their methods to individual game states. Additionally, they do not make use of graph neural networks, instead using more traditional approaches such as graph isomorphism algorithms to detect similarity between rule graphs.

## 2.5 Conclusion

We have provided an overview of the history of computer game playing and its impact on the study of artificial intelligence more broadly. A review of neural network techniques and its successful applications across various fields has been presented, with a specific focus on the use of neural networks on logical domains. A comprehensive summary of the development of reinforcement learning has been presented. Finally, we discuss the current literature on GGP and how deep learning and reinforcement learning have been used by GGP agents.

Our exploration of the literature and past work highlights the limited application of certain deep learning and reinforcement learning techniques to GGP despite being successful in very similar domains. While there have been attempts at implementing deep reinforcement learning agents for GGP, they tend to either forgo more generality for better performance or have relatively limited success on the wider range of games available. Furthermore, we find that these approaches only focus on applying learning to the *playing* aspect of GGP and rely on standard logical approaches for the GDL reasoners. The recent developments of graph neural networks have in particular provided a potential avenue for applying deep learning techniques for the much more structured

logical domain. As such, our work will focus on investigating these particular issues to see if these recent developments allows for new approaches to learning for GGP.

# Chapter 3

# Background

## 3.1 Introduction

As seen during the review of the literature, there has been many developments that have led to the possibility of applying learning techniques to both tasks of GGP. Therefore, there is some background that must be detailed prior to our implementation to facilitate a better understanding of the capabilities and behaviour of a learning based system.

Due to the fact that GGP and GDL act as a kind of standard for agents to follow, it is necessary to discuss several well-established definitions and features with a specific focus on their relevance for learning architectures. Additionally, while it is possible to develop a game playing agent or reasoner from scratch, many of the most successful attempts build on techniques and algorithms that have been proven to be effective in the past. As such, we must discuss some of these prior techniques and the adaptations of these commonly used algorithms in the context of GGP.

This chapter will introduce the key preliminary concepts that we base our work on. In Section 3.2, we provide an overview of the General Game Playing setting and its problem specifications. Section 3.3 introduces Game Description Language and its features. Section 3.4 provides technical detail on the General Game Playing

environment as well as the methods used by agents to play general games. Finally, Section 3.5 describes the key deep learning techniques used in game playing.

## 3.2 General Game Playing

General Game Playing (GGP) is an extension of traditional computer game playing, requiring agents to play arbitrary, general games given only the description of game rules. This models a more human-like approach to game playing, as humans typically approach a game by first learning the rules of the game from a natural language description of the game. To facilitate the implementation of algorithms for general game playing, a formalisation of game rules is used instead of a natural language representation. For our work, we follow the original Stanford specification of GGP (Love et al., 2008).

GGP considers perfect information games with an arbitrary number of players that are finite, discrete and deterministic. Games in GGP have a finite number of possible states with one state defined as the initial state and at least one state defined as a terminal state. Each game has a fixed, finite number of players with a discrete, finite number of actions possible for each player. Terminal states assign a utility value for each player. At each state, all players must select a legal action to play. A state transition is defined by the current state, the actions selected by each player and the game rules.

In standard computer game playing, the agents typically use a game-specific program that allows them to get the legal actions, subsequent states and resulting terminal states and their utilities. This allows the game agent to explore the game tree and generate the strategy that the agent uses during play. The algorithms used to generate these strategies are also typically game-specific, with heuristics that exploit features present in the specific game the agent is designed to play.

In contrast, the GGP setting only provides a description of the game and the agents must infer the rules of the game themselves. Although this adds an additional layer of

complexity, this also allows a single agent to play any game, as long as the description is provided to them. As this inference drives the agent's ability to explore and expand the game tree, the efficiency of the reasoning techniques can directly impact the effectiveness of the algorithms used to generate the strategies used by the agent. For example, a more efficient reasoner would be able to expand a larger game tree more quickly, allowing a search-based algorithm to evaluate more game states. Additionally, as the agent is tasked with playing general games, the algorithms that are used to generate strategies for play must also be general. Game-specific heuristics or optimisations cannot be used, unless the agent is able to detect which game is being played. It is more effective to instead take advantage of more general game features such as symmetry and composability.

In summary, a GGP agent is tasked with the two following problems:

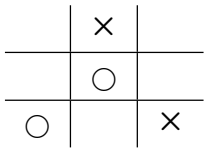
- *Reasoning*: efficiently infer the rules and dynamics of a general game from its description.
- *Play*: generate an effective strategy to maximise the agent's utility across general games.

## 3.3 Game Description Language

As GGP requires the agents to infer the games based only on a description of the game's rules, a method of encoding the rules is required.

Game Description Language (GDL) is a formal language used to define the rules of a game, representing the dynamics and features of the game and allowing for efficient reasoning and inference. GDL is a declarative logical programming language based on Datalog with syntax and semantics specifically defined for its use in GGP.

A state in GDL is defined by a set of dynamic predicates known as *fluents*, which



$$\left\{\begin{matrix} cell(1,1,o), cell(1,2,b), cell(1,3,x), \\ cell(2,1,b), cell(2,2,o), cell(2,3,b), \\ cell(3,1,b), cell(3,2,x), cell(3,3,b), \\ control(xplayer) \end{matrix}\right\}$$

Figure 3.1: A Tic-Tac-Toe state and its corresponding state in GDL.

represent certain facts that are present in the given state. Figure 3.1 shows an example of a state in Tic-Tac-Toe and how it is represented as a state in GDL.

Several key relations or keywords are defined in GDL to facilitate its use as an encoding of games: $role, init, true, does, next, legal, terminal, goal$ and $distinct$. Excluding $distinct$, each of these relations correspond directly to mechanisms necessary to define the dynamics of a game. The $role$ keyword defines the different roles the players can take in the game as well as the total number of players in the game (one player per role). The $init$ keyword defines the fluents that are true in the initial state. The $true$ keyword checks if a certain fluent is present in the current state and $does$ checks if a certain action is taken by a specified player in the current state. The $next$ keyword defines which fluents are present and true in the next state. The $legal$ keyword defines if an action is legal to be taken for a specified player in the current state. The $terminal$ keyword checks if the current state is a terminal state and the $goal$ keyword returns the utility value of the specified player for the current state.

Optionally, the $base$ and $input$ keywords are sometimes supported by certain GGP environments, but are not in the default specification of GDL. The $base$ keyword defines all the fluents that can appear in the game, while the $input$ keyword defines all the possible actions that can be taken by the players. These keywords are not technically required, as the possible fluents and actions will be generated as a consequence of inferring via the $next$ and $legal$ keywords. The $distinct$ keyword mentioned earlier checks if two terms are syntactically different. While technically also optional, without

the *distinct* keyword, requiring each game description to define relations for every unequal term would be cumbersome and as such is built into the standard definition of GDL. The base ten representations of integers $0 \ldots 100$ are also defined in GDL, but are only semantically tied to their numerical values in the context of the *goal* keyword. When used elsewhere, these integers are simply treated as characters for predicate and variable identifiers.

Figure 3.2 shows a description of Tic-Tac-Toe written in GDL. As can be seen, several of the keywords mentioned earlier are used to define the dynamics of Tic-Tac-Toe. Lines 1 and 2 use the *role* keyword to define the two players: `xplayer` and `oplayer`. Lines 4-13 define the initial state using the *init* keyword - all cells begin blank and `xplayer` has control during the initial state.

Lines 15-47 contain the rules for state transitions by using the *next* keyword to define which fluents are true in subsequent states. The two rules between lines 15-21 specify if a `cell` fluent with `x` or `o` is true in the next state, given that the cell was blank in the current state and the `mark` action was taken in that cell's coordinates. The rule in lines 23-25 retain unchanged `cell` fluents from state to state if they are not blank. Similarly, the rule in lines 27-31 retain blank `cell` fluents given that the `mark` action was not on the given cell. Lines 33-37 contain rules that define the turn-taking mechanism by alternating the `control` fluent between `xplayer` and `oplayer`.

Lines 39-64 contain rules that define the predicates `row`, `column`, `diagonal`, `line` and `open`. These are non-keyword user-defined predicates that are used to facilitate the definition of `terminal` and `goal` rules.

Lines 66-74 contain the rules that define the legal actions. The rule in lines 66 to 68 declare that it is legal for a player to `mark` a cell given that the cell is blank and that player is in control. For the player that is not in control, lines 70 to 74 contain rules that define `noop`, the only action the non-active player can take.

Using the previously defined `line` and `open` predicates, the rules in lines 76-96

```
(role xplayer)
(role oplayer)

(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 3 1 b))
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (control xplayer))

(<= (next (cell ?m ?n x))
(does xplayer (mark ?m ?n))
(true (cell ?m ?n b)))

(<= (next (cell ?m ?n o))
(does oplayer (mark ?m ?n))
(true (cell ?m ?n b)))

(<= (next (cell ?m ?n ?w))
(true (cell ?m ?n ?w))
(distinct ?w b))

(<= (next (cell ?m ?n b))
(does ?w (mark ?j ?k))
(true (cell ?m ?n b))
(or (distinct ?m ?j)
    (distinct ?n ?k)))

(<= (next (control xplayer))
(true (control oplayer)))

(<= (next (control oplayer))
(true (control xplayer)))

(<= (row ?m ?x)
(true (cell ?m 1 ?x))
(true (cell ?m 2 ?x))
(true (cell ?m 3 ?x)))

(<= (column ?n ?x)
(true (cell 1 ?n ?x))
(true (cell 2 ?n ?x))
(true (cell 3 ?n ?x)))

(<= (diagonal ?x)
(true (cell 1 1 ?x))
(true (cell 2 2 ?x))
(true (cell 3 3 ?x)))

(<= (diagonal ?x)
(true (cell 1 3 ?x))
(true (cell 2 2 ?x))
(true (cell 3 1 ?x)))

(<= (line ?x) (row ?m ?x))
(<= (line ?x) (column ?m ?x))
(<= (line ?x) (diagonal ?x))

(<= open
(true (cell ?m ?n b)))

(<= (legal ?w (mark ?x ?y))
(true (cell ?x ?y b))
(true (control ?w)))

(<= (legal xplayer noop)
(true (control oplayer)))

(<= (legal oplayer noop)
(true (control xplayer)))

(<= (goal xplayer 100)
(line x))

(<= (goal xplayer 50)
(not (line x))
(not (line o))
(not open))

(<= (goal xplayer 0)
(line o))

(<= (goal oplayer 100)
(line o))

(<= (goal oplayer 50)
(not (line x))
(not (line o))
(not open))

(<= (goal oplayer 0)
(line x))

(<= terminal
(line x))

(<= terminal
(line o))

(<= terminal
(not open))
```

Figure 3.2: Tic-Tac-Toe description in GDL. Retrieved from `http://games.ggp.org/base/games/ticTacToe/ticTacToe.kif`

define the goal values for each player given the condition of the terminal state. Finally, lines 98-105 define the rules for `terminal` states, also using the `line` and `open` predicates as conditions.

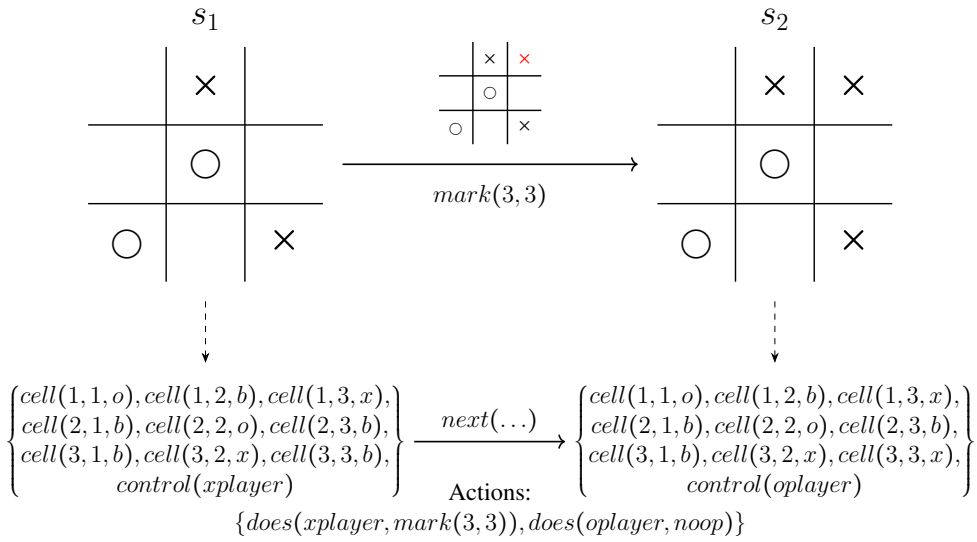



Figure 3.3: State trasition of a Tic-Tac-Toe state in GDL.

To give a concrete example of a state transition in GDL, consider the states in Figure 3.3 as well as the description from Figure 3.2.

In the case where action $mark(3,3)$ is taken by $xplayer$ in state $s_1$, we expect the fluent $cell(3,3,x)$ to be true in subsequent state $s_2$. According to the rules in lines 15-17, we can see that $does(xplayer, mark(3,3))$ is true and the $cell(3,3,b)$ is also true in $s_1$. Therefore, according to the rule, $next(cell(3,3,x))$ is true and thus fluent $cell(3,3,x)$ must be true in the subsequent state $s_2$. According to the rules in lines 36-37, $control(oplayer)$ must be true in the subsequent state $s_2$ to ensure turn-taking in the game. All $cell$ fluents with either $x$ or $o$ are retained from $s_1$ to $s_2$ according to the rule in lines 23-24, as they are non-blank cells - $x$ and $o$ are distinct from $b$. Finally, the $cell$ fluents with $b$ that are not $cell(3,3,b)$ are retained as well from $s_1$ to $s_2$ due to the rule in lines 27-31. This rule states that the blank cell is retained if the coordinates in $mark(3,3)$ are distinct from that particular blank cell.

### 3.3.1 Modularity

One feature of GDL that exists as a consequence of its logical structure is its modularity - games with similar features will often have similar rules defined within their descriptions. For example, games that feature turn-taking of some nature will contain rules defining a dummy *noop* action as well as rules that alternate which player must take the *noop* action. The minimal syntax required to define such rules is similar across differing games, with slight variation due to game-specific elements such as the number of players.

While this is helpful for users wanting to quickly write descriptions of other games (or to make variants of pre-existing games), these similar game rules are a key feature that must be exploited by both reasoners and game players if they wish to transfer learned inferences or strategies across different games.

### 3.3.2 Scrambling and renaming

To ensure GGP agents are not simply referencing a stored database of collected games and strategies, it is common to apply a *scrambling* and *renaming* to the game description prior to testing. Scrambling a game description reorders the rules and the positioning of arguments within a rule without changing the semantic meaning of the description. Renaming changes the identifiers used for predicates, variables and functions in a consistent manner that also retains the semantic meaning of the description. Essentially, scrambling and renaming obscures the original description to ensure that agents do not simply rely on rote memorisation.

As such, it is expected that reasoners and game players are to retain the same performance and accuracy across scrambled games. In the case of those that utilise a learning element, a model trained on game $G$ should be transferable to the scrambled version $scramble(G)$ with no effect on its effectiveness. However, those that overfit to

specific descriptions of a game would have to treat each scrambled game as a complete different game, severely hampering the transfer capabilities of such agents.

### 3.3.3 Grounding

While the GDL specification allows for the full use of variables to assist in the efficient writing of game rules, there are several cases where it is either necessary or beneficial to *ground* the variables in the description. Propositional networks, for example, require a grounded description to generate the logic gates it uses for inference. Prolog-based reasoners can also benefit from an increase in efficiency when using grounded descriptions. While some game descriptions naturally do not contain any variables, for the vast majority of game descriptions that do, a grounding process can be applied to generate a grounded version of the original game description. Note that this new grounded game description would be distinct from the original description, unlike a scrambling or renaming.

However, the grounding process itself can often be computationally expensive, with some game descriptions being intractable to ground at all. This means that methods that require the grounding of game descriptions will be limited only to games in which grounding is tractable and practical.

## 3.4 GGP Environment and Agent Methods

In this section, we will investigate the various items necessary to run a GGP environment and the agents that play in them. First, we present the infrastructure that facilitates the management and communication within a GGP setting between agents. Secondly, we examine the automated reasoning techniques that agents use to infer the facts of a game. Finally, we examine the most commonly used search methods that successful GGP agents utilise.

### 3.4.1 Game Management Infrastructure

The GGP setting requires some central authority which manages the interactions between the different agents and the game itself. Figure 3.4 shows an overview of a game management infrastructure, in which a *game manager* communicates with the players in the game, initially providing the game description of the game being played and subsequently sending information on the current state and receiving the selected actions of each player. The game manager then stores these selected actions and current state information temporarily to infer the actualised subsequent state, repeating this process until the game terminates. Additionally, the game manager can also be used to generate graphics or visualisations for spectators as well as storing match records for future use.

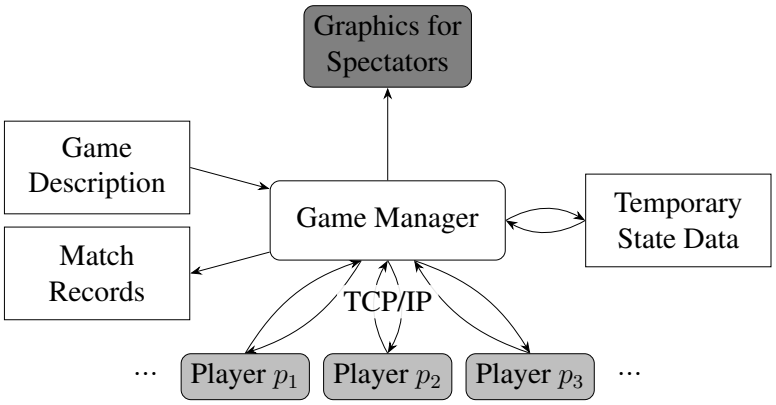



Figure 3.4: Network Game Manager. Adapted from (Genesereth et al., 2005)

For tournament settings, the game management infrastructure is usually operated over a network, with the game manager communicating to the agents over the network with a TCP/IP protocol. The game manager may also apply scrambling or renaming to the game description before providing it to the competitors, as well as providing the strict time limits they must select their actions under.

To simplify experimentation when dealing with agents that utilise machine learning methods, it is often helpful to use an alternative infrastructure that is hosted locally.

Instead of using a TCP/IP protocol to communicate between agents and the game manager, all communication can be done locally on shared memory. This is especially helpful for the implementation of training methods which require simulated game plays as the agents and game manager can more rapidly play out the games. Additionally, this allows for direct communication from GPU-based implementations with minimal overhead caused by network protocols or interfaces.

### 3.4.2 Automated Reasoning

As all agents are required to infer the game rules themselves, they must all implement some type of automated reasoning. The earliest agents usually implemented GDL-specific reasoners based on common techniques used in logic programming. More commonly today, agents use either of the following type of reasoner: a Prolog-based reasoner or a propositional network. These reasoners have several trade-offs, and the choice of reasoner to use is dependant on the agent's requirements (Björnsson & Schiffel, 2013; Schiffel & Björnsson, 2014).

#### Prolog-based reasoners

As GDL is based on Datalog, it shares many similar features to a wide array of logical programming languages that have already existed beforehand. As such, GDL syntax can be easily converted to equivalent Prolog syntax. This allows the agent to use a well-established Prolog engine such as YAP Prolog (Costa, Rocha & Damas, 2012) or SWI-Prolog (Wielemaker, Schrijvers, Triska & Lager, 2012), taking advantage of the decades of optimisations that have been implemented in these applications.

This type of reasoner is relatively simple to implement and is able to infer a large majority of games. Additional optimisations can be made such as caching and optionally games may be grounded beforehand for a boost in efficiency.

#### Propositional networks

Despite the optimisations of well-established Prolog compilers and interpreters, they can still suffer from inefficiencies that are fundamental to the reasoning techniques used by logical programming languages. As such, alternative reasoning techniques such as propositional networks were developed as a way to more efficiently reason GDL descriptions.

A propositional network represents the game rules of a GDL description as a circuit, consisting of propositions, logic gates and transition gates. Instead of a state-to-state transition like in Prolog reasoners, the transition gates handle the dynamics of a game by acting as a flip-flop latch that controls the flow of information from one step to the next.

However, converting a game description into an equivalent propositional network requires the game to be fully grounded. As mentioned earlier, grounding a description can cause issues, as some larger games can be either impractical or even intractable to ground. As a result, several agents utilise both types of reasoners, falling back to a Prolog-based reasoner in the case that the description cannot be grounded.

### 3.4.3 Search Methods

All agents in both traditional computer game playing as well as GGP typically use some sort of search algorithm to effectively find an optimal strategy. Search methods allow for the agent to look further ahead than simply evaluating the current state. As games typically do not have immediate rewards from actions and instead have very delayed rewards typically only in terminal states, this longer reaching evaluation is necessary for avoiding poorly performing policies that suffer from a lack of foresight.

Broadly speaking, all search methods used by a game playing agent will expand the game tree as much as possible from the current game state. The leaf nodes of this

expanded game tree, also known as the search horizon, will then be evaluated using either a heuristic (in the case of traditional minimax algorithms), a stochastic sampling (such as in MCTS) or simply using its actual utilities (when it is a terminal state). It will then apply a backup procedure to the currently expanded game tree using the values from the search horizon, propagating these values up the game tree. As the backup procedure should follow the edges of the game tree up to the root node, this process eventually provides an evaluation of all the possible action at the current game state as well as the directly subsequent game states. These can then be used directly as a policy for the agent to follow or for further processing before selecting an action.

The two most relevant search algorithms for use in GGP is the minimax algorithm and Monte-Carlo Tree Search. While minimax has been used by some earlier GGP agents, it has generally been replaced by MCTS-style algorithms in more recent agents. Regardless, it still has important theoretical implications for strategy optimisation. When applied to a complete game tree, minimax will always find a strategy that is a Nash equilibrium. However, as generating the complete game tree for more complex games can be intractable, the Nash equilibrium strategies are not achievable in most games (typically, agents that used minimax would use a depth-limited minimax search with a heuristic evaluation function for leaf nodes). As such, when discussing other search methods such as MCTS, convergence guarantees to Nash equilibria are relative to the theoretical minimax solution of the complete game tree.

MCTS has been shown to be successful in past GGP tournaments, but is more importantly the core reinforcement learning technique utilised in deep reinforcement learning. AlphaZero-style agents utilise a combination of MCTS and deep neural networks to reach the performance they are able to achieve.

### Minimax

One of the earliest forms of search used for game playing is the minimax algorithm. The algorithm applies depth-first search to the game tree to find the utilities of each player in terminal states. These utility values are propagated up the game tree by recursively selecting actions that maximise the active player's utility. Figure 3.5 shows an example of the minimax algorithm applied to a small game. When applied to a full game tree such as in this example, minimax will provide a Nash equilibrium solution to the entire game, effectively providing a perfect strategy for the agent to follow.

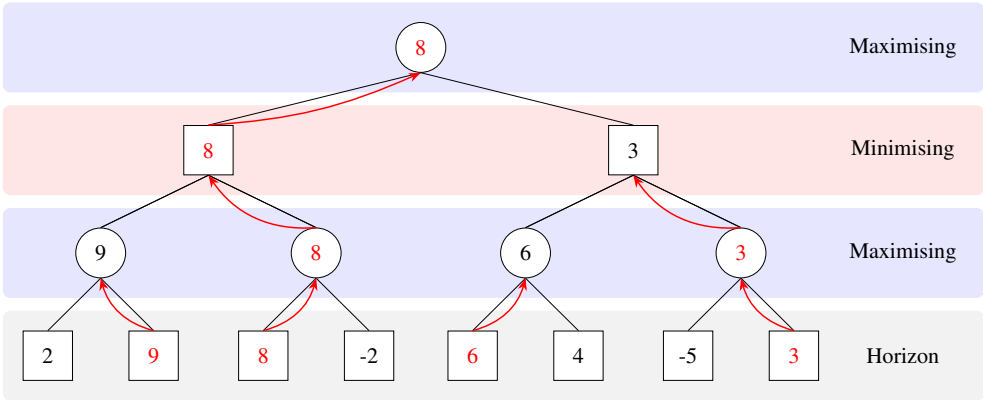



Figure 3.5: Minimax applied to an example game tree.

Additionally, alpha-beta pruning can be applied as an optimisation over standard depth-first search minimax. Alpha-beta pruning maintains a lower and upper bound of encountered values during the search to cut branches in which no improved utility can be found for the active player. Despite cutting entire sections of the game tree, minimax with alpha-beta pruning will still provide the same solution as a complete tree search minimax. However, intermediate values within the tree may not be the same.

While the "minimax" name refers to the two players in a turn-taking, zero-sum game (one player will be maximising the utility, while the other minimising), the general principle can be extended to general-sum games, games with more than two players and games with simultaneous actions. For general-sum games, instead of maximising or

minimising a shared score value, each agent instead simply maximises their own utility. Likewise, for games with more than two players, each player will simply maximise their own utility instead. For games with simultaneous actions, each state can be treated as its own normal form game, with the joint action selected being a Nash equilibrium. Alpha-beta pruning can also be extended to games with simultaneous actions (Saffidine, Finnsson & Buro, 2012).

Even with optimisations such as alpha-beta pruning, the game trees of more complex games such as Chess and Go are too large to generate in its entirety. Agents that utilised minimax in the past usually used a depth-limited form, in which the search horizon usually does not reach the terminal states (unless the game state is already near the end of the game). As a result, the states in the horizon will mostly be non-terminal and therefore do not have concrete utility values for the minimax algorithm to utilise. Evaluation functions and heuristic estimations must be used instead, approximating the values of these non-terminal states. These approximation methods either require experts to program them, or if learnt are much more susceptible to errors and inaccuracy. Using minimax with evaluation functions suffers from poor performance with more complex games. However, the minimax principle guarantees that a solution does exist for the game if a theoretical complete game tree can be solved with minimax. This provides a theoretical ideal solution that other more effective approximation algorithms may attempt to converge towards.

### Monte-Carlo Tree Search

As mentioned earlier, the minimax algorithm requires the usage of a heuristic estimation alongside a depth-limited search as generating the complete game is impractical for many games. As mentioned earlier, this heuristic estimation typically acts in a static manner and as such requires a relatively deep search into the game tree to generate effective strategies. Monte-Carlo Tree Search avoids this problem by dynamically

searching the game tree without a specific depth limit and uses stochastic sampling to estimate the value of states through random simulations.

Rather than expanding the game tree uniformly, MCTS iteratively expands the search tree according to a selection strategy, balancing exploitation and exploration to push the search horizon asymmetrically. Instead of using a heuristic evaluation function to estimate state values, it instead uses random *playouts*: a complete simulation consisting of randomly selected actions until a terminal state is reached. This stochastic sampling of simulated trajectories allows MCTS to make relatively effective estimates of state-action values without relying on any game-specific or expert-aided evaluation functions. This makes it an excellent choice in the more general setting of GGP.

The deep reinforcement learning technique used in AlphaZero-style agents make use of MCTS as part of its reinforcement learning component. We will mention some of the ways MCTS is modified for use in deep reinforcement learning in this section, but will provide a more comprehensive overview in Section 3.5.

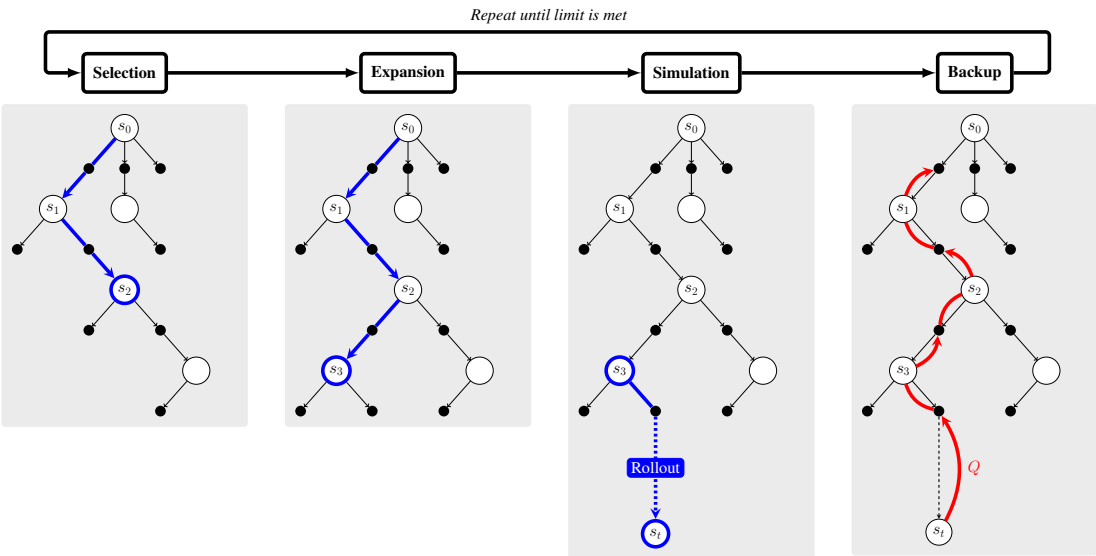



Figure 3.6: Monte Carlo Tree Search. Adapted from (G. Chaslot et al., 2008)

Generally, each iteration of MCTS consists of four main steps: Selection, Expansion, Simulation and Backup. Figure 3.6 illustrates these four steps and the following

describes the steps in more detail (Sutton & Barto, 2018):

1. **Selection.** Starting at the root node (the state in which an action is to be selected), MCTS follows a *tree policy* based on the state-action values to traverse the currently expanded game tree until it selects a leaf node or a node with an unexplored action. If the the selected leaf node is terminal, the Expansion and Simulation steps are skipped.

2. **Expansion.** Assuming the selected node is non-terminal, the game tree is expanded from that node, adding a new child node along an unexplored action.

3. **Simulation.** Starting from the newly added child node, a complete simulation playout episode is run by selecting random actions until a terminal state is found. The states and actions visited during this random playout are not saved.

   When used in deep reinforcement learning, this simulation is replaced with a neural network approximation of the newly added child node instead of a random playout.

4. **Backup.** The estimated value returned by the simulation (either from the terminal state from a playout or neural network approximation) are backed up the currently expanded game tree to update the state-action values along the edges traversed during the selection step.

These iterations are repeated until a limit is reached, such as a time limit for a competition setting. Once all iterations are completed, the accumulated state-action values on the root node and its actions, collected from multiple iterations worth of backups, are used as a policy for the agent to follow. For subsequent states, the previously expanded game tree can be retained, moving the root node to the new current state and retaining all previous expanded descendants.

As the tree policy used during the selection step directs the way in which the search tree is grown, it must balance between exploitation of regions of the tree with known state-action values and exploration of lesser known action paths. This can be done with an Upper Confidence Bound (UCB) selection rule, which combines the estimated state-action value and a term that prioritises lesser explored actions:

$$U(s,a) = Q(s,a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{3.1}$$

where $s$ is the current state being traversed, $a$ is the action being considered from state $s$, $Q(s,a)$ is the estimated state-action value, $N(s)$ is the number of times state $s$ has been visited, $N(s,a)$ is the number of times action $a$ has been taken from state $s$ and $c > 0$ is a constant for the exploration factor. $Q(s,a)$ is continually updated during the backup step of MCTS at the end of every iteration.

The $Q(s,a)$ term in UCB acts as a greedy action selection, exploiting highly valued state-action pairs known from previous iterations. The $\sqrt{\frac{\ln N(s)}{N(s,a)}}$ term controls the exploration of the tree policy. The term grows inversely proportional to the number of times an action has been traversed in a state, leading to preference to lesser explored actions. To aid in computation, a small non-zero value $\epsilon$ can be added to $N(s,a)$ to ensure that in cases when $N(s,a) = 0$ (the action has not been taken before) the unexplored action is prioritised.

After the $U(s,a)$ values have been computed for each action at a given state, the tree policy can either take the action with maximal value $\mathrm{argmax}_a U(s,a)$ or by treating the normalised UCB values as a probability distribution. For deep reinforcement learning, the UCB selection rule can be modified to also utilise the neural network approximation. The exploration term can be combined with the neural network's estimated policy.

## 3.5 Deep Learning for Games

In this section, we will provide preliminary information on the deep learning techniques that will be utilised in later chapters. First, we will examine the neural network architectures that are used in game playing agents and GGP. In particular, we describe the architectures of fully-connected neural networks, convolutional neural networks and graph neural networks. Secondly, we explore the deep reinforcement learning techniques used by AlphaZero-style agents and describe how such an agent is able to learn.

### 3.5.1 Neural Networks

Neural networks (NN) are nonlinear function approximators $f(x, \theta)$ where $x$ is the input to the network and $\theta$ are the trainable parameters or *weights* of the NN. The inputs $x$ are typically represented as a vector, matrix or tensor of features depending on the architecture used. We call a NN a *deep* neural network when it consists of multiple, sequential layers.

To train the parameters of a deep neural network, a backpropagation procedure propagates the gradients of a loss function through the layers of the neural network. This allows us to train a neural network in an end-to-end manner with a dataset of training examples, regardless of the number of layers or modules used in the architecture. This is typically done using techniques such as stochastic gradient descent (SGD) or algorithms such as Adam (Kingma & Ba, 2015). The loss function used is dependent on the task the neural network is meant to approximate, such as mean squared error for regression or cross entropy for classification.

Although neural networks are utilised in different types of learning, we will focus on using neural networks for supervised learning. In supervised learning, the neural network learns using training samples $(x, y)$ and a loss function $L(y, y')$, where $y'$ is

the output of the neural network $f(x, \theta) = y'$. The loss function compares the actual ground truth compared to the prediction made by the neural network. The resulting error is used to compute the gradients that are then applied to the weights of the neural network.

In most applications for game playing, neural networks are used to approximate some sort of value function or policy estimation from prior experience. This is best done by treating it as a supervised learning task, with game states (or a history of states) as input to the neural network and the utilities, policies or inferences as a training target.

### Fully-connected Neural Networks

Fully-connected neural networks (also known as feed-forward neural networks or multilayer perceptrons) are the basis of all deep neural network architectures that are used today. They consist of a sequence of fully-connected layers in which neurons apply an activation function over a linear combination of the output values from all the neurons of the previous layer. Fully connected neural networks are also commonly used as a component of other types of neural network architectures, usually as the last few layers prior to the output layer.

A typical fully-connected neural network consist of three types of layers: an initial input layer that corresponds to the size of the input elements, multiple hidden layers that transform the input, and a final output layer that converts the intermediate representations into an appropriate output. The output layer dictates the type of task the neural network is meant to learn, such as classification or regression. For example, classification tasks will either use a sigmoid or softmax function to convert the intermediate raw values into a final probability value for the classes. Figure 3.7 shows an example of a fully-connected neural network.

Fully-connected neural networks can also be efficiently represented as a sequence of mathematical functions. As each neuron consists of a linear combination of all neurons

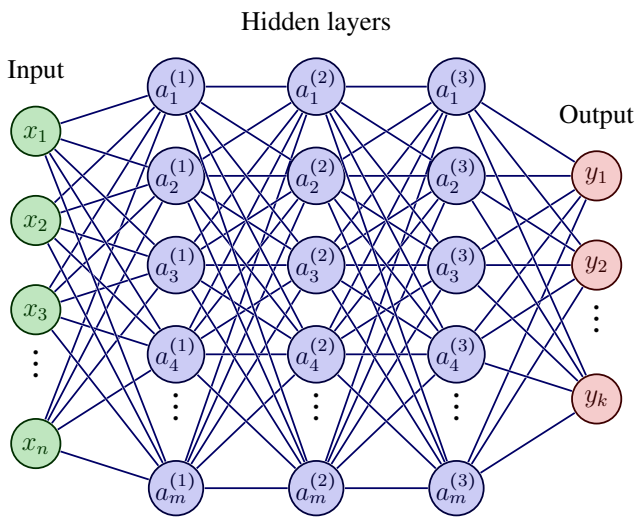



Figure 3.7: A fully-connected neural network. Retrieved from `https://tikz.net/neural_networks/`

in a prior layer, their corresponding weight values and an activation function, we can define the value of a single neuron as the following:

$$a_i^{(l+1)} = \sigma\left(\sum_{j=0}^{n} w_{i,j} a_j^{(l)} + b_i^{(l)}\right) \tag{3.2}$$

where $a_j^{(l)}$ is the activation value of the $j$th neuron of the $l$th layer, $w_{i,j}$ is the weight from the $i$th to the $j$th neuron, $b_i^{(l)}$ is the bias term and $\sigma$ is a non-linearity such as the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ or Rectified Linear Unit (ReLU) $\sigma(x) = max(0, x)$. The non-linearity acts as the activation function of the neuron.

As each layer consists of a stack of neurons, one can represent the entire layer using a vector of neurons. Likewise, the weights of a layer can be represented as a *weight matrix*. Using Equation 3.2, an entire layer can be defined as:

$$\begin{bmatrix} a_1^{(l+1)} \\ a_2^{(l+1)} \\ \vdots \\ a_m^{(l+1)} \end{bmatrix} = \sigma \left( \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_n^{(l)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_m^{(l)} \end{bmatrix} \right) \tag{3.3}$$

$$\mathbf{a}^{(l+1)} = f(\mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)}) \tag{3.4}$$

This matrix representation of a neural network layer illustrates why they have grown in effectiveness with the advent of powerful GPUs. As GPUs are able to execute matrix operations in a very efficient parallel manner, the computation of each layer of a neural network can be completed incredibly quickly. Combined with the large amounts of memory available on most GPUs, fully-connected neural networks are able to be scaled to larger sizes while still retaining efficient operation.

### Convolutional Neural Networks

For data modalities in which the elements can be arranged in some sort of array structure in which the relative positions of the elements are a relevant feature, convolutional neural networks are well suited as their architecture is able to take advantage of the grid-like structure present in such representations. The array-like data is usually represented as a matrix (if 1-dimensional or 2-dimensional) or a tensor (if 3-dimensional or higher).

Convolutional neural network layers utilise a kernel that applies a convolution to a localised area of the input. This kernel contains the trainable parameters or weights of a convolutional neural network and is applied repeatedly across the entire span of the input, moving according to a defined stride parameter. The values computed by the kernel convolution are then output for the next layer, following the same grid layout

as the input. A single convolutional layer can consist of multiple terminals, generating multiple layers of output. Figure 3.8 shows an example of a single convolutional layer with a $3 \times 3$ kernel.

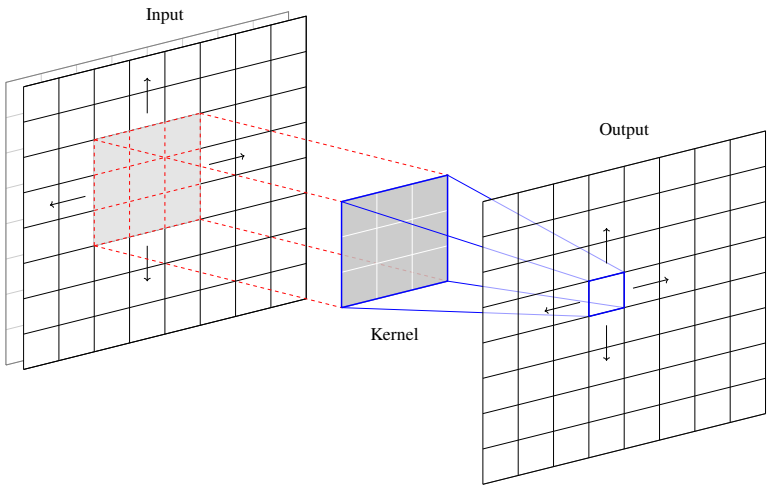



Figure 3.8: A convolutional layer.

Beyond the convolutional layers themselves, convolutional neural networks also typically utilise a pooling layer that either averages or takes the maximum value in a local kernel to group and merge locally detected features together. Like in fully-connected neural networks, convolutional neural networks also use a nonlinearity every layer in a similar manner. Deep convolutional neural networks are constructed by stacking these convolutional, pooling and nonlinear layers sequentially. The final layers of such an architecture typically uses several fully connected layers to generate the output, unrolling the array representation into a vector.

Due to the weight-sharing behaviour of the kernels in a convolutional layer, convolutional neural networks exhibit two strong relational inductive biases: locality and translation invariance. As the convolution operator applies to a localised grid of nearby elements, it imposes a strong relationship between entities in close proximity to each other. With the reuse of the kernels across the span of the input, local features can be detected regardless of where they are located in the input space.

### Graph Neural Networks

Graph neural networks are trainable functions that transform attributes in an input graph for a specified task, while still maintaining the relational structure of the original graph. The graphs can be directed or undirected, contain self-loops, have multiple edges between nodes and are *attributed*. An attributed graph allows for nodes and edges to have information embedded within them, as well as having a global graph-level attribute. The attributes can be any kind of information, but are typically encoded as a vector, matrix or tensor representation for ease of computation.

While there are several kinds of neural networks that operate on graphs, the most commonly used architectures utilise a *message-passing* mechanism. A message-passing graph neural network layer diffuses information throughout the graph by applying individual operations on every node of the graph. Although the specific operation applied can vary between different graph neural network implementations, they generally consist of two phases. Given a node $n$ and its neighbourhood $\mathcal{N}(n)$, the following are applied:

- **Message passing.** For each neighbour $m \in \mathcal{N}(n)$ of node $n$, a differentiable function (which may be some kind of neural network) is applied to each neighbour's attribute. The function can also applied to the attribute of node $n$ itself as well.

- **Aggregation.** The outputs of the function applied to all neighbour's attributes are collected and passed through an aggregation function (such as averaging, summation or other more complex function). The aggregated embedding can then be passed through another differentiable function.

Aside from the two phases mentioned above, the operator may apply additional functions and operations to the messages and aggregated embeddings. Following this general outline, a variety of graph neural network architectures have been implemented,

such as Graph Convolutional Networks (Kipf & Welling, 2017) and Graph Attention Networks (Veličković et al., 2018). Graph Attention Networks in particular apply a learnable attention weighting to neighbouring nodes prior to aggregation. Figure 3.9 shows an overview of a graph neural network layer utilising the message-passing mechanism.

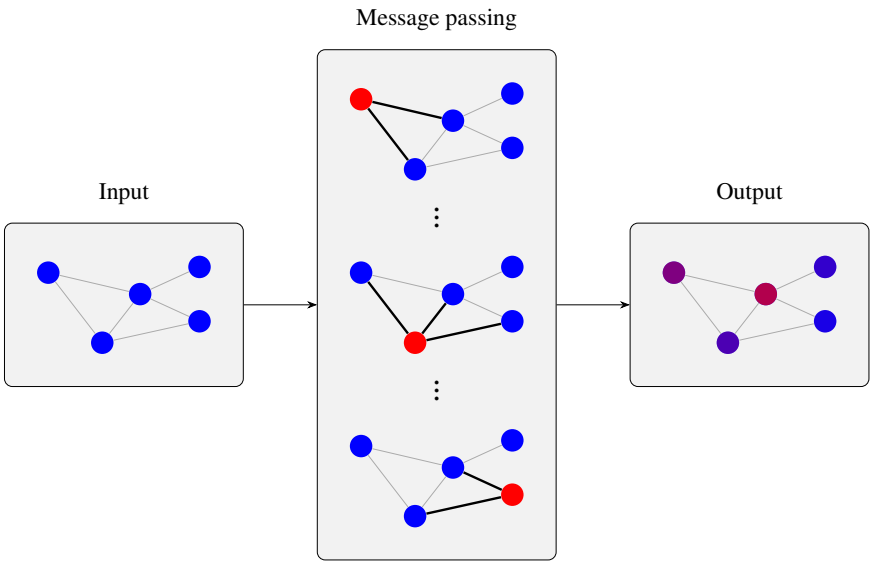



Figure 3.9: A message-passing GNN layer.

Like with convolutional neural networks, a graph neural network will typically consist of multiple message-passing layers, followed by several fully-connected layers prior to the output. However, with graph learning tasks, there are two types of possible outputs: either node classification or a global, graph-wide classification. In the case of node classification, the fully-connected neural network will attend to each node separately to classify them. For a graph classification task, a global pooling operator is applied across all nodes, edges and global attributes of the graph to generate a global, graph-wide embedding. The fully-connected neural network attends to this graph embedding to classify the entire graph.

### 3.5.2 Deep Reinforcement Learning

Deep reinforcement learning has emerged to be the premier game playing technique across a variety of games. The technique combines a neural network guided Monte-Carlo Tree Search and self-play training. As mentioned earlier, these techniques were popularised by AlphaGo's success in Go (D. Silver et al., 2016).

The general principle of the training scheme is that the resulting value and policy generated from combining both MCTS and a neural network will be more accurate than just the neural network itself. As such, one can consistently generate training samples to improve the accuracy of the neural network. This leads to a positive feedback loop in which the improved neural network generates improved training sample when it is combined with the MCTS.

A training iteration consists of the following steps:

- Generate training samples through self play.
- Train neural network with training samples
- *Optional:* Perform evaluation of newly trained network against prior network. The neural network is replaced with the newly trained network only if it outperforms the prior network.

This process repeats until a sufficient number of iterations are reached or some criteria is met. With sufficient tuning, a deep reinforcement learning agent can continuously train without reaching a limit.

#### Neural network guided MCTS

One of the hallmarks of the deep reinforcement learning technique utilised by AlphaGo, AlphaGoZero and AlphaZero was the novel combination of Monte Carlo Tree Search

and deep neural networks. The neural network is used in two key aspects: the selection rule and as a replacement for random playouts.

Given a state $s$, action $a$ and a neural network approximator $f(s, \theta) = (v, \boldsymbol{\pi})$ with estimated value $v$ and policy $\boldsymbol{\pi}$, the modified selection rule is the following:

$$U(s, a) = Q(s, a) + c\boldsymbol{\pi}_a \sqrt{\frac{\ln N(s)}{N(s, a)}} \tag{3.5}$$

where $\boldsymbol{\pi}_a$ is the probability of taking action $a$ according to policy $\boldsymbol{\pi}$.

This selection rule uses the policy learned by the neural network to encourage exploration towards more favourable areas of the game tree. As the neural network's accuracy improves, this preference will grow to be more effective over time.

During the simulation step of MCTS, the neural network's estimated value $v$ is directly used for the later backup step instead of a random playout. Again, this estimation will improve over time as the neural network's value estimation becomes more accurate.

### Self-play

To generate the training samples for the neural network, deep reinforcement learning utilises *self-play*. Essentially, the agent plays games against itself to collect game histories using the currently trained neural network $f(s, \theta) = (v, \boldsymbol{\pi})$. A game history is a sequence of states $s_1, s_2, \ldots, s_t$ where $s_t$ is the terminal state. A single training sample then consists of a state $s_i$ as the input and the policy generated by the neural network guided MCTS for the state $\mathrm{MCTS}_{NN}(s_i, \theta)$ as the target policy and the actual utility value at the terminal state $u(s_t)$ as the target value. Multiple self-play games are carried out to accumulate a large collection of states to ensure a good distribution of data. With these samples collected, the neural network can then be trained by computing the losses $L(v, u(s_t)) + L(\boldsymbol{\pi}, \mathrm{MCTS}_{NN}(s_i, \theta))$.

## 3.6 Conclusion

In this chapter, we have provided key background information and technical details necessary for a GGP agent that utilises learning. We have presented the standard specification of GGP and have discussed several key features of GDL relevant to a learning approach. We have discussed the infrastructure for managing games between various agents in GGP as well as the techniques used by the agents themselves. Finally, we presented the fundamental concepts behind deep neural networks and deep reinforcement learning. The concepts covered here will be further developed and expanded upon in later chapters.

# Chapter 4

# Exploring a Learning Architecture for GGP

## 4.1 Introduction

The recent accomplishments of DeepMind's AlphaGo (D. Silver et al., 2016) have reignited interest in game-playing Artificial Intelligence, showing the effectiveness of Monte-Carlo Tree Search (MCTS) with deep neural networks and learning through self-play. Further work on AlphaGo Zero (D. Silver, Schrittwieser et al., 2017) showed that this method was effective even without any human expert knowledge, and in fact able to defeat their previous AlphaGo agent despite learning from effectively zero knowledge apart from the games rules. This method of learning to play a new game was extended beyond Go to AlphaZero (D. Silver, Hubert et al., 2017) for learning to play Chess and Shogi with effectively the same architecture.

In the same vein as AlphaZero, General Game Playing (GGP) proposes a challenge: developing agents that are able to play any game, given a general set of rules (Love et al., 2008). GGP uses a logical language known as Game Description Language (GDL) to represent rules of arbitrary games. These game descriptions are a combination of a

knowledge base containing static rules and a state containing dynamic facts. This raises the natural question whether the same methods that AlphaZero uses to learn to play Go, Chess and Shogi can be applied to general games described in GDL. This was recently addressed with Generalised AlphaZero (Goldwaser & Thielscher, 2020), a system that applies deep reinforcement learning to GGP. While the overarching approach outlined by AlphaZero provides a general architecture for a self-play learning agent, there are a few key limitations that had to be overcome for this purpose:

- AlphaZero assumes the games are two-player, turn-taking and zero-sum.
- Neural network architectures are hand-crafted for each game, encoding features such as board geometry.
- As a result of the specialised neural networks, the agents have some form of implicit domain knowledge for each game.

In the domain of GGP, games are neither required to be two-player, turn-taking nor zero-sum so that a GGP agent must be able to account for games with any number of players, simultaneous action games and non-zero-sum games. With regard to the second point, the agent must also be able to generate a neural network for any given game description, without requiring any additional modification from a human. Finally, the system should be general for any game with no specialised domain knowledge at all.

In this chapter, we present a system to overcome these limitations in an AlphaZero-style self-play reinforcement learning agent designed to learn games in the GGP setting. Our main contributions lie in the further exploration of this learning architecture:

- We confirm the feasibility of deep reinforcement for GGP under different settings, including the use of a different GGP reasoner.
- We analyse the impact of type and depth of the underlying neural network.
- We investigate simulation vs. time limitations on training.

The rest of this chapter is organized as follows. Section 4.2 describes the architecture of our self-play reinforcement learning agent, GGPDeepRL . In the extensive Section 4.3 we present the results of our experiments, showing the agent's capabilities in a variety of games as well as an in-depth examination in the agent's architecture. Finally, Section 4.3 concludes the chapter by discussing the key observations from the results and the limitations of the approach.

## 4.2 GGP Learning Agent Architecture

Like all other GGP agents, GGPDeepRL still requires two core components: a reasoner and a search algorithm. As such, the general flow must still exist: a game description is provided to the agent, a reasoner infers the relevant facts about the game to be able to expand a game tree and finally a search algorithm finds an optimal strategy to play. Adding a learning element to a GGP agent would require additional components that augment the search algorithm used by the agent. Using an AlphaZero-style architecture would require these additional components to be a neural network that is used within the search algorithm as well as a self-play reinforcement learning component to train the neural network.

Figure 4.1 provides an overview of the overall architecture of our system. In more detail, GGPDeepRL consists of four main components, each made up of smaller subcomponents. The first component is the reasoner, which takes the GDL description and processes it to generate the fluent and action lists used to initialise neural network as well as conduct inference during play. The second component is the neural network, which is used as the policy and utility estimator for the search algorithm. The third component is the search algorithm, using both neural network estimation and MCTS for policy generation. Finally, the self-play reinforcement learning component is used to train the neural network with training samples generated by the search algorithm.

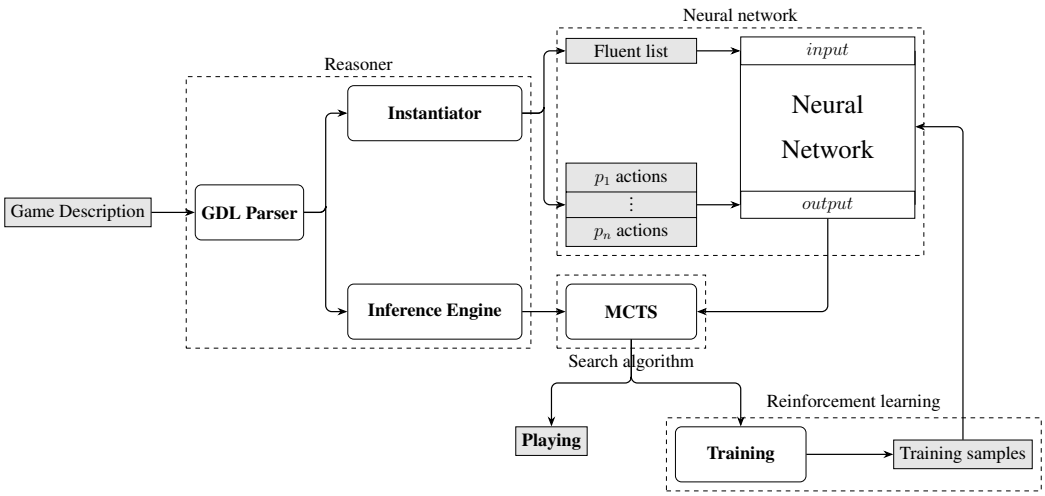



Figure 4.1: Architecture of GGPDeepRL agent.

While GGPDeepRL 's architecture is similar to that of AlphaZero, there are two key differences. Firstly, as the GGP platform requires the agent to infer the rules of a game from a GDL description without human aid, the agent requires a reasoner to do this inference. Secondly, we use the reasoner's instantiated fluent and action lists to automatically generate the neural network, as we cannot rely on hand-crafted neural network architectures for the more general setting of GGP. Additionally, the search algorithm and reinforcement training have been designed with the GGP setting in mind, as the games may have an arbitrary number of players and are not strictly zero-sum. The rest of this section will provide more detail on each individual component of the GGPDeepRL architecture.

### 4.2.1 Reasoner

The GDL parser takes as input a game described in GDL and converts it into appropriate representations for the instantiator and the inference engine. The instantiator generates the sets of all possible *fluents* (dynamic predicates that are true for a given state) and all possible actions, using the method described in (Vittaut & Méhat, 2014b, 2014a).

The inference engine is used by MCTS to infer facts about the game such as

legal moves, subsequent states, terminal conditions and utility values. There are two main inference methods: Prolog-based and Propositional network (Propnet) based (Schkufza, Love & Genesereth, 2008). A Prolog-based method converts the GDL rules to an equivalent Prolog representation and uses an off-the-shelf Prolog reasoner for inference. In our implementation, we use the Prolog-based inference engine for its greater generality, while a Propnet-based inference method is used in Generalised AlphaZero(Goldwaser & Thielscher, 2020). While this is more efficient than a Prolog-based method, Propnets are not applicable to some larger games as the grounding process can be intractable. Additionally, to improve the efficiency of the agent, we save the results of inference queries. This allows repeated visits to states during the MCTS to be more efficient, especially as the neural network will tend to prefer certain optimal states and actions after sufficient training.

### 4.2.2 Neural Network

#### State representation

We make use of a *fluent list* vector representation as input for the neural network. As a state in GDL is defined by a set of fluents, we can define a fluent list as a fixed list of all possible fluents that can occur in a game. The fluent list vector is then a one-hot encoding of the true fluents of the given state. For an input vector **f** = $(a_1, a_2, \ldots, a_n)$ and a state consisting of fluents in the set $S \subseteq \{s_1, ..., s_n\}$, the value of element $a_i = 1$ if $a_i \in S$, otherwise $a_i = 0$. The neural network outputs for each player a normalised policy vector for player's actions and the estimated goal value.

#### Automatic neural network generation

The neural network architecture is automatically generated based on the game description and a network depth parameter $L$. The reasoner instantiates a fluent list of length

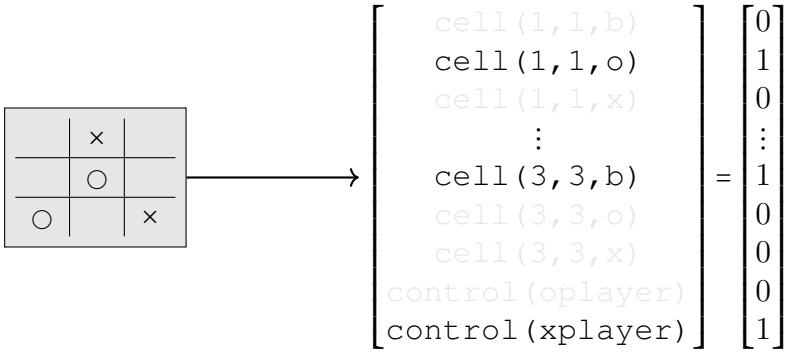



Figure 4.2: Generating a fluent list from GDL state

$f$ (the game has $f$ possible fluents that can occur). It is also used to infer the number of players $n$ and the lists of possible actions for each player, each being of length $a_1, \ldots, a_n$ for each player. The input layers are generated as fully connected layers with ReLU activation based on the length of the fluent list $f$. The hidden layers are initialised by generating $L$ fully connected layers with ReLU activation as well. A set of output layers reduces the width in preparation for the multiple output heads. For each player, an policy head with width $a_1, \ldots, a_n$ and value head is generated. Softmax is applied to the output of the policy heads to generate estimated policy distributions $\boldsymbol{pi}_1, \ldots, \boldsymbol{pi}_n$ and a sigmoid activation is applied to the value heads to generate estimated values $v_1, \ldots, v_n$. As various games will have varying number of possible fluents and actions, the neural network must be separately generated for each game. We plan to develop a neural network that can learn from arbitrary games in the future.

Unlike in AlphaZero, the neural network used in GGPDeepRL uses fully connected layers to ensure compatibility across games. Although convolutional neural networks (CNN) have shown to be effective in the case of AlphaZero, they are not general enough to be used for general game playing. CNNs require the input to have a grid-like structure, which suits the states present in the board games Go, Chess and Shogi. In GGP, the games are not necessarily played on a board and those that are would require a hand-crafted CNN and method to encode the state features into an appropriate grid-like

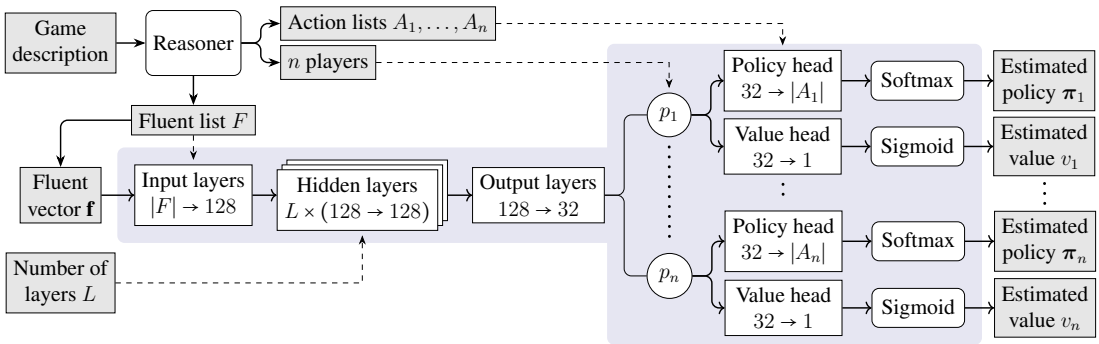



Figure 4.3: Fully connected neural network architecture generated from game description

input.

### 4.2.3 Search algorithm

The search algorithm outputs the finalised policy vector to be used in play and self-play reinforcement training. *Simultaneous action* MCTS (Lanctot, Lisỳ & Winands, 2013) is used as it is able to accommodate multi-player, general sum games with simultaneous actions. Rather than using random playouts during the expansion phase, the search algorithm uses the neural network to estimate the value of a newly expanded state. Additionally, the policy estimated by the neural network is used in conjunction with the UCT term during the selection phase. As a result, a modified form of upper confidence bound on trees $U(s, a)$ is used as the selection strategy:

$$U_n(s, a_n) = Q_n(s, a_n) + C_{uct} \cdot P_n(s, a_n) \cdot \frac{\sqrt{N(s)}}{1 + N(s, a_n)}$$

where for player $n$, state $s$ and action $a_n$, $Q(s, a_n)$ is the estimated goal value in the for player $n$ in state $s$ for action $a_n$, $P_n(s, a_n)$ is the estimated policy, $N(s)$ is a node count of state $s$, $N(s, a_n)$ is the action count of $a_n$ at $s$, and $C_{uct}$ is a weight constant for the exploration factor which is set as a parameter. This specific formulation of the selection strategy is based on the PUCT algorithm, like in AlphaGo (Rosin, 2011).

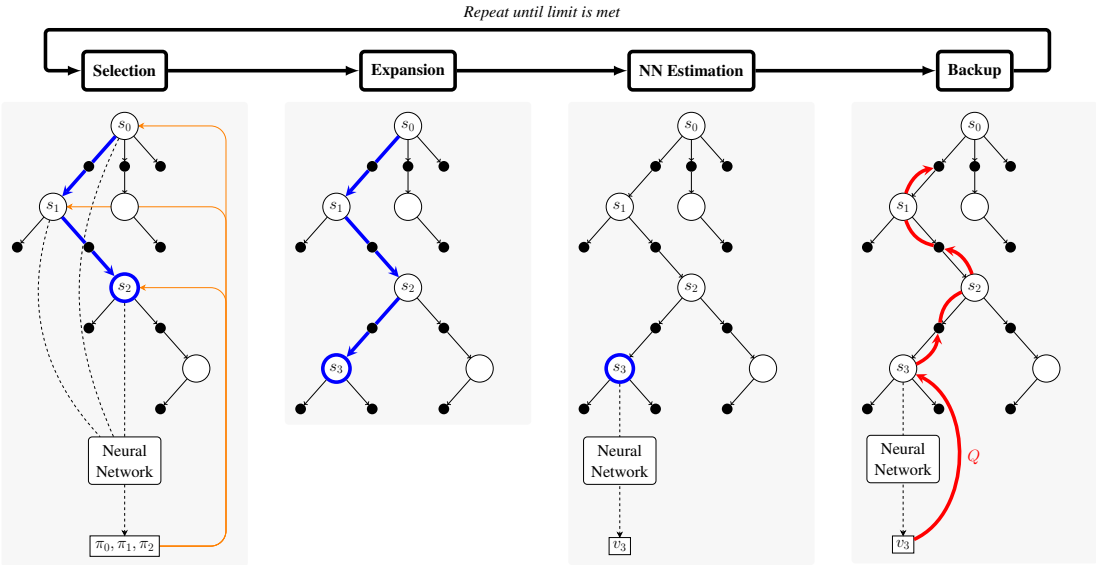



Figure 4.4: Monte Carlo Tree Search with neural-network guided selection and state estimation.

Figure 4.4 shows the modified MCTS algorithm using the estimated policy during the selection phase and the estimated value in place of random playouts. Compared to the standard MCTS algorithm, the selection phase utilises the estimated policy in addition to just the node and action counts when traversing through the tree. The expansion phase is not modified, although which nodes are expanded are influenced by the neural network policy during the selection phase. In lieu of the simulation phase in which a random playout is done, the neural network is used to directly estimate the value of the newly expanded state. Finally, the backup phase is conducted similarly but uses the estimated value from the neural network rather than the playout values.

### 4.2.4 Reinforcement Learning

The agent is tasked with learning an optimal policy for a game by training the neural network to improve its approximation of the function $f^{\star}(s) = (\mathbf{V}, \mathbf{\Psi})$, where $\mathbf{V}$ is the estimated utility and $\mathbf{\Psi}$ is the estimated policy. The neural network with weights $\boldsymbol{\theta}$ is trained on the policy estimated through $MCTS(s; \boldsymbol{\theta})$. In essence, the agent uses the

neural network guided MCTS to continuously improve the neural network only through iterations of self-play. A single iteration of learning has 3 distinct phases:

- **Self-play**: generate training examples using MCTS with the current best network weights $\boldsymbol{\theta}_{best}$.
- **Training**: train the current best neural network with new examples generated during self-play.
- **Evaluation**: compare the performance of the newly trained network $\boldsymbol{\theta}_{new}$ against the current best network $\boldsymbol{\theta}_{best}$. If $\boldsymbol{\theta}_{new}$ outperforms the current $\boldsymbol{\theta}_{best}$, then $\boldsymbol{\theta}_{best}$ is updated by $\boldsymbol{\theta}_{new}$.

To generate training examples for the neural network, the agent plays episodes of self-play games against itself. The policy generated by the MCTS is used as the training target for the policy vector, while the actual goal value at the terminal state of a game play is used as the training target for the estimated goal value. The neural network is trained by minimising the sum of the following, across all players for a training example: (1) **mean squared error** between the neural network estimated goal values and the self-play terminal goal values; (b) **cross entropy** of the neural network estimated policy and the MCTS generated policy. Algorithm 1 provides an overview of the deep reinforcement learning algorithm described above.

## 4.3 Experimental Studies and Results

We first present an evaluation of the GGPDeepRL agent on several games listed in Table 4.1 with their features. The selection of games is similar to (Goldwaser & Thielscher, 2020) with a mixture of complexity levels and game types. Ideally, we would like to have games with varying numbers of players, utility distributions, action sequencing, number of fluents and number of actions. Further experiments are conducted to evaluate various key aspects of the architecture, such as neural network depth, the usage of a

**Algorithm 1** Deep Reinforcement Learning for GGP

**Require:** Game $G$, Neural network weights $\boldsymbol{\theta}$, Training iterations $N$, Self-play games $S$, Game history length $H$

```
1: function DEEPREINFORCEMENTLEARNING(G, θ, N, S, H)
2:     θ_best ← θ
3:     X ← ∅
4:     for 1 to N do                                   ▷ Execute N training iterations
5:         for 1 to S do                               ▷ Execute S self-play games
6:             X_new ← ∅
7:             s ← GETINITIALSTATE(G)
8:             while not ISTERMINAL(s) do
9:                 π_0, ..., π_i ← MCTS(s, θ_best)     ▷ Get action policies for each player i
10:                for all i ∈ GETPLAYERS(G) do
11:                    a_i = max(π_i, GETLEGALACTIONS(s, i))   ▷ Select action for each player i
12:                s ← GETNEXTSTATE(s, a_0, ..., a_i)
13:                x ← (s, π)                          ▷ Create training example x
14:                X_new ← X_new ∪ {x}
15:            u_0, ..., u_i ← GETUTILITIES(s)
16:            for all x ∈ X_new do
17:                x ← (s, π_0, ..., π_i, u_0, ..., u_i)
18:        X ← X ∪ X_new
19:        θ_new ← TRAINNETWORK(θ_best, X, H)          ▷ Train network with at most H newest examples
20:        u_best, u_new ← COMPARENETWORKS(θ_new, θ_best)   ▷ Play comparison games
21:        if u_new > u_best then
22:            θ_best ← θ_new
23:    return θ_best
```

CNN and MCTS simulation limits.

Three benchmark agents are used for these evaluations: a random agent (RANDOM) that samples uniformly from legal actions, a neural-network based agent (NN) which uses the trained network of GGPDeepRL without MCTS search, and a standard MCTS based agent (MCTS). Where applicable, these benchmark agents are given the same parameters as GGPDeepRL .

| Game | Players | Utilities | Action sequencing | Fluents | Actions |
|---|---|---|---|---|---|
| tictactoe (TTT) | 2 | Zero-sum | Turn-taking | 29 | 10 |
| tictactoelarge ($TTT_L$) | 2 | Zero-sum | Turn-taking | 52 | 26 |
| doubletictactoe ($TTT_D$) | 2 | Zero-sum | Simultaneous | 58 | 19 |
| connectfour (C4) | 2 | Zero-sum | Turn-taking | 135 | 8 |
| breakthrough | 2 | Zero-sum | Turn-taking | 130 | 155 |
| blocker | 2 | Zero-sum | Simultaneous | 48 | 16 |
| kalah | 2 | Zero-sum | Turn-taking | 448 | 23 |
| knightstour | 1 | Single-player | | 91 | 124 |
| hamilton | 1 | Single-player | | 80 | 20 |
| babel | 3 | Cooperative | Simultaneous | 157 | 9 |
| pacman3p | 3 | Mixed | Mixed | 476 | 5 |
| connectfour3p | 3 | Mixed | Turn-taking | 147 | 9 |

Table 4.1: Features of games used in experiments. Results are presented as win/loss/draws.

### 4.3.1 Two-player zero-sum games

Table 4.2 shows the results of the experiments conducted on two-player, zero-sum games. Both GGPDeepRL and the MCTS agent use a 1.5 second time-limited search for all games.

| Game | Opponent agent | | |
|---|---|---|---|
| | random | NN | MCTS |
| tictactoe (TTT) | 91/1/8 | 96/0/4 | 1/9/90 |
| tictactoelarge ($TTT_L$) | 48/2/50 | 82/7/11 | 23/25/52 |
| doubletictactoe ($TTT_D$) | 83/3/14 | 62/8/30 | 16/37/47 |
| connectfour (C4) | 95/5/0 | 97/3/0 | 83/17/0 |
| breakthrough | 68/32/0 | 39/61/0 | 48/52/0 |
| blocker | 96/4/0 | 90/10/0 | 54/46/0 |
| kalah | 69/23/8 | 100/0/0 | 8/89/3 |

Table 4.2: GGPDeepRL 's performance on two-player zero-sum games against random, NN and MCTS agents.

GGPDeepRL was able to outperform the NN-only benchmark and achieve comparable performance with the MCTS agent. Interestingly, the MCTS agent seems to perform better in the relatively simpler TTT and its variants. This is likely due to the fact that these less complex games have state spaces small enough for MCTS to search a large enough proportion of the game tree to be more competitive.

However, the agent noticeably underperforms in several key games, such as a less than 50% win rate in breakthrough against the MCTS agent. We note that in prior works, similar agents were able to perform much better in breakthrough (Goldwaser & Thielscher, 2020; Gunawan, Ruan, Thielscher & Narayanan, 2020), likely due to better optimised and more specific neural network structures. The agent also performs very poorly in kalah, which has the largest fluent size of these games.

### 4.3.2 Single player games

While the AlphaZero architecture was initially designed to play two-player, zero-sum games, GGPDeepRL has been adapted to accommodate the more general requirements of GGP. As such, we are able to conduct experiments with single-player games as well, treating them essentially as planning problems. Table 4.3 shows the results of the experiments conducted on single player games knightstour and hamilton. Figure 4.5 shows the average utility of GGPDeepRL on hamilton over various training iterations.

| Game | Agent | | | |
|---|---|---|---|---|
| | random | NN | MCTS | GGPDeepRL |
| knightstour | 0.1482 | 0.39 | 0.6774 | 0.4524 |
| hamilton | 0.2782 | 0 | 0.81 | 0.4382 |

Table 4.3: Comparison of average utility over 100 games on single player games.

In both games, GGPDeepRL is not able to achieve utility values greater than the standard MCTS agent, but it is still able to outperform the random and NN based agents. The NN agent performs noticeably poorly in hamilton, achieving a utility of 0 across all 100 games. As the NN based agent simply selects the action with highest probability, it is likely that as it is consistently selecting the same actions in all games which lead to the poor performance. It is likely the case that the self-play reinforcement learning approach requires more modifications to effectively learn the planning-like setting of single-player games.

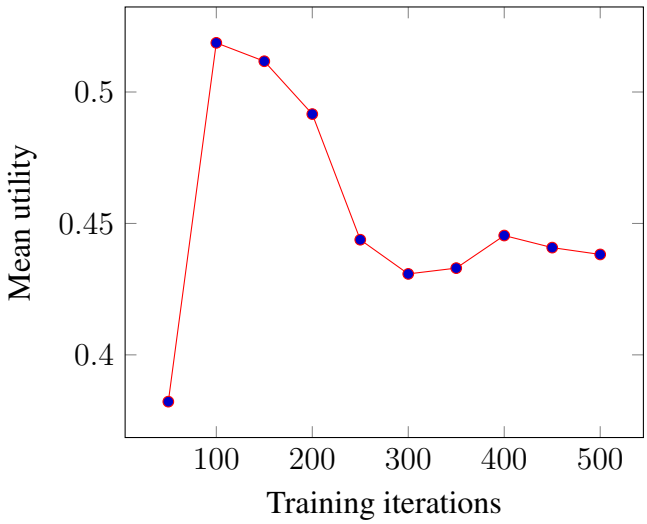



Figure 4.5: Average utility in hamilton with increasing training iterations over 100 games

Figure 4.5 shows the average utility achieved by GGPDeepRL over increasing training iterations, showing a peak at 100 iterations before gradually trending downwards with additional training iterations. The peak mean utility achieved by GGPDeepRL is still lower than the utility achieved by MCTS and the neural network is likely overfitting with the additional training iterations, leading to the lower utility observed.

### 4.3.3 Multi player games

As previously mentioned, GGPDeepRL is able to learn to play games with arbitrary numbers of players. As such, games with more than two players are of particular interest, as they are analogous to multi-agent systems that allow for more complex experiment settings. For example, pacman3p consists of a mixture of cooperation and competition, with the two *ghost* players sharing their utilities against the opponent *pacman* player. On the other hand, babel is a game of pure cooperation - all agents must work together to maximise the same utility. Finally, connectfour3p is a three-player variant of connectfour and retains remains a game of pure competition, albeit with three players instead. Tables 4.4, 4.5 and 4.6 show the results on pacman3p, connectfour3p

and babel respectively.

| Agents | *pacman* | *ghosts* | *pacman* score | *ghosts* score |
|---|---|---|---|---|
| MCTS vs RANDOM | MCTS | RANDOM | 1.96 | 8 |
| | RANDOM | MCTS | 1.12 | 10 |
| GGPDeepRL vs RANDOM | GGPDeepRL | RANDOM | 2.34 | 7 |
| | RANDOM | GGPDeepRL | 0.66 | 10 |
| GGPDeepRL vs MCTS | GGPDeepRL | MCTS | 1.53 | 10 |
| | MCTS | GGPDeepRL | 0.79 | 10 |

Table 4.4: Total goal value of various agents playing 20 games of Pacman3p

GGPDeepRL outperforms the MCTS agent when playing as *pacman*, as it is able to achieve a greater total score when playing against the random agent. Additionally, when playing as the ghosts, GGPZero caused the random agent to collect less pellets in total as well. The game rules however do not provide an explicit incentive for the ghost players to prevent the *pacman* player from collecting pellets. We assume GGPZero's learned strategy was either more efficient in capturing the *pacman* player, or was less likely to make a mistake, letting the *pacman* player continue to collect pellets. In all games where either GGPZero or the MCTS agent played as the ghosts, the *pacman* player was always captured. However, when the random agent played as the ghosts, there were several games (3 games for GGPZero, 2 games for the MCTS agent) where the ghosts were unable to capture the *pacman* player throughout the entire game length. When comparing the performance of GGPZero against the MCTS agent directly, we can see that GGPZero still out performs the MCTS agent when playing as the *pacman* player. However, in all games, both GGPZero and the MCTS agent were able to capture the *pacman* player when they were playing as the ghosts.

As connectfour3p is simply a 3-player variant of standard connectfour, it is not surprising that GGPDeepRL also performs well in this game. In all settings GGPDeepRL reachest the highest average utility, outperforming all benchmark agents by a significant margin. Notably, the NN agent also performs nearly as well as the MCTS agent in all settings, which is contrary to most other games as MCTS generally performs better than

| Agents | *red* utility | *yellow* utility | *blue* utility | mean utility |
|---|---|---|---|---|
| GGPDeepRL | 0.9848 | 0.8181 | 0.8787 | 0.8939 |
| random | 0.3030 | 0.2575 | 0.3030 | 0.2878 |
| GGPDeepRL | 0.6363 | 0.8787 | 0.7878 | 0.7676 |
| NN | 0.3408 | 0.4772 | 0.2348 | 0.3509 |
| GGPDeepRL | 0.8333 | 0.7272 | 0.6212 | 0.7272 |
| MCTS | 0.4469 | 0.3636 | 0.3484 | 0.3863 |
| MCTS | 0.8636 | 0.7121 | 0.7575 | 0.7777 |
| random | 0.4242 | 0.2802 | 0.3332 | 0.3459 |
| MCTS | 0.3939 | 0.5606 | 0.3939 | 0.4494 |
| NN | 0.72725 | 0.553 | 0.2954 | 0.5252 |
| GGPDeepRL | 0.7162 | 0.7241 | 0.5757 | 0.6717 |
| MCTS | 0.4868 | 0.3055 | 0.5800 | 0.4444 |
| NN | 0.6041 | 0.2352 | 0.3780 | 0.3838 |

Table 4.5: Average utility of agents in connectfour3p over 100 games. Underlined agents play both opposing roles.

the NN agent.

| Agents | Average goal value |
|---|---|
| RANDOM | 0.122 |
| MCTS | 0.777 |
| GGPDeepRL (100 iter) | 0.443 |
| GGPDeepRL (500 iter) | 0.638 |

Table 4.6: Results of playing 100 games of Babel. The goal values are shared between all 3 players as it is a cooperative game.

Table 4.6 shows the experimental results on babel. 100 games are played for each agent playing as all three roles in the game. RANDOM gets an average goal value of 0.122, MCTS gets 0.777, GGPDeepRL (with 20 layers and 100 training iterations) gets 0.443, and GGPDeepRL (with 20 layers and 500 training iterations) gets 0.638. We can see GGPDeepRL performs better than RANDOM but worse than MCTS, similar to the results obtained in (Goldwaser & Thielscher, 2020). As this is a game of pure cooperation, it is likely that GGPDeepRL suffers from the same issues as faced in the single-player games, as babel can be seen as a multi-agent planning problem.

### 4.3.4 Neural Network Depth

To investigate the effect of increasing the depth of the neural network (number of hidden layers) used in GGPDeepRL , we train and test 20-layer, 40-layer and 80-layer versions of the agent. We use the single-player game of knightstour and compare the average utility over 100 games to evaluate the effect of layer depth on performance.

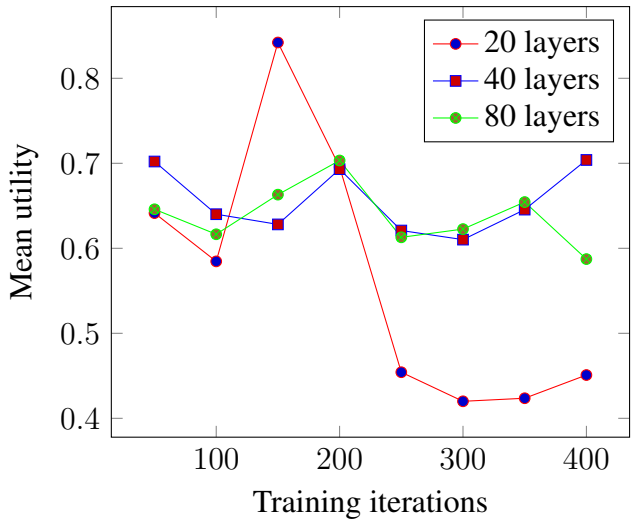
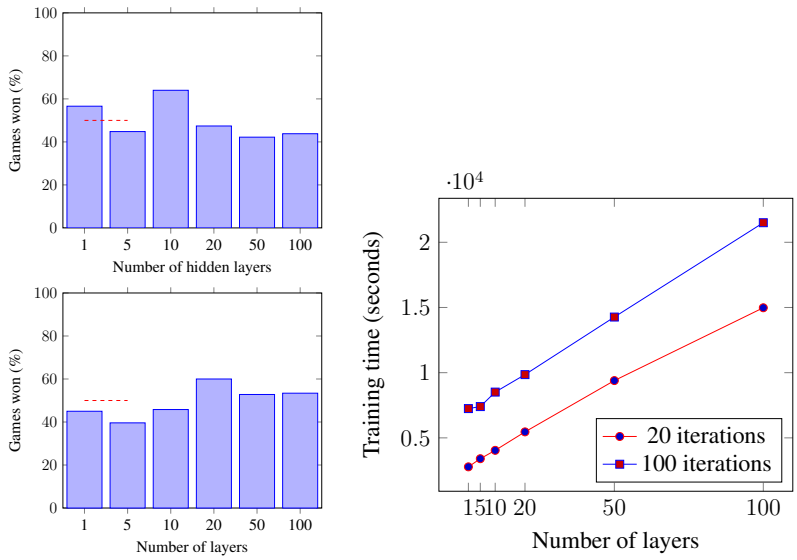



Figure 4.6: Average utility in knightstour with varying number of layers and training iterations over 100 games

Figure 4.6 shows the effect of increased training iterations with the varying layer depths. Interestingly, the model with the least number of layers (20-layer model) achieves the highest average utility across all models. However, this peak utility is reached at 150 iterations and with additional training begins to decrease, eventually performing less effectively than the larger models. In contrast, the two larger models are able to maintain more consistent performance through training iterations, but are unable to reach the peak utility achieved by the 20-layer model. These results suggest that the smaller networks are able to be trained to perform effectively given a limited amount of training, but will overfit rapidly with additional training. On the other hand, the larger networks suffer from inefficiency during training and are unable to be trained as affectively as the smaller models, but are more stable throughout their training iterations.




(a) Top: 20 iterations. Bottom: 100 iterations.

(b) Training time over number of layers.

Figure 4.7: (a): Winning rate of GGPDeepRL with varying number of hidden layers. (b): Training time for GGPDeepRL with varying number of hidden layers.

Additional testing with the two player game of connectfour also presents similar results. Figure 4.7a shows the winning rate by each agent over 500 games (100 games against each other agent). With 20 iterations of training, the agent with 10 layers wins the most amount of games, while with 100 iterations of training, the agent with 20 layers has the best winning rate. Overall, this shows that smaller networks generally perform better with less training, while larger networks perform better with more training. Figure 4.7b shows that increasing the number of hidden layers causes the total training time to increase linearly, and that training 100 iterations takes proportionally more time with a larger model.

### 4.3.5 Convolutional Neural Network Experiments

AlphaGo, AlphaGoZero and AlphaZero all use a convolutional neural network as their estimators as they have strong inductive biases with regards to locality and translation invariance. In the case of Go, Chess and Shogi, all of these games are played on

a grid-like board in which locality and translation invariance are essential features that the neural network must take advantage of. However, not all GGP games are played on a board and even those that are might not have be easily detected. As such, the GGPDeepRL architecture makes use of a fully-connected neural network to ensure compatibility across a wide variety of games. However, we would still like to investigate the effectiveness of convolutional neural networks for general game playing. We compare the performance and training stability of the CNN based agent against the standard FCNN based agent in connectfour.

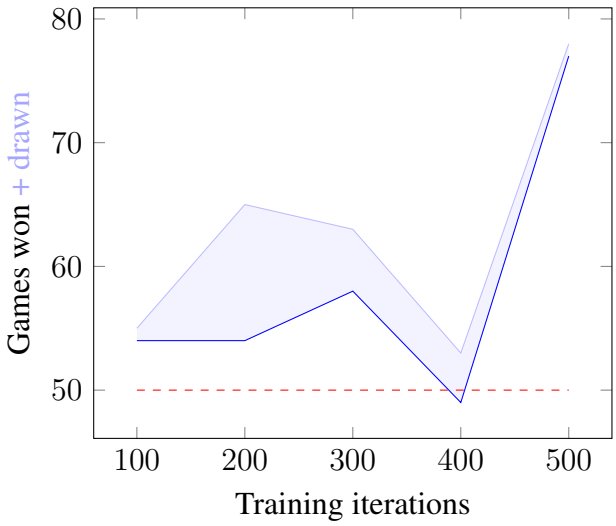



Figure 4.8: Games won by CNN agent versus FCNN agent in 100 games of connectfour. Shaded area includes games that end in a draw.

Figure 4.8 shows the number of games won and drawn by the CNN-based agent against a FCNN-based agent with increasing training iterations. As can be seen, the CNN agent outperforms the FCNN agent throughout all training iterations (when taking into account drawn games as well). Furthermore, at 500 iterations of training, the CNN agent is able to significantly outperform the FCNN agent.

The training loss curves shown in Figure 4.9 highlight the advantages of the CNN-based architecture. The CNN-based architecture is able to both achieve a lower overall training loss value as well as suffering from less instability. Although training loss does

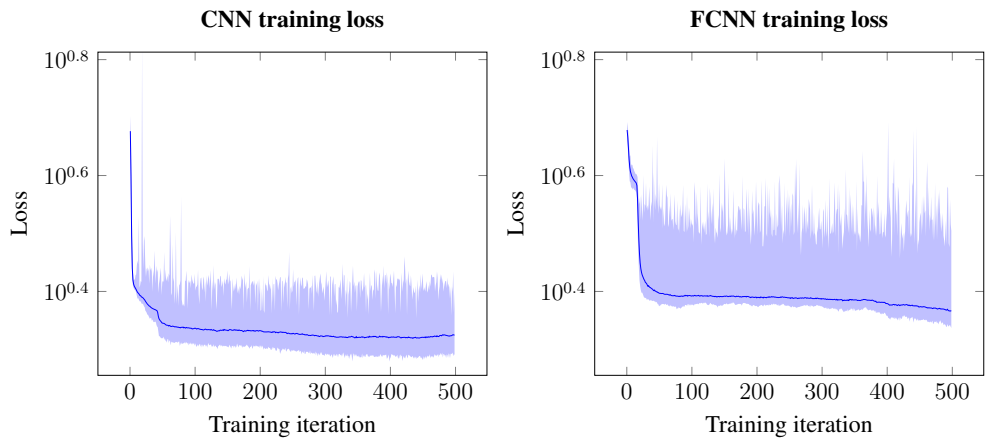



Figure 4.9: Training loss of FCNN and CNN on connectfour.

not directly correlate to overall performance due to the nature of self-play learning, it is still an approximate indicator of overall training efficiency.

### 4.3.6 Effects on Simulation-limited and Time-limited MCTS

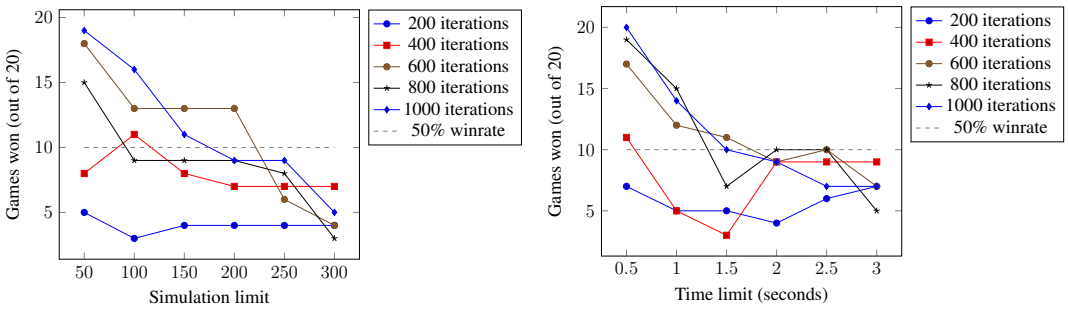
In this section, we compare the effects of simulation and time limitation on the GGP-DeepRL and MCTS during the game play. For the following experiments, we use the game Breakthrough as it has a large state space and branching factor. For GGPDeepRL , we look at 5 versions with training iterations from 200 to 1000.

#### Simulation Limited

In the prior experiments and test games, we use a *simulation limited* variant of MCTS. When selecting an action to play, the agents (both GGPDeepRL and MCTS) perform a limited number of simulations, expanding the search tree. After completing these simulations, the action policy is then generated. We use this variant of MCTS to reduce the impact of the inference engine's efficiency on the performance of the agent, allowing both GGPDeepRL and MCTS agent to search a similar amount of the game states. Increasing the simulation limit allows the agents to search more extensively,

potentially leading to better performance at the cost of slower action selection. This is especially evident with the MCTS agent, as the playout phase requires extensive use of the inference engine. GGPDeepRL is able to circumvent this computationally taxing operation by using the neural network.

Fig. 4.10a shows the GGPDeepRL 's winning rate against MCTS when simulation limit varies from 50 to 300. For GGPDeepRL with 1000 training iterations, it clearly outperforms the MCTS when the simulation limit is low, but loses out when the limit increases. A similar trend can be seen for GGPDeepRL with 600 and 800 iterations. This result is not entirely unexpected as the strategy generated by the MCTS agent begins to approach the Nash equilibrium when the number of simulations performed increases. While for GGPDeepRL with lower training iterations, the winning rate stays relatively flat and is generally worse than the MCTS. This suggests that when GGPDeepRL 's training is insufficient, the MCTS component inside GGPDeepRL is not as effective as the pure MCTS agent due to the limitation of the neural network.



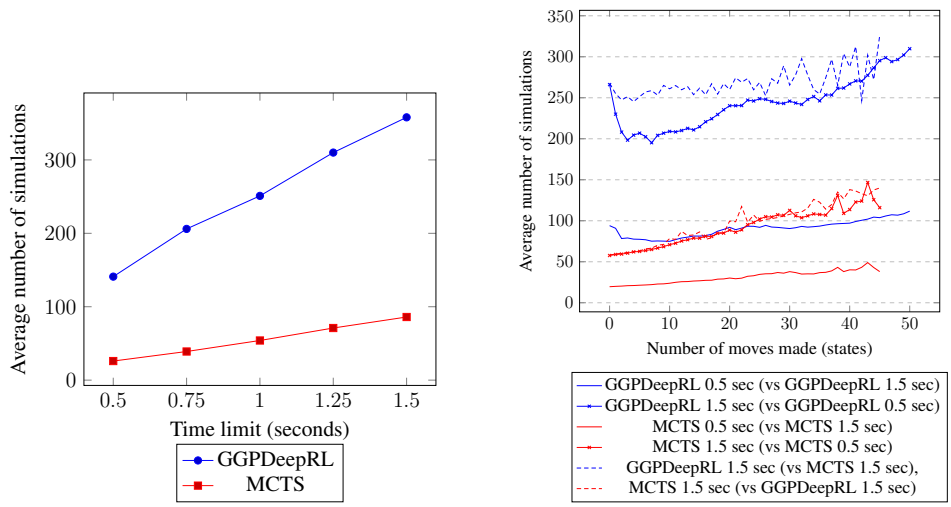

(a) Simulation-limited (b) Time-limited

Figure 4.10: GGPDeepRL winning rate against MCTS with increasing time and simulation limits with varying amounts of training iterations on Breakthrough games.

**Time Limited** In the usual GGP format, actions must be selected under a limited time budget ranging between 10 to 90 seconds, depending on the complexity of the game. As a result, most GGP agents favour using a time-limited variant of MCTS, performing as many simulations as possible in a limited time frame before generating

their action policies. In the next set of experiments, we set time limits from 0.5 seconds to 3 seconds to examine the effects of time limit. Fig. 4.10b presents GGPDeepRL 's winning rate against MCTS under different time limit. For the GGPDeepRL with 1000 training iterations, it significantly outperforms the MCTS agent when the time limit is short, but performs worse when the time limit increases. Similar trend is shown in GGPDeepRL with 600 or 800 iterations, but less so in GGPDeepRL with 200 or 400 iterations. These results are consistent with the simulation-limited case in Fig. 4.10a.

We further examined the simulation numbers under different time limits and during different stage of the game play. Fig. 4.11a shows that as the time limit increases, the average number of simulations of the MCTS agent does not increase as rapidly as that of the GGPDeepRL . Despite that, the MCTS agent still performs better when given a longer time limit. Note that GGPDeepRL with different training iterations performs similar numbers of simulations within the same time limit. This is due to the fact that the neural network can be seen as an $O(1)$ operation regardless of the amount of training; there is no trade-off with regard to efficiency when using a more trained agent.




(a) Average simulations per time limit

(b) Average simulations over the course of a game

Figure 4.11: Average number of simulations completed by agents

In the time limited scenario, another aspect to consider is the varying amount of time

required to perform search at different stage of the game. Although in Fig. 4.11a we see that the MCTS agent has significantly less *average simulations* throughout the course of a game, we see in Fig. 4.11b that as the game progresses, the average number of simulations tends to increase for all agents. This is likely due to the search more likely visiting terminal states as it progresses through the game, causing simulations to finish earlier. In the case of the MCTS agent, this early stopping by finding terminal states presents a significant reduction in simulation time, as it does not require the search to enter the playout phase.

To conclude, in both the simulation-limited and time-limited settings, we see that GGPDeepRL with enough training significantly outperforms the standard MCTS agent when given a strict simulation or time limit. However, this difference in performance diminishes and eventually is reversed once these limits are large enough. From these observations, we can view the trained neural network as a form of *compression*, replacing the computationally expensive playout phase with a neural network. The quality of this compression is in part affected by the amount of training provided to the agent but also by the architecture of the neural network itself. As we saw in subsection 4.3.5, the convolutional neural network had better and more stable performance than the fully connected neural network when given sufficient training. The final comment is that the neural network with limited training can also be a limiting factor for the MCTS component in the GGPDeepRL , as when giving a longer time limit, the pure MCTS agent can always outperform the GGPDeepRL as in our experiments.

## 4.4 Conclusion

We described GGPDeepRL , an AlphaZero-style self-play reinforcement learning agent and implementation, designed to learn games in the GGP setting. While the architecture was inspired by the recent Generalised AlphaZero (Goldwaser & Thielscher, 2020), the

main differences are in the choice of GGP reasoner and the neural network construction; furthermore we carried out systematic experimental evaluation of various aspects of the use of deep reinforcement learning for GGP. Our main contribution is in the exploration of this learning architecture: confirming its feasibility, the impact of neural network depth, types of network, and simulation vs. time limitation on training. The experiments with network architectures have shown that the agent scales well in terms of both efficiency and effectiveness.

To further improve the overall performance of the agent, the current implementation could be improved by parallelising several components. Firstly, training could be parallelised to allow for multiple games to be played concurrently, allowing for either faster training times or more extensive training within the same timescale. Secondly, the MCTS itself could be parallelised (G. M.-B. Chaslot, Winands & van den Herik, 2008) — however, the selection of which parallelisation method to use must be investigated within the context of the agent's architecture.

The Babel results show that the agent is capable of learning and playing games that are not zero-sum, turn-taking or two-player. This means that GGPZero is able to generalise beyond the games that AlphaZero was able to play. However, we also see that a pure MCTS agent still outperforms GGPDeepRL in this game. Further investigation is required into the nature of the training method itself, as it may have limitations when learning to play more general games.

A key observation across all experiments conducted is the importance of the neural network architecture used by the agent. Despite having modified the type of neural network used to accommodate the general nature of games found in GGP, it still does not overcome a major limitation of an AlphaZero-style agent: the requirement of a new, bespoke neural network for each game that must be trained from scratch. Although some type of ensemble mechanism to train multiple networks for multiple games may alleviate some of this, it would still mean the agent is still significantly less general

than a non-learning counterpart. A truly general learning agent would be able to learn multiple games, transfer learned knowledge across different games and play new games relatively well with only prior learned knowledge. Implementing such an agent would require novel game state representations and neural network architectures that can effectively capture the general features of various games. In the following chapter, we will investigate the possibility of such an approach. However, we will use it for the reasoning task instead, as applying neural networks and learning to this task was not possible prior to such an architecture. Furthermore, we would like to avoid having the other components of deep reinforcement learning affect the experimental results and instead prefer to focus on just the neural network itself.

# Chapter 5

# A Graph Neural Network Reasoner for GDL

## 5.1 Introduction

In order to play games in GGP, the players must parse the game rules using logic-based inferences to obtain game states and expand the game tree to search for an optimal move. These inferences are typically done through Prolog or PropNet (propositional networks) (Schkufza et al., 2008) implementations. The recent accomplishments of AlphaGo (D. Silver et al., 2016) and AlphaGo Zero (D. Silver, Schrittwieser et al., 2017) use Monte-Carlo Tree Search (MCTS), deep neural networks and learning through self-play to achieve super-human performance in Go. The generality of this method is extended in AlphaZero(D. Silver, Hubert et al., 2017) to also allow the agent to play Chess and Shogi effectively. However, AlphaZero is limited in that it does not automatically extend to play other games without additional human information and the game types are always assumed to be two-player, turn-taking and zero-sum. In the domain of GGP, an agent must be able to account for games with any number of players, simultaneous action and non-zero-sum, without the aid of human information. GDL

serves the purpose of representing such games in a flexible way.

In this chapter, we propose methods for a neural network based reasoner that is able to learn logical inferences for GDL. As GDL allows flexible representation of various games, the reasoner has to handle game states with various sizes and features; e.g., a state in Tic-Tac-Toe is quite different to a state in Connect Four. Typical neural network architectures such as multilayer perceptrons and convolutional networks would not be able to capture the general nature of the different state features of various games. This is due to the incompatibility of input spaces when using vector, matrix or tensor-based representations to encode state information. These representations only implicitly contains information about the game rules through their structure. The size of such representations would differ between different games and using padding would introduce additional inconsistencies between games. The relative positioning of elements would also be inconsistent across different games as their positions in such representations are dependent on the games themselves. As such, changing either the size through padding or repositioning the elements in these representations implicitly affects the rules of the game itself. As an example, the convolutional neural networks used in AlphaZero were game-specific and had to be trained individually for each game as they all required different input sizes corresponding to the positional information present in each game - the Chess network used stacks of 8×8 matrices while the Go network used 19×19 matrices. The neural networks used in earlier GGP works (Goldwaser & Thielscher, 2020; Gunawan et al., 2020) also suffered from similar issues, as each game would have a different set of predicates used to define the state of a game, leading to input vectors of varying length and positional encoding.

To overcome this, we propose graph-based representations of GDL rules and game states, using a Graph Neural Network (GNN) based architecture for inference. The graph-based representations explicitly encode the game rules into the state representation, allowing for the GNN to infer across varying games while maintaining consistency

and generality. This combination provides an avenue for the development and implementation of GNN based reasoners that are able to learn to infer the rules of general games. We present three key contributions: (i) a general, game-agnostic graph-based representation for game states described in GDL, (ii) methods for generating samples and datasets to frame the GDL inference task as a machine learning problem and (iii) a GNN based *neural reasoner* that is able to learn and infer various game states with high accuracy. The neural reasoner is capable of transferring learned knowledge across games as well as learn multiple games simultaneously in a mixed training scheme.

The rest of the sections are organised as follows: Section 5.2 gives an overview of the different components of our neural reasoner. Section 5.3 describes a general representation for game states in GGP, and in particular we propose *instantiated rule graphs*. Section 5.4 introduces our proposed GNN architecture that is able to handle the instantiated rule graphs and outputs the predicted legal actions and next fluents. Section 5.5 presents and discusses the results of our neural reasoner across a variety of games as well as a mixture of games. The neural reasoner is evaluated across a set of games to investigate its generalisation capability. In particular, we find that our reasoner can achieve 100% accuracy in several games and over 80% for the majority when trained and evaluated on games individually. The game-agnostic nature of the state representations combined with the GNN architecture allows the neural reasoner to transfer learned knowledge from one game to another. We demonstrate this with experimental results showing the neural reasoner's transfer learning and mixed learning capabilities. Finally, Section 5.6 concludes by summarising the results of our experiments and discusses possible future directions for research.

## 5.2 Overview of the Neural Reasoner

We first show the inference tasks in GGP that our neural reasoner learns to approximate and then present an overview of the different components in the neural reasoner. Games in the GGP setting are described using Game Description Language (GDL), a logical programming language based on Datalog that can describe game states and mechanics with logical rules.

Figure 5.1 shows an example fragment of the description for game Tic-Tac-Toe. The left rule states that in the next state, `(cell 1 1 o)` is true if `oplayer` does action `(mark 1 1)` and `(cell 1 1 b)` is true in the current state. The right rule states that it is legal for `oplayer` to take the action `(mark 1 1)` if `(cell 1 1 b)` and `(control oplayer)` are true in the current state. The terms `(next ...)`, `(legal ...)` are examples of keyword terms[1], while `(cell ...)`, `(mark ...)` are user-defined, non-keyword terms. The terms `1`, `o` and `b` are also user-defined terms as numbers and constants are not pre-defined in GDL. A full GDL description of a game will consist of many other rules to define the full dynamics of the game.

```
(<= (next (cell 1 1 o))          (<= (legal oplayer (mark 1 1))
    (does oplayer (mark 1 1))        (true (cell 1 1 b))
    (true (cell 1 1 b)))             (true (control oplayer)))
```

Figure 5.1: Fragment of GDL description of Tic-Tac-Toe.

A game state described with GDL consists of a set of dynamic predicates called *fluents*, such as `(cell 1 1 o)`, `(cell 1 1 b)` `(control oplayer)`. A subsequent state (next-state) can be logically derived by the combination of rules, fluents present in the current state and the actions made by the players. Suppose we have a game $G$, defined as a set of rules in GDL, two of the key inference tasks for a GDL reasoner are to find out *(1) what are the legal actions for the players at the current state and (2) what is the next state if the agents make a joint move*. Task (1) can be formalized

[1]A full list of all keywords defined in GDL is provided in Section 5.3.1

as the following: to find out for each player $i$, its legal move $m$ at the current state $s$, as in a logical representation $G \wedge s \vDash legal(i, m)$. There can be multiple legal actions in a state. Task (2) can be formalized as the following: to find out all the fluent $f$ that holds in the next state, given each player $i \in [1, k]$ doing a move $m_i$ at the current state $s$, as in a logical representation $G \wedge s \wedge does(1, m_1) \wedge ... \wedge does(k, m_k) \vDash next(f)$.

Figure 5.2 gives an overview of the different components in our neural reasoner for GGP. We start with a game described as a set of rules in GDL. It is converted into a rule graph, which is built upon the dependency of the fluents or other predicates in such rules. The rule graph is then combined with a game state into an instantiated rule graph (IRG). We apply a node-vector embedding to the nodes of the IRG to prepare it as input of the Graph Neural Network component. The output of the GNN component is a rule graph with node probabilities. Node masking is then applied based on node type to extract the probabilities of the `legal` and `next` nodes, using information from the input rule graph. During training, the output probabilities are compared to the ground truth training targets to calculate the error for backpropagation. Finally, the nodes with output probabilities greater than a threshold are selected as the legal actions and next fluents. Algorithm 2 provides pseudocode describing this process.

**Algorithm 2** Neural Reasoner

```
1:  function InferState(Game G, state S, actions a, threshold t_next, threshold t_legal)
2:      Rule graph R ← transformToRuleGraph(G)
3:      Instantiated rule graph I ← InstantiateRuleGraph(R, S, a)
4:      X ← getVectorEmbedding(I)
5:      A ← getAdjacencyMatrix(I)
6:      X' ← GNN(X, A)                    ▷ X' is the output with node probabilities
7:      next ← ∅, legal ← ∅
8:      for all x ∈ X' do
9:          if L(x) = next and x > t_next then        ▷ L(x) is the label for node x
10:             next ← next ∪ {x}
11:         if L(x) = legal and x > t_legal then
12:             legal ← legal ∪ {x}
13:     return next, legal
```

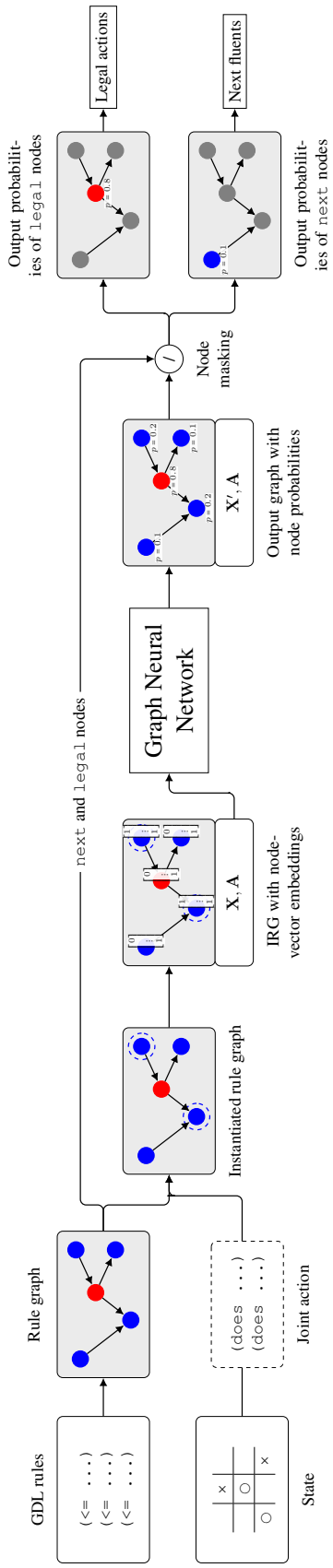



Figure 5.2: Overview of the Neural Reasoner

The training samples, generated by a logical reasoner, are in the form of the pairs $((G, s), S_{\text{legal}})$ for Task (1) and $((G, s, Does), S_{\text{next}})$ for Task (2), where $G$ is a game, $s$ the current state, $S_{\text{legal}}$ the set of legal actions, $Does$ the joint action set, and $S_{\text{next}}$ the set of fluents true in the next state. A split of the training samples are used for validation as well as selecting the threshold values $t_{next}$ and $t_{legal}$. Once the graph neural network is trained and the threshold values selected, we then use the network to predict the legal actions and next fluents of any given state. In the next two sections, we will provide more details on these components.

## 5.3 Game State Representations

A key element of applying neural networks to games is finding a state representation that allows for effective feature extraction, while also being computationally efficient. Typically, these representations consists of a vector, matrix or tensor based structure, depending on the type of neural network architecture used. For example, in Chess or Go, the game board can be directly translated into a matrix, with elements representing the presence of different pieces on the board. These matrices can then be used directly as input to a convolutional neural network, as the grid-structure of the matrix retains the positional information of the various pieces. Although they are effective and efficient, these vector or matrix based representations are “fixed" in size for each game and are not compatible across different games.

As different games described in GDL may have varying sets of possible fluents, a simple vector or matrix is not capable of representing the general features of various game states. Consider the games Tic-Tac-Toe and Connect-Four: one could represent various GDL states by constructing a one-hot vector representing all possible fluents that could be present in a state, such as `cell(1, 1, o)`, `cell(1, 2, x)` and so on. However, as Tic-Tac-Toe and Connect-Four have different numbers of fluents

(28 and 127 respectively), these vectors would have different sizes. Likewise, if we were to instead represent a state by mapping its board positions to a matrix, these matrices would also have different sizes. A multilayer perceptron or convolutional neural network would have a fixed size for its input layer and would not be able to take both Tic-Tac-Toe and Connect-Four states as input without additional preprocessing or padding, which may introduce unintended biases. This means a different neural network needs to be trained for each game separately.

In this section, we propose a more general “game-agnostic" representation that suits the domain of GGP better. This representation will be able to fully capture the features present in states, while still allowing for states of various games to be valid inputs for the same neural network. The basis of this representation is a *rule graph*, a graph-based representation of game rules first introduced in (Kuhlmann & Stone, 2007) and further extended in (Schiffel, 2010), initially used to identify similarity in game descriptions. The rule graph captures the relational inductive biases present in the game rules. We first give the details of rule graphs in section 5.3.1 and then propose to extend the rule graphs to be a general state representation in section 5.3.2.

### 5.3.1 Rule Graphs

Rule graphs are coloured, directed graphs consisting of keyword nodes, expression nodes, symbol nodes and symbol argument nodes. *Keyword nodes* represent logical and relational sentences consisting of a predefined GDL keyword such as `legal`, `next`, `does`, etc. and each keyword corresponds to its own node type. *Expression nodes* represent sentences consisting of non-keyword nodes, i.e., custom defined predicates or functions declared specifically for the game. In the fragment in Figure 5.1, the expressions `mark` and `cell` are examples of non-keyword expressions. *Symbol nodes* are used to identify which expressions are the same throughout the description without

keeping explicit or specific lexical labels. This is done by drawing an edge from symbol nodes to each expression which is an instance of said symbol. Finally, *symbol argument nodes* are used to identify the position of arguments in an expression, encoding the positional information of expressions into the rule graph. Similarly to symbol nodes, an edge is drawn between the symbol argument node to each instance of an argument given its position. Additionally, variable nodes can also be included, however for our purposes the game descriptions are initially grounded which instantiates all variables. Edges are drawn between expressions and their arguments, constructing a syntax tree. The GDL keywords $\mathcal{K}$ are defined as follows:

$$\mathcal{K} = \left\{ \begin{array}{l} \Leftarrow, \texttt{not}, \texttt{or}, \texttt{distinct}, \texttt{does}, \\ \texttt{goal}, \texttt{init}, \texttt{legal}, \texttt{next}, \texttt{role}, \\ \texttt{terminal}, \texttt{true}, \texttt{input}, \texttt{base} \end{array} \right\}$$

**Definition 5.3.1 (Rule Graph for a game description G).**
*Given a game description $G$, the rule graph of game $G$ is a coloured, directed graph $R_G = (V, E, T)$. $V$ is the set of nodes, $E$ is the set of edges and $T : V \rightarrow \mathcal{K} \cup \{\texttt{expression}, \texttt{variable}, \texttt{symbol}_{\texttt{const}}, \texttt{symbol}_{\texttt{arg}}, \texttt{symbol}_{\texttt{var}}\}$ is a colouring function that denotes the node type of each node. A lexical mapping $F_G : V \rightarrow \Sigma$ is induced from the description of the game where $\Sigma$ is the set of ground terms present in $G$. $V$, $E$ and $T$ are defined as follows:*

1. ***Implication nodes:*** *For rule $\texttt{h} \Leftarrow \texttt{b}_1 \wedge \ldots \wedge \texttt{b}_\texttt{n}$ in $G$, we construct nodes*[2] *$v$, $h$, $b_1, \ldots, b_n$ such that $F_G(v) = \texttt{h} \Leftarrow \texttt{b}_1 \wedge \ldots \wedge \texttt{b}_\texttt{n}$, $F_G(h) = \texttt{h}$, $F_G(b_1) = \texttt{b}_1, \ldots, F_G(b_n) = \texttt{b}_\texttt{n}$, and the edges $(v, h)$, $(v, b_1), \ldots, (v, b_n)$, $(h, b_1), \ldots, (h, b_n) \in E$. Node type $T(v) = \Leftarrow$ and the types of $h$, $b_1, \ldots, b_n$ will be given below.*

[2]We use italic font to indicate a node in a rule graph, e.g., $h$, $b_1$, in contrast to the typewriter font used to indicate a term in a rule, e.g., $\texttt{h}, \texttt{b}_1$.

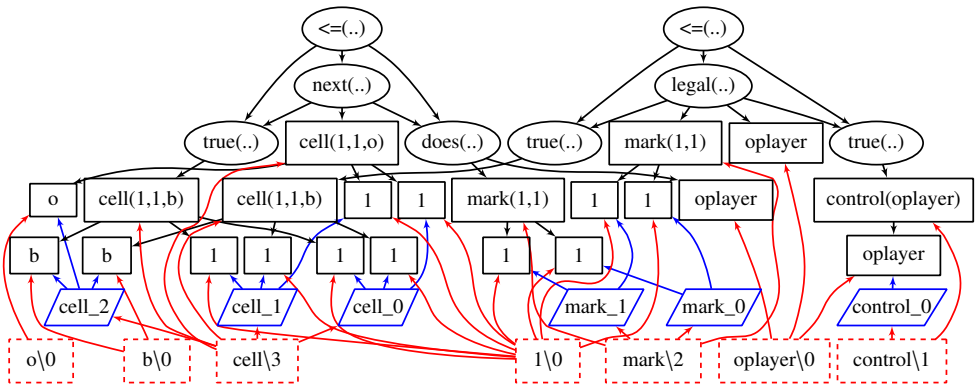



Figure 5.3: Rule graph of the fragment in Figure 5.1. Lexicographic information such as predicate names are not stored and only shown here to aid in visualisation.

2. ***Keyword nodes:*** *If a term* $\mathtt{k}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ *occurs in a rule in* $G$ *and* $\mathtt{k} \in \mathcal{K}$ *we construct keyword node* $k \in V$ *if it does not already exist and construct nodes* $t_1, \ldots, t_n$. *We construct edges* $(k, t_1), \ldots, (k, t_n)$, $(t_1, t_2), \ldots, (t_{n-1}, t_n) \in E$. *We assign node type* $T(k) = \mathtt{k}$ *and lexical mappings* $F_G(t_1) = \mathtt{t}_1, \ldots, F_G(t_n) = \mathtt{t}_n$. *The node types of* $t_1, \ldots, t_n$ *are given by applying this definition recursively.*

3. ***Symbol nodes:*** *For every* $n$*-ary relation and function symbol* $\mathtt{p} \notin \mathcal{K}$ *in* $G$, *we construct symbol nodes* $s$, $s^1, \ldots, s^n \in V$, *construct edges* $(s, s^1), \ldots, (s, s^n) \in E$, *assign node types* $T(s) = \mathtt{symbol}_{\mathtt{const}}$, $T(s^i) = \mathtt{symbol}_{\mathtt{arg}}$ *and assign lexical mappings* $F_G(s) = \mathtt{p}\backslash n$ *and* $F_G(s^i) = \mathtt{p\_}i$.

4. ***Expression nodes:*** *If a term* $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ *occurs in a rule in* $G$ *and* $\mathtt{p} \notin \mathcal{K}$, *we construct expression node* $p$ *if it does not already exist and construct nodes* $t_1, \ldots, t_n$. *With symbol nodes* $s, s^1, \ldots, s^n \in V$ *where* $F_G(s) = \mathtt{p}\backslash\mathtt{n}$ *and* $F_G(s^i) = \mathtt{p\_}i$, *we construct edges* $(p, t_1), \ldots, (p, t_n)$, $(s, p)$, $(s^1, t_1), \ldots, (s^n, t_n) \in E$. *We assign node types* $T(p) = \mathtt{expression}$ *and lexical mappings* $F_G(p) = \mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$, $F_G(t_1) = \mathtt{t}_1, \ldots, F_G(t_n) = \mathtt{t}_n$. *The node types of* $t_1, \ldots, t_n$ *are given by applying this definition recursively.*

While not a part of the rule graph itself, a lexical mapping $F_G$ is induced from the

game description when transformed. This mapping allows us to retain a connection between the nodes in the rule graph and the terms used in the description, as the rule graph itself does not store this information within the nodes themselves. When used in downstream tasks, the rule graph will not contain any lexical information to ensure generality across different game descriptions. However, for some tasks, the lexical information is still required. While the rule graph representation itself is still uniquely identifiable by itself (while retaining high levels of generality), the lexical mapping allows us to specify exactly what terms are being used when integrated into a larger GGP system as a whole.

Variable nodes are also defined in a similar manner as expression nodes, but are omitted for clarity as we apply a grounding to the game descriptions prior to generating a rule graph which removes all variables. Figure 5.3 shows the rule graph generated from the rule fragment presented in Figure 5.1. Oval nodes are keyword nodes. Rectangular nodes are expression nodes. Red nodes are symbol nodes. Blue nodes are symbol argument nodes. Lexical labels are only for visualisation and are not retained in the actual rule graph. To generate a rule graph from a game description $G$, we follow the method in (Schiffel, 2010):

- Ground all rules to eliminate all variables in $G$.
- For each keyword and expression term in a rule in $G$, create a node in the graph. E.g.,
  - The rule graph in Figure 5.3 consists of nodes for each keyword and expression present in the fragment in Figure 5.1: the keyword `legal` is represented by an oval keyword node and the expression `(control oplayer)` is represented by a rectangular expression node.
- Add an edge between each term's node to all the term's argument nodes. For the

backwards implication node, add edges from the head node to the body nodes. E.g.,

  - The node `legal(oplayer,mark(1,1))` has edges to its arguments, nodes `oplayer` and `mark(1,1)`. Aditionally, as it is the head argument of a backwards implication node, it has edges to the body nodes `true(control(oplayer))` and `true(cell(1,1,b))`.

- For each unique non-keyword expression symbol in the game description, create a symbol node and an edge to each node that represents an instance of that expression being used. E.g.,

  - As the expression `mark` is used in both rules, the red `mark\2` symbol node has edges to each instance of the expression.

- For each symbol node's expression with arity $n$, create $n$ symbol argument nodes, representing each possible argument's position. Add an edge from the symbol node to each corresponding symbol argument node. Add an edge to each node that represents an argument with the corresponding symbol argument node according to its position. E.g.,

  - The blue `mark\2` node has two symbol argument nodes: `mark_0` and `mark_1`.

To ensure GGP agents are not simply referencing a stored database of collected games and strategies, it is common to apply a *scrambling* to the game description prior to testing. Scrambling a game description reorders the rules and the positioning of arguments within a rule without changing the semantic meaning of the description. Scrambling also applies a renaming to the description, changing the identifiers used for predicates, variables and functions to new identifiers in a consistent manner that

also retains the semantic meaning of the description. Essentially, scrambling obscures the original description's syntax to ensure that agents do not simply rely on rote memorisation while still describing the same game and retaining its semantic meaning.

As the rule graph $R$ itself only retains the relations and positions of the various expressions without storing lexical information, the representation is isomorphic to scrambling. This is due to the general nature of the syntax tree-like structure of a rule graph retaining the syntactic information and the symbol nodes retaining the symbolic information of shared expression identifiers. Two different descriptions of the same game that have their predicate and function names changed would have the same rule graph representation. In other words, for a game $G$ and its rule graph $R$, a scrambling of game description $G_s$ would induce a new rule graph $R_s$ that is isomorphic to $R$ but with different lexical mappings $F_G \neq F_{G_s}$ as the positions and expression names may have been changed.

### Grounding

In typical logic-based inference methods for GDL, finding variable substitutions of $legal(i, m)$ and $next(f)$ for a given state gives the `legal` actions and `next` fluents. However, for a neural network based approach, the reasoner instead provides probabilities over the set of possible fluents or actions. As a result, the sets of all fluents and actions possible in the game must be determined prior to inference. Note that this refers to all fluents and actions possible in a game rather than a specific state and as such per-state inference is not needed to collect them.

Although rule graphs as described in prior works are able to represent the variables present in game descriptions, our current method requires the grounding of all variables, removing them from the rule graph representation. This process replaces variables present in the rules with a grounded instance, duplicating the rules when multiple groundings are possible. The grounded instances of the `legal` actions and `next`

```
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
```

```
(<= (legal oplayer (mark 1 1))
    (true (cell 1 1 b))
    (true (control oplayer)))
...
(<= (legal xplayer (mark 3 3))
    (true (cell 3 3 b))
    (true (control xplayer)))
```

Figure 5.4: Left: Standard definition of a rule in Tic-Tac-Toe. Right: Two instances of the grounded rule

fluents correspond to the output nodes - effectively the set of possible fluents and actions for the given game description. The graph neural network then provides probabilities over these output nodes to approximate the inference tasks.

We use a Prolog-based method with tabling (Vittaut & Méhat, 2014b) to efficiently ground the game descriptions. Additionally, the grounding process also instantiates all possible fluents and legal actions present in the game, which is used later in the dataset generation process. Figure 5.4 shows an example of an ungrounded fragment of a GDL description and its grounded counterparts.

### 5.3.2 Instantiated Rule Graphs

Although GDL provides a representation for rules that are general across different games, it does not provide a general *state* representation. Likewise, the rule graphs discussed in the previous section are still only a representation of game rules and are not a general representation of game states. To overcome this limitation, we present *instantiated rule graphs* (IRG), a general state representation generated by instantiating the rule graph of a game, localising it to a specific state by providing a unique node labelling for each state.

**Definition 5.3.2 (Instantiated Rule Graph).**

*Given game $G$, its rule graph $R = (V, E, T)$ and a state $S = \{f_1, f_2, \ldots, f_n\}$ consisting of fluents $f_i$, an instantiated rule graph of state $S$ is a graph $I = (V, E, T, L_S)$ where $L_S : V \rightarrow \{false, true\}$ is a labelling function based on state $S$.*

The labelling function $L_S(v)$ defines an additional colouring on the rule graph, instantiating it to a single, unique state. This labelling should be unambiguous between states and provide the graph representation with sufficient information for reasoning tasks. Besides these restrictions, the labelling function itself is relatively flexible and several can be defined. For example, we can consider two possible functions: a *minimal labelling* and a *subtree-complete labelling*.

**Definition 5.3.3** (**Labelling functions for IRG**).
*For node $n$ in rule graph $R$ of game $G$, the induced lexical mapping $F_G$ and current state $S$, we define the functions as follows:*

- ***Minimal labelling:*** *If node $n$ is node type $T(n)$ = `expression` and $F_G(n) \in S$, then $L_S(n) = true$. For all other nodes $n_o \neq n$, $L_S(n_o) = false$.*

- ***Subtree-complete labelling:*** *If node $n$ is node type $T(n)$ = `expression` and $F_G(n) \in S$ and parent node $n_p$ is node type $T(n_p)$ = `true` with edge $(n_p, n) \in E$, then $L_S(n) = true$, $L_S(n_p) = true$ and $L_S(n_c) = true$ for all children nodes given $(n, n_c) \in E$. For all other nodes $n_o \neq n$, $L_S(n_o) = false$.*

As seen above, the minimal labelling only labels the expressions present in a rule, while the subtree-complete labelling also labels the corresponding `true` node as well as all children expression nodes, labelling the entire subtree rooted at a `true` node. The subtree-complete labelling colours a more extensive set of relevant nodes, potentially allowing the graph neural network to skip several message-passing steps. The minimal labelling is slightly more efficient, not requiring a search of the parent and children of a expression node. These labelling functions are similarly extended to also label `does` nodes and action expressions for `next` fluent prediction.

Figure 5.5 shows the rule graph from Figure 5.3 instantiated to the given state $S$ = $\{\texttt{cell}(1, 1, \texttt{b})\}$ using both minimal and subtree-complete labelling. Nodes labelled

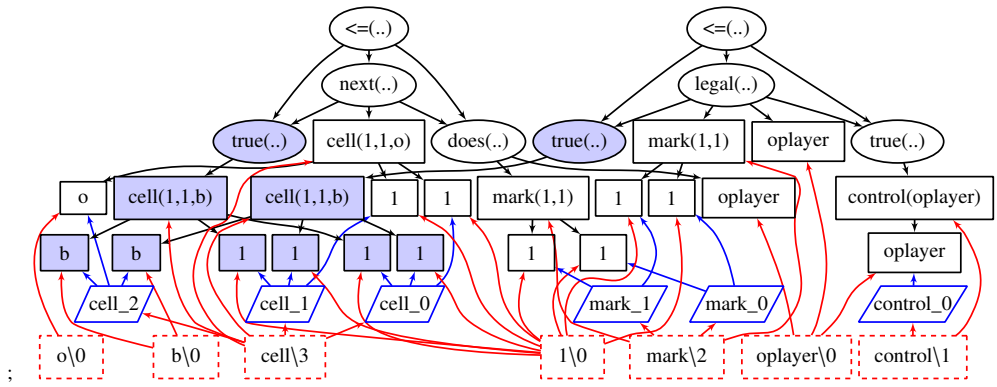



Figure 5.5: An example of an instantiated rule graph for state $S = \{\texttt{cell}(1,1,\texttt{b})\}$. With minimal labelling, only the nodes with a yellow shading would be labelled. With subtree-complete labelling, all the nodes with a blue shading would be labelled. The $\texttt{cell}(1,1,\texttt{b})$ expression nodes are labelled with both labelling functions.

$true$ are shaded in blue. All other nodes are labelled $false$. A benefit of the instantiated rule graphs is that any modifications or improvements can be made to the labelling function without loss of generality across all games. While the current labelling functions serve our purposes, it is possible that additional modifications to them or entirely new labelling functions could provide better performance. The optimal labelling function for GDL inference is still not known and is an open problem.

## 5.4 Machine Learning Framework

This section presents the machine learning framework for GDL inference, including the neural network architecture used with IRGs and dataset generation for training. Firstly we introduce our strategy of embedding the node features into a vector. Secondly, we provide details of the graph neural network based architecture. Thirdly, we present a general method of generating datasets for different games. Finally, we provide details on the training of the neural network itself.

| Index | Embedding |
|---|---|
| 1 | True in state (label) |
| 2 | Constant symbol |
| 3 | Variable symbol |
| 4 | Expression |
| 5 | Variable |

(a) Label and non-keywords

| Index | Embedding | Index | Embedding |
|---|---|---|---|
| 6 | `<=` | 13 | `legal` |
| 7 | `not` | 14 | `next` |
| 8 | `or` | 15 | `role` |
| 9 | `distinct` | 16 | `terminal` |
| 10 | `does` | 17 | `true` |
| 11 | `goal` | 18 | `input` |
| 12 | `init` | 19 | `base` |

(b) Keywords

Table 5.1: Vector embedding values

### 5.4.1 Node-Vector Embeddings

To prepare the instantiated rule graph as input for a neural network, we convert the labelling and the node types into a vector embedding. For node $n \in V$ in instantiated rule graph $I = (V, E, T, L)$, we construct a vector embedding $\vec{v}^n \in \{0, 1\}^{19}$. This embedding provides a general fixed length vector that can represent without ambiguity the various types of nodes and labelling present in an instantiated rule graph. The first value of the vector directly corresponds to the labelling function, with $true$ and $false$ represented by values $1$ and $0$ directly. The rest of the vector acts as a one-hot encoding representing the type of node, with the first four corresponding to the non-keyword node types and the rest corresponding to the keywords present in GDL. Table 5.1 shows the node feature that each value $\vec{V}_{1...19}$ corresponds to.

Figure 5.6 shows an example of a vector embedding of the `cell(1, 1, b)` nodes in the instantiated rule graph from Figure 5.5. Note that as the node is a expression node, the value of $\vec{v}^n_4 = 1$. The node `true(cell(1, 1, b))` is a `true` keyword node, resulting in the value of $\vec{v}^n_{17} = 1$. Both nodes are labelled $true$ by the subtree-complete labelling function, so both vectors will have the value $\vec{v}^n_1 = 1$.

The vector embeddings of all nodes in the graph are then stacked to create node feature matrix $\mathbf{X} \in \{0, 1\}^{|V| \times 19}$. Alongside the adjacency matrix of the graph $\mathbf{A}$, the node feature matrix $\mathbf{X}$ is used as the input for the graph neural network described in the next section.

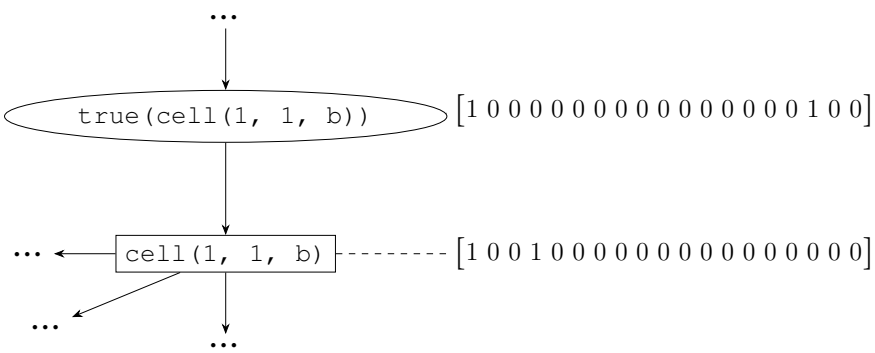



Figure 5.6: Example vector embedding of two nodes of the instantiated rule graph from Figure 5.5 using subtree-complete labelling.

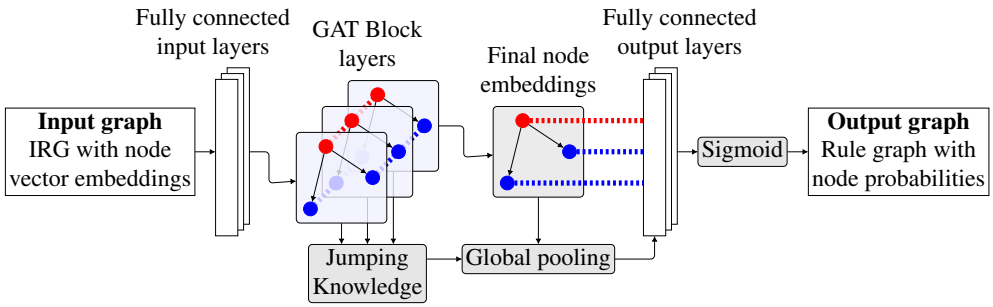



Figure 5.7: Architecture of Graph Neural Network used in the Neural Reasoner

### 5.4.2 Graph Neural Network Architecture

As the game state representation we have discussed in the previous section is defined as a graph, we use a graph neural network. The overall architecture of our GNN, shown in Figure 5.7, consists of three distinct layers: an initial set of fully connected input embedding layers, a set of 3 GAT Block layers and finally a set of fully connected output layers. The GAT Blocks consist of two pairs of Graph Attention Networks (GAT) (Veličković et al., 2018) acting as bi-directional edge layers, a skip connection and ReLU nonlinearity.

An input IRG with node vector embeddings is first passed through a set of fully connected input layers, attending only to individual node embeddings. Then, the graph is passed through a set of GAT Blocks, propagating messages to update the node embeddings. After the last GAT Block, final node embeddings are generated.

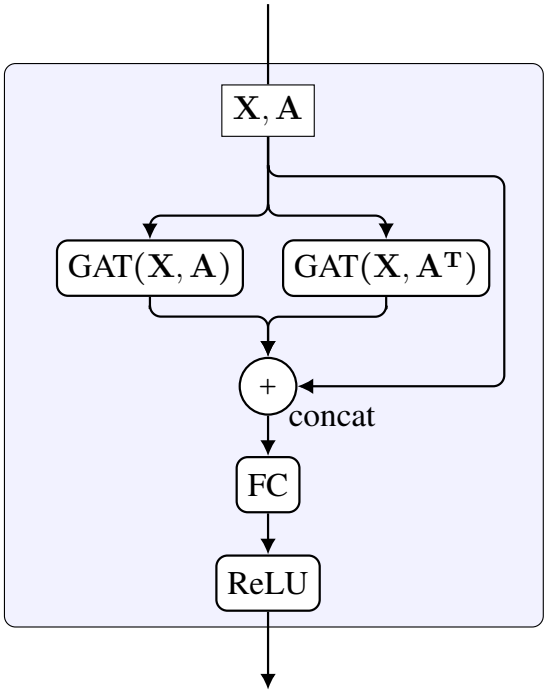



Figure 5.8: GAT Block design with bi-directional layers.

Jumping Knowledge (Xu et al., 2018) is applied to all intermediate node embeddings and combined with final node embeddings. Global soft attention (Li, Tarlow, Brockschmidt & Zemel, 2016) is applied across the final node embeddings to pool into a graph-level embedding. This graph-level embedding is combined with each individual final node embedding and passed through the fully connected output layers. Lastly, a sigmoid activation is used to provide the final output node probabilities.

Figure 5.8 shows the design of the GAT Blocks inspired by (Rawson & Reger, 2020). $\mathbf{X}$ is the node feature matrix and $\mathbf{A}$ is the adjacency matrix. A GAT layer is used for each edge direction. The outputs of the GAT layers are concatenated together alongside the initial node feature matrix $\mathbf{X}$ through a skip connection and passed through a ReLU nonlinearity. The adjacency matrix and node positions (row positions of $\mathbf{X}$) are unchanged throughout the block.

We chose the Graph Attention Network architecture for the graph neural network

layers as they provided the best balance between performance and memory usage. Experiments with other architectures such as Graph Convolutional Networks (Kipf & Welling, 2017) exhibited worse performance, while more complex architectures such as Directed Acyclic Graph Neural Networks (Thost & Chen, 2021) and heterogeneous graph neural networks (Schlichtkrull et al., 2018) restricted the size of games that were usable while providing minimal improvement in performance.

To finally infer the `next` fluents and `legal` actions, node masking is applied to the output graph to extract `next` and `legal` nodes. To discretise the output probabilities, the extracted nodes with predicted probability above a threshold value are selected. As the nodes in the output graph retain the same positions as in the input graph, we can re-contextualize the selected output nodes by recalling their lexical mappings $F_G(v)$.

### 5.4.3 Dataset Generation

To train the neural network, we require a dataset of training samples containing game states and the inferred legal actions and next fluents. Using the definition of IRGs and node-vector embeddings, we propose a general method of generating training datasets for a game described in GDL.

To generate a dataset, the GDL description of the selected game $G$ is used to play out random games. As new states are visited throughout the game plays, we store the states as instantiated rule graphs and use a Prolog based reasoner to generate ground truth logical inferences for the states as the training target. While inferring the legal actions only requires the fluents of a state, inferring the fluents present in the next state additionally requires the actions selected. To account for this, we additionally store a randomly selected legal joint action in a given state. The training targets for each example is the truth valuations of the `legal` and `next` rules for a given state or state and joint action pair respectively. These correspond directly to the `legal` and `next`

nodes present in the IRG. The task of the neural network is then to learn to approximate a function that converts each initial node-vector embedding into an output probability. As such, the dataset can be seen as a partial node classification task across multiple graphs.

**Algorithm 3** Dataset $\mathcal{D}_G$ Generation algorithm

```
1:  function GENERATEDATASET(Game description G, number of examples d)
2:      S_v ← ∅                                   ▷ Initialise empty set of collected examples
3:      P ← p_0, ..., p_k = GETROLES(G)
4:      while |S_v| < d do                        ▷ Collect training examples
5:          S ← f_0, ... f_i = GETINITIAL(G)
6:          while not ISTERMINAL(G, S) do         ▷ Play a game to generate examples
7:              for all p_i ∈ P do
8:                  M_i ← GETLEGALACTIONS(G, p_i, S)
9:                  m_i ∈_R M_i                   ▷ Randomly select a legal action
10:             a ← (p_0, m_0), ..., (p_k, m_k)
11:             S_next ← GETNEXTSTATE(S, a)
12:             if (S, a, S_next, M_0 ... M_k) ∉ S_v then S_v ∪ {(S, a, S_next, M_0 ... M_k)}
13:             S ← S_next
14:     R ← (V, E, T) = TRANSFORMTORULEGRAPH(G)
15:     D_G ← ∅                                   ▷ Initialise empty dataset
16:     for all (S, a, S_next, M_0 ... M_k) ∈ S_v do
17:         I ← (V, E, T, L) = INSTANTIATERULEGRAPH(R, S, ∅)
18:         I_a ← (V, E, T, L_a) = INSTANTIATERULEGRAPH(R, S, a)
19:         X ← GETVECTOREMBEDDING(I)
20:         X_a ← GETVECTOREMBEDDING(I_a)
21:         A ← GETADJACENCYMATRIX(R)
22:         l⃗ ← LABELLEGALTARGETS(R, M_0 ... M_k)  ▷ Generate legal training targets
23:         n⃗ ← LABELNEXTTARGETS(R, S_next)        ▷ Generate next training targets
24:         D_G ∪ {((X, A), l⃗), ((X_a, A), n⃗)}
25:     return D_G
```

Algorithm 3 shows our implementation, in which a game description $G$ is used to generate dataset $\mathcal{D}_G$ consisting of training examples of the form $((\mathbf{X}, \mathbf{A}), \vec{l})$ for legal actions and $((\mathbf{X}_a, \mathbf{A}), \vec{n})$ for next fluents. For a given state $S$ and joint action $a = does(1, m_1) \wedge \ldots \wedge does(k, m_k)$, $\mathbf{X}$ is the node feature matrix of IRG $I = (V, E, L)$ of $S$ and $\mathbf{X}_a$ is the the node feature matrix of the IRG $I_a = (V, E, L_a)$ of $S$ with `does`

| Game | IRG size | | Outdegree | | Mean path length |
|---|---|---|---|---|---|
| | Nodes | Edges | Max | Mean | |
| blocker | 5087 | 12201 | 559 | 2.40 | 4.23 |
| connectfour (C4) | 19672 | 47221 | 2249 | 2.40 | 4.33 |
| connectfour3p ($C4_{3p}$) | 17076 | 42623 | 1731 | 2.50 | 4.25 |
| hamilton | 14963 | 31516 | 1601 | 2.11 | 5.27 |
| hanoi6disks | 28055 | 70237 | 4098 | 2.50 | 4.98 |
| knightstour | 12291 | 30154 | 1393 | 2.45 | 4.45 |
| parallelbuttonsandlights | 1602 | 3245 | 182 | 2.03 | 4.93 |
| tictactoe (TTT) | 4810 | 11031 | 591 | 2.29 | 4.19 |
| doubletictactoe ($TTT_D$) | 7379 | 16338 | 803 | 2.21 | 4.61 |
| tictactoelarge ($TTT_L$) | 3553 | 8185 | 323 | 2.30 | 4.69 |
| connectfour-flat | 21450 | 51477 | 2543 | 2.40 | 4.29 |
| doubletictactoe-flat | 27280 | 64342 | 2583 | 2.36 | 4.52 |
| tictactoelarge-flat | 4203 | 9735 | 423 | 2.32 | 4.60 |

Table 5.2: Graph metrics of game datasets

nodes labelled $true$ according to actions $a$ in addition to the standard fluent labelling. $\mathbf{A}$ is the adjacency matrix of IRGs $I$ and $I_a$.

The training targets $\vec{l} = \{0,1\}^{|V|}$ and $\vec{n} = \{0,1\}^{|V|}$ are the target output probabilities of the nodes $V$ that the network must predict. As $\mathbf{X}_i$ corresponds to node $i$, if node $i$ is a `legal` node, $\vec{l}_i = 1$ if the valuation of the legal rule represented by node $i$ is $true$ and $\vec{l}_i = 0$ otherwise. Likewise, for `next` node $j$, $\vec{n}_j = 1$ if the valuation of the next rule represented by node $j$ is $true$ and $\vec{n}_j = 0$ otherwise.

The method described in Algorithm 3 is general and can be applied to any game described using GDL. For the experiments in Section 5.5, we generate 12000 training examples for each game. Although the datasets are generated on a per-game basis for simplicity, multiple datasets can be combined for multi-game mixed training or sequential training.

Table 5.2 shows some metrics of the IRGs present in the game datasets generated. The IRG sizes refer to the total number of nodes and edges in any given IRG for a state in the games listed. For the same game, IRGs for different states differ only in the

labelling function $L_S$ and as such the number of nodes and edges are the same for all states. However, between different games, the number of nodes and edges to differ as can be seen in Table 5.2. The outdegree refers to the number of outgoing edges from the nodes of a state's IRG. The maximum is the largest outdegree of all nodes in an IRG and the mean is the average outdegree of all nodes of an IRG. As mentioned before, this outdegree metric is the same for all states of a single game as only the labelling function changes. The mean path length calculates the shortest path length between all pairs of nodes in the graph and finds the average length of all these paths. Although the IRGs are directed graphs, we calculate the average path length by treating them as undirected graphs as the graph neural network architecture is able to propagate messages along both directions separately. While the size of the IRG gives a sense of the complexity of the game description, it is distinct from the state complexity of the game itself. These metrics highlight the discrepancy between game state complexity and game description complexity. For example, while $\text{TTT}_L$ has a larger number of game states compared to standard TTT due to its larger board size, the IRG for $\text{TTT}_L$ is smaller in both the number of nodes and edges. These metrics correspond roughly to the complexity of the game description, as the IRG size corresponds directly to the number and length of rules present in the description, while the average path length corresponds to the average distance of logical dependencies present in the rules. In the end of the table, $G$-flat refers to an equivalent game of $G$, which we examine later in Section 5.5.1.

We have only generated the datasets for 10 different games which are typically used in GGP tournaments, but there are many other game descriptions that exist in various repositories such as the Stanford [3] and Dresden[4] GGP repositories. Explanations of game rules and example game plays can also be found on these repositories. Descriptions for more complex games like Chess and Go are also available, but due to

[3] http://ggp.stanford.edu
[4] http://general-game-playing.de

hardware limitations we avoid using these games for our experiments. Additional game descriptions can be written for games that do not have a description already available. This presents an opportunity to provide an extensive benchmark for the inference tasks in GDL, which we encourage others to use in further graph neural network research.

### 5.4.4 Training Details

We train the neural network using the cross-entropy loss criterion for both legal action and next fluent prediction. Only `legal` and `next` node output probabilities are used in the calculations for the cross-entropy loss, with other nodes masked out. These other nodes contribute to the loss values implicitly through the message propagation mechanism in the GAT layers and the global pooling operator. Like in many other works, we use the standard Adam optimiser (Kingma & Ba, 2015) with a base learning rate of 0.0001 as well as dropout applied to the fully connected layers during training. Additionally, we use Pairnorm (Zhao & Akoglu, 2020) normalisation throughout the graph neural network layers as it has been shown to reduce oversmoothing for GNNs.

The neural reasoner is trained on datasets generated from various games with 10% of each dataset reserved as a validation split. A batch size of 32 examples are used for most games, however for larger game descriptions a smaller batch size is used due to hardware limitations. Once the network training is complete, threshold values $t_{next}$ and $t_{legal}$ for `next` and `legal` node probabilities are selected. For each task, threshold values $0.1, 0.15, 0.2, \ldots, 0.9$ are used to discretise the output probabilities, selecting the threshold value that provides the lowest mean squared error between the discretised output probabilities and the examples from the validation set.

## 5.5 Results and Discussion

To validate the effectiveness of graph-based representations such as IRGs in conjunction with a neural reasoner for GDL reasoning, we expect to see accurate prediction of `next` fluents and `legal` actions in a variety of games. The neural reasoner should have high accuracy when trained and evaluated on individual games, but also have sufficient generality to transfer its learned knowledge across different games.

We conduct a series of experiments to evaluate the performance of the neural reasoner on a collection of games and to investigate training behaviour in a variety of settings and learning modalities. Section 5.5.1 presents our results on individual games and Section 5.5.2 presents our results on transferability and generality when evaluated on multiple games.

All experiments are conducted on an Intel Xeon Silver 4210 CPU with 96 GB of memory running Ubuntu 20.04.2 LTS and 4 NVIDIA GeForce RTX2080 Ti. The implementation is written in Python using the PyTorch Geometric library (Fey & Lenssen, 2019) and PySwip (Tekol & contributors, 2020).

### 5.5.1 Training and Evaluation on Individual Games

We first present the accuracy of the neural reasoner trained and evaluated on individual games. Section 5.5.1 investigates issues caused by long-range dependencies in game rules and the effect of a *flattening* procedure on the neural reasoner's accuracy. Section 5.5.1 compares the effect of different IRG labelling functions used for the neural reasoner. Finally, Section 5.5.1 presents our ablation studies across a variety of training and architecture parameters.

As the networks are trained to predict `legal` actions and `next` fluents, we measure the accuracy of the neural reasoner over 100 randomly played games. For a given input state, the predicted legal actions and next fluents are given by the neural reasoner's

| Game | Next fluents | | | Legal actions | | |
|---|---|---|---|---|---|---|
| | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ |
| parallelbuttonsandlights | 0.9319 | 0.8989 | 0.9151 | 1.0000 | 1.0000 | 1.0000 |
| tictactoe (TTT) | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| tictactoelarge ($\text{TTT}_L$) | 1.0000 | 1.0000 | 1.0000 | 0.7814 | 0.7750 | 0.7782 |
| doubletictactoe ($\text{TTT}_D$) | 1.0000 | 1.0000 | 1.0000 | 0.9212 | 0.9982 | 0.9581 |
| connectfour (C4) | 0.9358 | 1.0000 | 0.9668 | 1.0000 | 1.0000 | 1.0000 |
| connectfour3p ($\text{C4}_{3p}$) | 0.9945 | 0.9601 | 0.9770 | 0.9175 | 1.0000 | 0.9570 |
| blocker | 0.8894 | 1.0000 | 0.9415 | 1.0000 | 1.0000 | 1.0000 |
| knightstour | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| hamilton | 1.0000 | 0.9494 | 0.9740 | 1.0000 | 1.0000 | 1.0000 |
| hanoi6disks | 0.8755 | 1.0000 | 0.9336 | 1.0000 | 0.7266 | 0.8417 |

Table 5.3: Neural reasoner accuracy over 100 games.

output node probabilities that are greater than the selected threshold. This is then compared to the ground truth which is inferred with a Prolog based reasoner.

Table 5.3 shows the accuracy of the neural reasoner on 10 different games, over 100 game plays on each. Across all games, the neural reasoner shows fairly high accuracy, achieving 100% at several games and at least 80% accuracy in most games. Notable exceptions are $\text{TTT}_L$ and hanoi6disks at legal prediction, which we investigate further in Section 5.5.1.

Figure 5.9 shows that the training behaviour across four different games. The differences exhibited here also contribute to the varying levels of accuracy that the neural reasoner achieves for different games. Dropout is not applied when calculating the validation loss, leading to the the lower values. The differences in epochs correspond to smaller batch sizes due to hardware limitations as mentioned earlier in Section 5.4.4.

### Rule Flattening

Of note in the individual games results was the relatively poor performance of the variants of TTT, namely $\text{TTT}_D$ and $\text{TTT}_L$, when compared to default TTT. We discovered that a potential cause for this disparity is in the way some rules are defined. In TTT, all of the legal and next rules are defined in a *flat* manner, with the heads (legal or

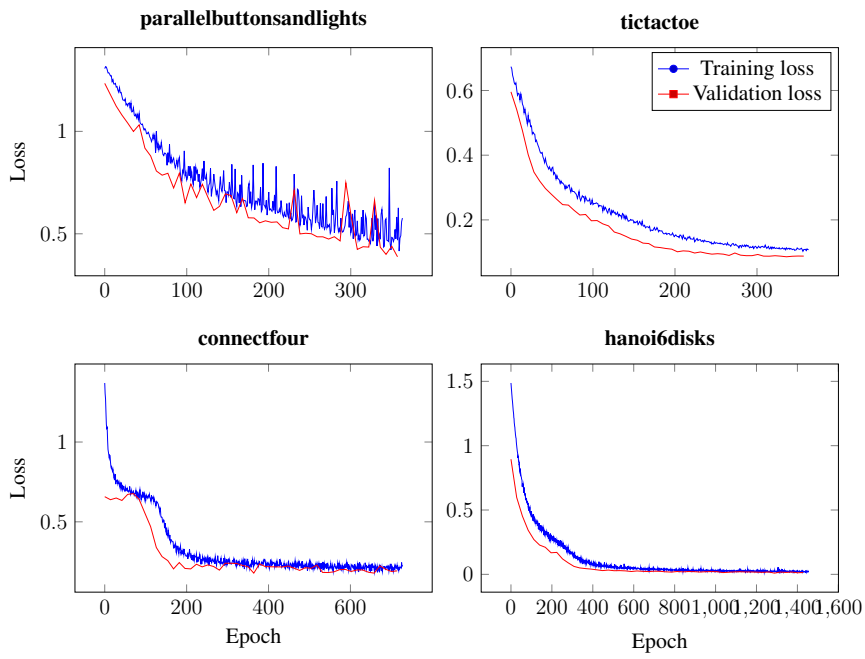



Figure 5.9: Training loss for parallelbuttonsandlights, tictactoe, connectfour and hanoi6disks.

```
(<= (legal xplayer (mark 1 1 x))
    (true (control xplayer))
    (emptycell 1 1))

(<= (emptycell 1 1)
    (index 1)
    (index 1)
    (not (true (cell 1 1 x)))
    (not (true (cell 1 1 o))))
```

```
(<= (legal xplayer (mark 1 1 x))
    (true (control xplayer))
    (index 1)
    (index 1)
    (not (true (cell 1 1 x)))
    (not (true (cell 1 1 o))))
```

Figure 5.10: Left: Standard definition of rules in $\text{TTT}_L$. Right: Flattened version of the same rules

next predicates) directly depending on fluents that represent the basic features of the states. However, in other games such as $\text{TTT}_D$ and $\text{TTT}_L$, several of the legal and next predicates have a longer dependency on fluents via other intermediate predicates.

For example, in the rule fragments shown in Figure 5.10, the left two rules are from the standard description of $\text{TTT}_L$. The legal predicate `(legal ...)` in the first rule depends on the predicate `(emptycell 1 1)`, which is not a fluent and further depends on the fluent `(cell 1 1 x)` in the second rule. In this example, the `(legal ...)` predicate is dependent on the `(cell 1 1 x)` fluent through this

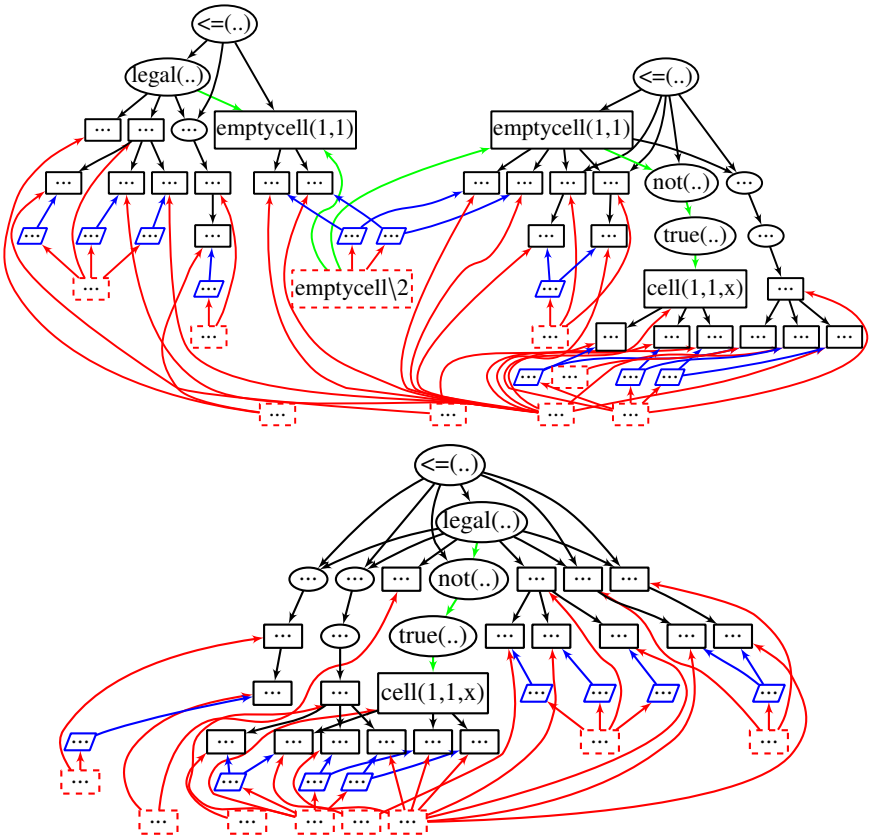



Figure 5.11: Rule graphs of unflattened (top) and flattened (bottom) fragment from Figure 5.10. Labels have been removed for clarity.

chain of reasoning. This constitutes a long-range dependency when compared to the other fluent `(control ...)`, which is a direct dependency for the `(legal ...)` predicate. In the rule on the right, the legal predicate `(legal ...)` directly depends on the fluent `(cell 1 1 x)`. We give simple flattening as a process analogous to substitution, inserting the body of the second rule into the first rule. In particular, `(emptycell 1 1)` is replaced with its corresponding body. This process shall retain the equivalence of the games.

Figure 5.11 shows the rule graphs of the unflattened and flattened rules, highlighting the effect of this process: the dependency path from the `legal(..)` node to the fluent `cell(1,1,x)` node reduces from 6 in unflattened version to 3 in the flattened version. This shortened dependency between nodes can be seen when comparing the average

| Game | Description | Next fluents | | | Legal actions | | |
|---|---|---|---|---|---|---|---|
| | | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ |
| $\text{TTT}_L$ | standard | 1.0000 | 1.0000 | 1.0000 | 0.7083 | 0.8916 | 0.7894 |
| | flat | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| $\text{TTT}_D$ | standard | 1.0000 | 1.0000 | 1.0000 | 0.8169 | 1.0000 | 0.8992 |
| | flat | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| C4 | standard | 0.9337 | 1.0000 | 0.9657 | 1.0000 | 1.0000 | 1.0000 |
| | flat | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |

Table 5.4: Next fluent and legal action prediction for 100 games of standard and flattened games

path lengths of the unflattened and flattened versions of the same game in Table 5.2 (last three rows), with all the flattened versions showing shorter average paths lengths than their unflattened counterparts.

To test whether the poor performance seen in some games is caused by the behaviour seen in unflattened rules, we train and evaluate the neural reasoner on the flattened versions and compare the performance to the standard, unflattened versions. Table 5.4 shows the improved accuracy of the neural reasoner trained on flattened rules compared to the standard unflattened rules, on the three games.

These results suggest that the distance of the dependency of legal and next predicates to the fluents is an important factor in our representation and flattening the rules could reduce the distance and potentially increase overall performance of the neural reasoner. Some limitation of this approach is that a systematic automated flattening process can be difficult, e.g., with recursively defined rules and exponential growth of the number of the rules. Note that game hanoi6disks also exhibited a large number of unflattened legal rules, which could not be easily flattened.

### Labelling Functions

We conducted experiments to compare the effects of the different labelling functions on neural reasoner accuracy by training on data with minimal labelling as well as with

| Game | Train labelling | Test labelling | Next fluents | | | Legal actions | | |
|---|---|---|---|---|---|---|---|---|
| | | | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ |
| $\text{TTT}_L$ | minimal | minimal | 0.9967 | 0.9593 | 0.9776 | 0.9117 | 1.0000 | 0.9538 |
| | | subtree | 1.0000 | 0.9056 | 0.9505 | 1.0000 | 0.5063 | 0.6722 |
| | subtree | minimal | 0.0000 | 0.0000 | 0.0000 | 0.6667 | 0.2175 | 0.3280 |
| | | subtree | 1.0000 | 0.9507 | 0.9747 | 0.9793 | 0.8980 | 0.9369 |
| $\text{C4}_{3p}$ | minimal | minimal | 1.0000 | 0.9245 | 0.9607 | 0.6394 | 0.9667 | 0.7697 |
| | | subtree | 1.0000 | 0.9259 | 0.9615 | 0.5882 | 0.9936 | 0.7389 |
| | subtree | minimal | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| | | subtree | 1.0000 | 1.0000 | 1.0000 | 0.6950 | 0.8960 | 0.7828 |

Table 5.5: Next fluent and legal action prediction with varying labelling functions

subtree-complete labelling. As the neural reasoner's input only requires a node feature matrix and an adjacency matrix, we are also able to test the effect of using minimal labelling with a neural reasoner trained on subtree-complete labelled data and vice-versa. Table 5.5 shows the results of these experiments on $TTT_L$ and $\text{C4}_{3p}$.

We find that the minimal labelling performs best in $\text{TTT}_L$ while subtree-complete labelling performs best in $\text{C4}_{3p}$. In $\text{TTT}_L$ we find that using a labelling that the neural reasoner was not trained on exhibited reduced accuracy, but minimal train labelling exhibits slightly increased next fluent prediction accuracy when using subtree-complete test labelling.

In the case of subtree-complete train labelling with minimal test labelling, for both games the neural reasoner exhibits extremely poor accuracy. We found that the neural reasoner outputs probabilities close to zero for all output nodes regardless of state which combined with the threshold values selected for the neural reasoner lead to all next fluents (and legal actions as well in the $C4_{3p}$ case) to be predicted as $false$.

In contrast, when using minimal train labelling with subtree-complete test labelling, the neural reasoner is still able to perform well although with some diminished accuracy. We observe that with minimal train labelling, the output probabilities of the neural reasoner with subtree-complete labelled inputs are slightly lower than that with minimal labelled inputs.

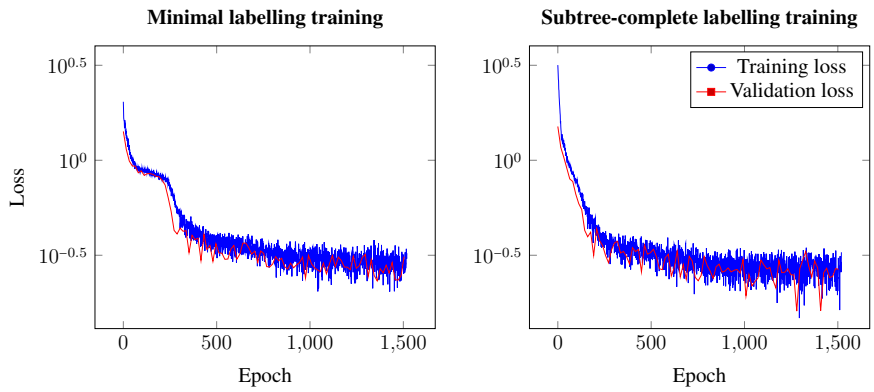



Figure 5.12: Comparison of minimal labelling and subtree-complete labelling during training for connectfour$_{3p}$

Figure 5.12 shows that although both labelling functions are able to achieve fairly similar results in terms of accuracy, they show different training characteristics. The minimal train labelling suffers a short plateau at around 200 epochs, while the subtree-complete train labelling does not and is also able to reach lower loss values.

As described in Section 5.3.2, the minimal labelling function labels a subset of the nodes labelled by the subtree-complete labelling function. When using a subtree-complete labelled input with a reasoner trained on minimal labelled data, the additionally labelled nodes may likely be seen as the network as a form of noise which affects the overall accuracy. However, when trained on subtree-complete labelled data, minimal labelled input may not contain sufficient information for the network to perform an accurate prediction, leading to the extremely poor performance. Additional work investigating the exact cause of this behaviour is needed and could potentially be a way to find the optimal labelling function, as we observed that neither labelling functions presented perform best in all games tested.

### Ablation Studies

To validate our choices for the model architecture, we complete ablation studies to investigate the effects of various components of the neural reasoner. This requires

| Model | Next fluents | | | Legal actions | | |
|---|---|---|---|---|---|---|
| | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ |
| default | 0.9660 | 0.9540 | 0.9600 | 0.9571 | 0.9734 | 0.9652 |
| 1-layer | 1.0000 | 0.9516 | 0.9752 | 0.8301 | 1.0000 | 0.9072 |
| 5-layer | 0.9398 | 0.9648 | 0.9522 | 0.9686 | 0.9195 | 0.9434 |
| GCN | 1.0000 | 0.9502 | 0.9745 | 0.8718 | 0.9293 | 0.8996 |
| SAGE | 0.9841 | 0.9552 | 0.9694 | 0.8378 | 1.0000 | 0.9117 |
| no-pool | 1.0000 | 0.9504 | 0.9746 | 0.9745 | 0.9084 | 0.9403 |
| no-jk | 0.9420 | 0.9650 | 0.9533 | 0.9527 | 0.9753 | 0.9639 |
| no-jk-pool | 0.9860 | 0.9517 | 0.9686 | 0.9186 | 1.0000 | 0.9576 |
| no-input-fc | 1.0000 | 0.9508 | 0.9748 | 0.9185 | 1.0000 | 0.9575 |
| unidirectional | 0.0000 | 0.0000 | 0.0000 | 0.3700 | 0.5145 | 0.4305 |
| undirected | 1.0000 | 0.9512 | 0.9750 | 0.9173 | 1.0000 | 0.9569 |

Table 5.6: Model accuracy over 100 games of $C4_{3p}$.

several modified versions of the neural network architecture with the aforementioned components either removed or modified. The default model refers to the architecture described in Section 5.4.2. The 1-layer and 5-layer models use the same architecture as the default model, however uses either only 1 GAT Block layer or 5 GAT Block layers. The GCN and SAGE models replace the GAT in the model with Graph Convolution Layers (Kipf & Welling, 2017) and GraphSAGE layers (Hamilton, Ying & Leskovec, 2017) respectively. The no-pool, no-jk and no-jk-pool models remove the Global Attention pooling layer, the Jumping Knowledge aggregation and both respectively. The no-input-fc removes the fully connected input layers prior to the GAT Block layers. The unidirectional model removes the reverse-edge GAT layer in the bi-directional GAT Block, while the undirected model converts the input graph into an undirected graph before processing. We train and test the various models on $C4_{3p}$ and compare their performance to the default architecture. We use $C4_{3p}$ as it has a good balance between complexity and sample efficiency for rapid experimentation.

Table 5.6 shows that the architecture that we have chosen exhibits balanced performance when compared to the other models. While several models show marginally

improved accuracy for next fluent prediction, they sacrifice accuracy in legal action prediction. In the case of the 1-layer and 5-layer models, we find that both show reduction in accuracy - 1 layer might not be sufficient for the neural reasoner to reach optimal performance, while 5 layers might cause the network to suffer from over-smoothing, a common issue with GNNs (Chen et al., 2020). Both GCN and SAGE models show a minor improvement to next fluent prediction but suffer from noticeably reduced legal action prediction. Similarly, removing the Jumping Knowledge, Global Attention and fully connected input layers can increase next fluent accuracy at the cost of reduced legal action accuracy. The poor performance of the unidirectional model highlights the necessity of utilising both directions of the edges present in the graph representation, while the undirected model presents a similar trade-off between next fluent accuracy and legal action accuracy.

### 5.5.2 Transfer Learning and Mixed Training

We first present the neural reasoner's ability to transfer its learned knowledge across different games. This is also compared to a *mixed training* scheme in which multiple games are being learned simultaneously. Section 5.5.2 compares the random mixed training scheme with a sequential mixed training scheme. In Section 5.5.2, we apply principal component analysis on the learned inferences when transferring across different games.

To evaluate the neural reasoner's transfer learning capabilities, we run evaluations across the following games: Tic-Tac-Toe (TTT), Double Tic-Tac-Toe ($\text{TTT}_D$) and Tic-Tac-Toe Large ($\text{TTT}_L$), Connect-Four (C4). We select these games as $\text{TTT}_D$ and $\text{TTT}_D$ can be seen as variants of $TTT$, while C4 is a completely different game. $\text{TTT}_D$ is composed of two boards of TTT that are played simultaneously and as such the board structure is very similar to TTT, but several rules are modified to accommodate

| Train dataset | TTT | | | $TTT_D$ | | | $TTT_L$ | | | C4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ |
| TTT | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 0.9345 | 1.0000 | 0.9662 |
| $TTT_D$ | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 0.9477 | 0.9732 | 0.9353 | 1.0000 | 0.9666 |
| $TTT_L$ | 0.4066 | 1.0000 | 0.5782 | 0.4106 | 1.0000 | 0.5822 | 1.0000 | 1.0000 | 1.0000 | 0.4092 | 1.0000 | 0.5808 |
| C4 | 1.0000 | 1.0000 | 1.0000 | 0.9468 | 1.0000 | 0.9727 | 1.0000 | 0.9249 | 0.9610 | 0.9358 | 1.0000 | 0.9668 |
| Mixed | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 0.9800 | 0.9899 |

(a)

| Train dataset | TTT | | | $TTT_D$ | | | $TTT_L$ | | | C4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ |
| TTT | 1.0000 | 1.0000 | 1.0000 | 0.0824 | 0.0159 | 0.0267 | 0.5602 | 1.0000 | 0.7182 | 1.0000 | 1.0000 | 1.0000 |
| $TTT_D$ | 1.0000 | 0.9136 | 0.9549 | 0.9212 | 0.9982 | 0.9581 | 0.7435 | 0.8322 | 0.7854 | 1.0000 | 0.8971 | 0.9458 |
| $TTT_L$ | 0.5902 | 1.0000 | 0.7423 | 0.3705 | 0.1869 | 0.2484 | 0.7814 | 0.7750 | 0.7782 | 1.0000 | 0.1285 | 0.2277 |
| C4 | 1.0000 | 1.0000 | 1.0000 | 0.0846 | 0.0163 | 0.0273 | 0.5647 | 1.0000 | 0.7218 | 1.0000 | 1.0000 | 1.0000 |
| Mixed | 1.0000 | 1.0000 | 1.0000 | 0.8262 | 1.0000 | 0.9048 | 0.5580 | 1.0000 | 0.7163 | 1.0000 | 1.0000 | 1.0000 |

(b)

Table 5.7: Neural reasoner accuracy over 100 games for (a) `next` and (b) `legal` prediction.

simultaneous play. While TTT is played on a $3 \times 3$ board, $TTT_L$ is instead played on a $5 \times 5$ board. Apart from the board size difference, the other rules remain the same, with minor changes to accommodate the larger board size. After training on each game, we evaluate the neural reasoner on the other games it was not trained on. Additionally, we conduct mixed training, where the reasoner is trained on all four games simultaneously with a uniform random mixture of game states. All networks are initialised with random weights.

Table 5.7 shows the results of transferring across different games and the potential to achieve “zero-step generalisation". Table 5.7 (a) show the results for `next` fluent prediction while (b) shows the results for `legal` move prediction. Each row of the table represents the results of a neural reasoner trained on the game specified in the first column. Each subsequent column shows its accuracy (precision $P$, recall $R$ and $F_1$ scores) on the games specified on the top row of the table. Aside from the game the neural reasoner is trained on, each other game evaluates the generality of the neural reasoner as these games are not used during training. The network is able to achieve 100% accuracy on several games that it was not trained for. For example, the network trained on TTT was able to achieve $F_1$ scores of 1.000 on the `next` prediction task for

$\text{TTT}_D$ and $\text{TTT}_L$ as well a high $F_1$ score of 0.9662 for C4.

However we can see that in several cases the accuracy of the neural reasoner is quite poor. The neural reasoner trained on $\text{TTT}_L$ shows comparatively poor accuracy when predicting both `next` fluents and `legal` moves for all games besides $\text{TTT}_L$ itself. For `legal` move prediction, the neural reasoners trained on TTT, $\text{TTT}_L$ and C4 all perform quite poorly on $\text{TTT}_D$, compared to the neural reasoner trained on $\text{TTT}_D$ itself. Furthermore, all trained neural reasoners exhibit equally poor `legal` move prediction accuracy when evaluated on $\text{TTT}_L$. This issue is likely caused by the long-range dependencies most noticeably present in $\text{TTT}_L$ as discussed in Section 5.5.1.

The neural reasoner that uses mixed training shows balanced performance across all games it is evaluated on for both `next` and `legal` prediction. Notably, it outperforms the C4 trained network on C4 itself on `next` prediction. Earlier we noted that all other trained neural reasoners performed poorly in `legal` prediction for $\text{TTT}_D$. While the neural reasoner trained on other games individually are unable to transfer `legal` prediction directly to $\text{TTT}_D$ with high accuracy, the mixed training scheme allows the neural reasoner to achieve comparable `legal` prediction accuracy on $\text{TTT}_D$ despite being trained on the other games simultaneously. This suggests that the combination of the more general graph-based representations of IRGs and neural reasoners allows a mixed training scheme to robustly learn from multiple games simultaneously.

#### Random vs Sequential Mixed Training

In the previous section, the uniform random mixed training scheme shows that the neural reasoner can achieve similar accuracy to its individually trained counterpart when trained on multiple games simultaneously. However, the method in which the examples are randomly mixed might not be practical or suitable for all game-playing scenarios. It would not be realistic to expect to have a dataset containing all possible games describable in GDL prepared and randomly mixed. It is more likely that the

| Training method | | TTT | | | $TTT_D$ | | | $TTT_L$ | | | C4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ | $P$ | $R$ | $F_1$ |
| Random | Next | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 0.9345 | 1.0000 | 0.9662 |
| | Legal | 1.0000 | 1.0000 | 1.0000 | 0.8209 | 1.0000 | 0.9016 | 0.5640 | 1.0000 | 0.7213 | 1.0000 | 1.0000 | 1.0000 |
| Sequential | Next | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 0.9252 | 0.9612 | 0.9363 | 1.0000 | 0.9671 |
| | Legal | 1.0000 | 1.0000 | 1.0000 | 0.8235 | 1.0000 | 0.9032 | 0.7707 | 0.7974 | 0.7838 | 1.0000 | 1.0000 | 1.0000 |

Table 5.8: Neural reasoner accuracy in random and sequential mixed training.

neural reasoner requires additional training when new games are encountered and new datasets are generated.

Therefore, we investigate a *sequential mixed training* scheme. Rather than randomly mixing the examples from various games into a single, large dataset, the network is trained on the different game datasets sequentially. There are two questions this raises: Firstly, does the sequential training cause the network to “forget" the earlier trained games? Secondly, can the prior training on other games assist the later training, leading to faster convergence? To answer this, we train a neural reasoner on the game sequence TTT→$TTT_D$ → $TTT_L$ → C4 and compare its accuracy to the random mixed training scheme.

Table 5.8 shows the accuracy of the sequentially trained neural reasoner shows similar performance to the random mixed trained reasoner. For the sequentially trained neural reasoner, the evaluation is conducted after it has completed training on all four game datasets. This shows that the network does not “forget" its learned inference on earlier games after completing training on the other games. Although TTT is the first game in the sequence, the sequentially trained neural reasoner achieves equal `next` and `legal` prediction accuracy compared to the randomly mixed neural reasoner. Overall, the sequentially trained neural reasoner also shows balanced performance across the games it is trained on when compared to the individually trained networks.

To further understand the differences between random and sequential mixed training, we examine the training and validation loss of both methods over the course of a full training run. For both methods, the validation loss is calculated over a mixture of

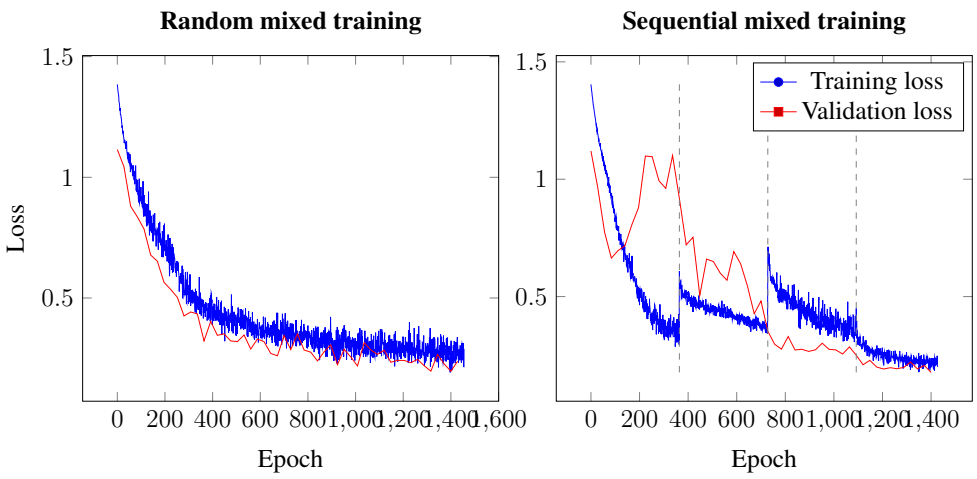



Figure 5.13: Comparison of random and sequential mixed training. Dashed lines show when the sequential training scheme transitions between datasets.

games to ensure consistency between them. Figure 5.13 shows that the sequentially trained neural reasoner exhibits much more instability during training as can be seen in the fluctuating validation loss. However, the sequentially trained network is still able to achieve final loss values equivalent to the mixed training counterpart despite the instability.

Within a single game dataset, the sequentially trained network exhibits similar stable training loss comparable to that of the random mixed trained network. When transitioning to a new game dataset, we can observe a spike in the training loss of the sequentially trained network. However, these spikes never exceed the initial loss values at the beginning of training. This suggests that the prior training can help the network to converge faster on newly encountered game datasets. The random mixing of various game states likely acts as a type of regularisation, whereas the sequential mixed training can potentially lead to overfitting on each individual game before moving on to the next. Regardless, sequential mixed training presents an alternative training scheme when random mixed training is not possible or impractical.

The capability of sequential mixed training to achieve similar results emphasises the robustness and generality of our approach. The neural reasoner has the ability to

retain learned inferences as well as learn new games as they are introduced.

### Principal Component Analysis on Transferred and Mixed Reasoners

To further investigate the behaviour of the trained networks when transferring learned inferences to other games we utilise principal component analysis (PCA). We apply PCA to the intermediate globally pooled graph embedding generated by the GNN prior to the fully connected output layers. PCA reduces the dimensionality of this intermediate graph embedding. This allows us to visualise the the learned inferences of the trained networks on various states across different games. Figure 5.14 shows the plots of these graph embeddings when the dimension is reduced to 2. Each plot is associated with a separate trained network. The first four are trained individually on TTT, TTT$_D$, TTT$_L$ and C4 respectively; the fifth is trained using a random mixed training scheme on all four games. Each point represents the output of PCA on the graph embedding of a game state after it has passed through the corresponding trained network. This visualisation allows us to observe the similarity of various game states as inferred by the trained networks.

As seen in Figure 5.14, across all trained networks, the states of the same game tend to be clustered together. With the mixed trained network, the $TTT, TTT_D$ and $TTT_L$ states are clustered closer to each other relative to the states of $C4$. This is likely due to the similarity in these games as $TTT_D$ and $TTT_L$ are variants of $TTT$. The individually trained networks do not exhibit this behaviour and instead cluster the $TTT_L$ states more distant than the other states. This again could be caused by syntactic differences from the long-range dependencies present in $TTT_L$ as mentioned in Section 5.5.1. While the individually trained networks are able to accomplish some level of transfer across different games, the mixed trained network learns a more general understanding of the inference tasks required for various GDL games. This corresponds with the overall more robust performance across all tested games in the mixed trained network as seen

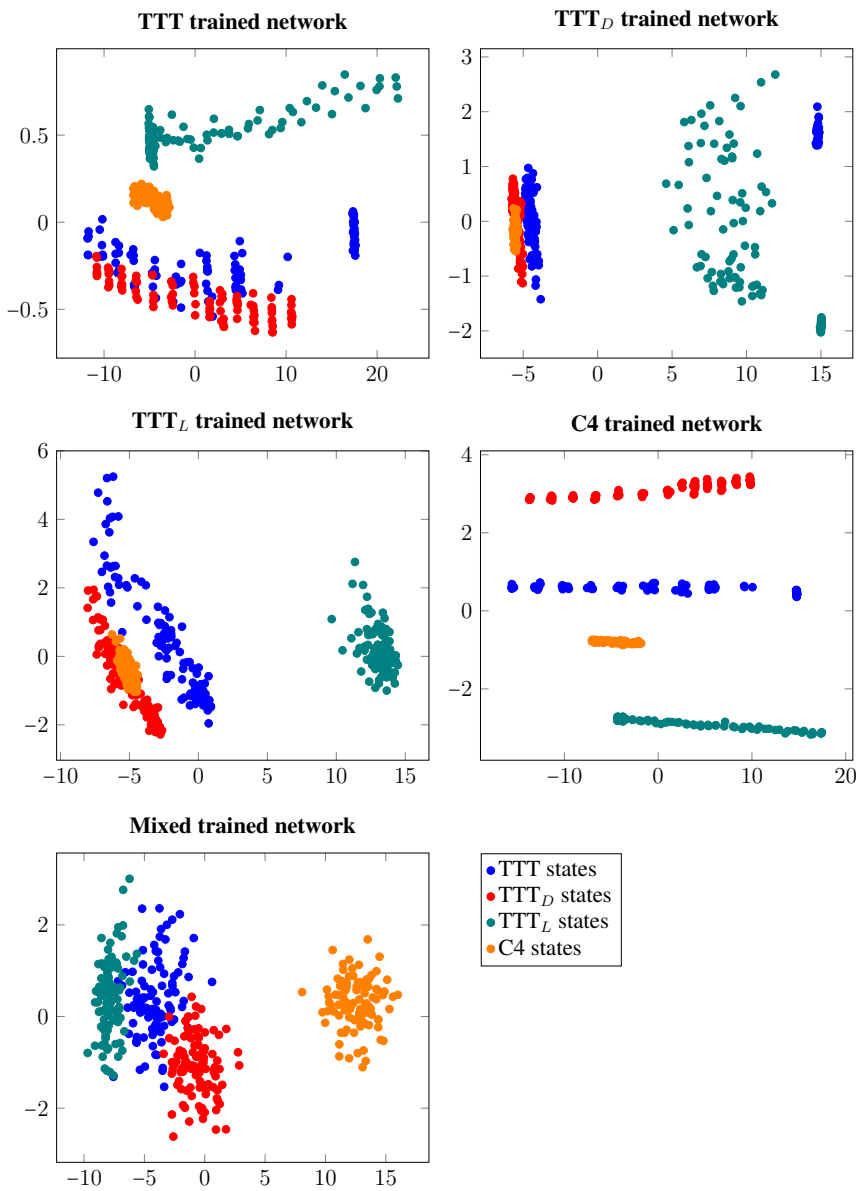



Figure 5.14: Principal component analysis of intermediate graph embedding generated by neural reasoner prior to output layers.

in Table 5.7.

### 5.5.3 Discussion

The current implementation of instantiating the rule graph as defined in Section 5.3.2 can be improved in both expressiveness and efficiency. Firstly, the expressiveness of the labelling functions for IRGs has a direct impact on the effectiveness of the neural reasoner. We have explored two such functions, minimal and subtree-complete, and found that they both show comparable performance with trade-offs in different games. There are still many other possible labelling functions yet to be explored that may have better performance on a larger selection of games. Secondly, the current instantiation method requires the game description to be grounded, which is a computationally expensive process that can lead to extremely large graphs. While the GNN-based neural reasoner can accommodate graphs of arbitrary size, larger graphs lead to slower inference times and limitations in batch size during training. It is possible that only targeted rules need to be grounded instead, as the rule graph representation is capable of representing ungrounded variables. Using variables and only partially grounded rules would allow for more compact and efficient representations. However, this may cause other issues due to variables inducing additional long-range dependencies due to their referential nature.

A key limitation we have discovered is the poor performance of the neural reasoner on games descriptions that contain long-range dependencies. We found that by flattening these rules, the neural reasoner was able to improve its accuracy. However, as mentioned before, this flattening process is resource intensive due the recursive definition of some rules and potential exponential increase in size. Furthermore, there are several games where this flattening is not possible without completely rewriting the rules. More generally, this issue is likely due to the graph neural network architecture being unable

to make inferences across long-range dependencies. This also includes other relational structures such as negation, which can be seen as a two-hop dependency. Rather than relying on a flattening process, it would be preferred to improve the graph neural network architecture to accommodate these long-range dependencies. Further research is required to find suitable GNN architectures that are deep enough to learn across long message paths without suffering from extreme over-smoothing.

Currently, we have only trained the neural reasoner to predict legal actions and next fluents. To implement a complete GDL reasoner, we also need to infer whether a state is terminal and the goal values of each player when the state is terminal. We leave this for further research as there are issues that arise from the unbalanced nature of the dataset (there are many more non-terminal states than terminal states) as well as the limitations faced due to unflattened rules (most terminal and goal rules contain long-range dependencies).

## 5.6 Conclusion

In this chapter we have presented a method to approach GDL reasoning with neural networks in a general manner. The translation of the GDL rules and game states to instantiated rule graphs allows for the application of graph neural networks that are able to reason on various games without resorting to game-specific networks.

We have implemented a neural reasoner that can learn to reason on various games individually, achieving high accuracy in most games. For the games in which the neural reasoner's accuracy suffered, we found that a flattening process was able to improve its performance. Further investigation into the effect of the labelling functions for the IRGs found that both the minimal and subtree-complete functions had comparable results for the neural reasoner's accuracy. We conducted ablation studies to evaluate the architecture of the neural reasoner. We experimented with alternate architectures

consisting of varying GNN depth, GNN type, pooling, IRG edge direction and other parameters. Compared to the other models, we found that our chosen architecture was able to balance high prediction accuracy between both the next and legal prediction tasks.

Furthermore, we are able to train the neural reasoner to transfer learned knowledge across unlearned games as well as train on a random mixture of multiple games simultaneously. With mixed training, we found that the neural reasoner exhibited additional robustness when learning to reason across all evaluated games, outperforming the individually trained networks when they had to transfer to certain games. The neural reasoner can also be trained using a sequential mixed training scheme. Rather than randomly mixing the game datasets together, the network is trained on a sequence of game datasets. The network achieves comparable results to random mixed training and does not forget the trained games from earlier in the sequence. This is likely the most practical method of training a neural reasoner for use in a game player, as this allows the neural reasoner to learn new games as they are introduced to them. Both mixed training methods highlight the improved robustness and generality that IRGs and graph neural networks provide when combined. Additionally, principal component analysis applied to the intermediate graph embeddings visualise the differences in the learned inferences of the trained networks. The neural reasoner that uses mixed training exhibits more general reasoning capability, clustering similar games closer than the individually trained counterparts.

While the instantiated rule graph representation and the graph neural network architectures have been used to learn the reasoning task in this chapter, further work could extend these methods to apply them to the task of game playing. Using a modified network architecture based on these approaches in an AlphaZero style self-learning agent would allow for the agent to learn and play multiple game simultaneously, in a truly general manner.

Despite the limitations we have encountered, instantiated rule graphs present a new paradigm of learning for GDL reasoning and beyond. Many prior approaches that attempted to detect game features could now be framed as a learning problem using our representations. In the same way that a CNN is able to effectively take advantage of the features present in the grid-like boards of Go, Chess and Shogi, a graph neural network would be able to take advantage of much more general features present in various games. Essentially, this would allow the neural network to learn beyond game specific features and rather learning higher-level features about games more broadly, gaining a conceptual understanding of how games function and how they can be exploited.

# Chapter 6

# Conclusion

We showed that it is possible to utilise deep reinforcement learning for GGP with our agent, GGPDeepRL . We have investigated the behaviour of the agent and explored the affects of various choices for such an architecture. However, we did note that it underperformed in several games and struggled with cooperative games. More work can be done to optimise the neural network architecture, the MCTS component or the training scheme to further improve performance.

A key limitation of directly implementing an AlphaZero-style agent for GGP is the type of neural network used. Convolutional neural networks cannot be used as not all games can be effectively represented in matrix form and those that are still require handcrafted networks for each game. Fully-connected neural networks (with fluent lists as input) are more general and thus are capable of capturing any game described in GDL, but are not as capable as other architectures at taking advantage of structural information within a game state. Additionally, each game requires its own neural network as the input space of each game varies differently. This also means that for each game, the agent must train the neural network from scratch.

The neural reasoner we have presented has shown promising results, with high accuracy in most games and strong transfer learning capabilities. The ability to train

on multiple games (in a mixed or sequential manner) presents a practical use-case for such a reasoner. The neural reasoner did show some inaccuracy in some games with the architecture we have proposed so far. Additional optimisation could be done to improve this performance, such as in the labelling of the instantiated rule graph or the specific graph neural network architectures used.

However, we do note several limitations. Firstly, the architecture we have presented has issues when game descriptions contain long-range dependencies. While we are able to remove some of these dependencies by flattening some rule descriptions, ideally future work could involve optimising and improving the graph neural network architecture to overcome this issue. Secondly, the instantiated rule graph representation we use requires the grounding of the game description. Like in the case of propositional networks, this limits the games we are able to reason with. However, it is possible that a more localised grounding of the game rules might be usable for a neural reasoner, as neural networks are typically more resilient to noisy or incomplete inputs compared to traditional logical approaches.

As mentioned before, we have presented solutions to the two main tasks of GGP separately: that is, we have presented a deep reinforcement learning agent and a neural reasoner. Note that the GGPDeepRL uses a standard Prolog-based reasoner for its inference and that the neural reasoner was only evaluated on the inference task itself. It is possible to replace the Prolog-based reasoner in GGPDeepRL with a neural reasoner. However, as we have noticed in the results of the neural reasoner, there are still some issues with prediction accuracy in certain games. With the experiments for GGPDeepRL , we wanted to focus on the deep reinforcement learning algorithm itself. Using an inaccurate reasoner would cause additional issues not directly related from the core algorithm and impede our investigations.

Furthermore, we noted that the neural network architecture used by GGPDeepRL imposed a severe limitation to the agent. Each game would necessitate a unique neural

network and would require each network to be trained from scratch. Using instantiated rule graphs, a graph neural network could possible be constructed and trained to predict the policy and value required for deep neural networks. This would be a more difficult than the inference task, as the dependencies would be much more implicit and longer-range in the graph.

However, if a deep reinforcement learning agent were able to learn from multiple games with a single neural network, it would allow it to transfer learned strategies across different games and learn to play multiple games simultaneously. This, at least from a learning perspective, epitomises the ideals of general game playing: the capability of an agent to *learn* to play any game, given only the description of the game.

General game playing has from its initial foundations had the goal of encouraging the development of more general artificial intelligence agents. With the use of GDL as a powerful language capable of encoding a large number of games, GGP itself acts as a type of super-domain consisting of multiple games, each acting as their own domain with certain features unique to each game. With the development of more general learning techniques such as the ones we have proposed, we can now see that GGP agents are capable of not only learning to play specific games, but rather the larger super-domain of all GDL-describable games. Such an agent would have to learn general features that exist across multiple games. Essentially, these agents are learning to solve this super-domain and instead have to gain knowledge about how games function in a more expansive manner. Ideally, with the further development that we have recommended, one would be able to develop a GGP agent that not only can learn to play any game, but learn to play *all* games at a super-human level.

# References


Abdelaziz, I., Crouse, M., Makni, B., Austil, V., Cornelio, C., Ikbal, S., ... others (2021). Learning to guide a saturation-based theorem prover. *arXiv preprint arXiv:2106.03906*.

Abdullah, W. A. T. W. (1992). Logic programming on a neural network. *International Journal of Intelligent Systems*, *7*(6), 513–519.

Abramson, B. D. (1987). *The expected-outcome model of two-player games* (PhD thesis). Columbia University.

Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Bellman, R. (1956). A problem in the sequential design of experiments. *Sankhyā: The Indian Journal of Statistics (1933-1960)*, *16*(3/4), 221–229.

Bellman, R. (1957). A Markovian decision process. *Journal of Mathematics and Mechanics*, 679–684.

Bellman, R. (1966). Dynamic programming. *Science*, *153*(3731), 34–37.

Benacloch-Ayuso, J. L. (2012). *RL-GGP*. http://users.dsic.upv.es/~flip/RLGGP.

Bertsekas, D. (2012). *Dynamic programming and optimal control: Volume I* (Vol. 1). Athena Scientific.

Bertsekas, D. & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming*. Athena Scientific.

Björnsson, Y. & Finnsson, H. (2009). Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, *1*(1), 4–15.

Björnsson, Y. & Schiffel, S. (2013). Comparison of GDL reasoners. In *Proceedings of the IJCAI-13 workshop on general game playing (GIGA'13)* (pp. 55–62).

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... others (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, *33*, 1877–1901.

Campbell, M., Hoane Jr, A. J. & Hsu, F.-h. (2002). Deep blue. *Artificial Intelligence*, *134*(1-2), 57–83.

Chaslot, G., Bakkes, S., Szita, I. & Spronck, P. (2008). Monte-Carlo tree search: A new framework for game AI. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Vol. 4, pp. 216–217).

Chaslot, G. M.-B., Winands, M. H. & van den Herik, H. J. (2008). Parallel Monte-Carlo tree search. In *International Conference on Computers and Games* (pp. 60–71).

Chen, D., Lin, Y., Li, W., Li, P., Zhou, J. & Sun, X. (2020). Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 34, pp. 3438–3445).

Clune, J. (2007). Heuristic evaluation functions for general game playing. In *AAAI* (Vol. 7, pp. 1134–1139).

Costa, V. S., Rocha, R. & Damas, L. (2012). The YAP Prolog system. *Theory and Practice of Logic Programming*, *12*(1-2), 5–34.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *International Conference on Computers and Games* (pp. 72–83).

Coulom, R. (2007). Computing "Elo ratings" of move patterns in the game of go. *ICGA Journal*, *30*(4), 198–208.

Cropper, A., Evans, R. & Law, M. (2020). Inductive general game playing. *Machine Learning*, *109*(7), 1393–1434.

Crouse, M., Abdelaziz, I., Cornelio, C., Thost, V., Wu, L., Forbus, K. & Fokoue, A. (2019). Improving graph neural network representations of logical formulae with subgraph pooling. *arXiv preprint arXiv:1911.06904*.

Degris, T., White, M. & Sutton, R. S. (2012). Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*.

Draper, S. & Rose, A. (2016). *SanchoGGP*. https://github.com/SanchoGGP/.

Ebner, M., Levine, J., Lucas, S. M., Schaul, T., Thompson, T. & Togelius, J. (2013). Towards a video game description language.

Emslie, R. (2017). *Galvanise zero*. https://github.com/richemslie/galvanise_zero.

Engesser, T., Mattmüller, R., Nebel, B. & Thielscher, M. (2021). Game description language and dynamic epistemic logic compared. *Artificial Intelligence*, *292*, 103433. Retrieved from https://www.sciencedirect.com/science/article/pii/S0004370219300797 doi: https://doi.org/10.1016/j.artint.2020.103433

Evans, R., Saxton, D., Amos, D., Kohli, P. & Grefenstette, E. (2018). Can neural networks understand logical entailment? *arXiv preprint arXiv:1802.08535*.

Fey, M. & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. *CoRR*, *abs/1903.02428*. Retrieved from http://arxiv.org/abs/1903.02428

Garcez, A. S., Artur S d'Avila & Zaverucha, G. (1999). The connectionist inductive learning and logic programming system. *Applied Intelligence*, *11*(1), 59–77.

Garcez, A. S. d., Broda, K., Gabbay, D. M. et al. (2002). *Neural-symbolic learning systems: foundations and applications*. Springer Science & Business Media.

Gelly, S., Wang, Y., Munos, R. & Teytaud, O. (2006). *Modification of UCT with Patterns in Monte-Carlo Go* (Research Report No. RR-6062). INRIA. Retrieved from https://inria.hal.science/inria-00117266

Genesereth, M. & Björnsson, Y. (2013). The international general game playing competition. *AI Magazine*, *34*(2), 107–107.

Genesereth, M., Love, N. & Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI Magazine*, *26*(2), 62–62.

Genesereth, M. & Thielscher, M. (2014). General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *8*(2), 1–229.

Glorot, X., Anand, A., Aygun, E., Mourad, S., Kohli, P. & Precup, D. (2019). Learning representations of logical formulae using graph neural networks. In *Neural Information Processing Systems, Workshop on Graph Representation Learning*.

Goldwaser, A. & Thielscher, M. (2020). Deep reinforcement learning for general game playing. In *AAAI* (pp. 1701–1708).

Gunawan, A., Ruan, J., Thielscher, M. & Narayanan, A. (2020). Exploring a learning architecture for general game playing. In *AI 2020: Advances in Artificial Intelligence - 33rd Australasian Joint Conference, AI 2020, Proceedings* (pp. 294–306).

Hamilton, W., Ying, Z. & Leskovec, J. (2017). Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, *30*.

Haufe, S., Schiffel, S. & Thielscher, M. (2012). Automated verification of state sequence invariants in general game playing. *Artificial Intelligence*, *187*, 1–30.

Huang, X., Ruan, J. & Thielscher, M. (2013). Model checking for reasoning about incomplete information games. In S. Cranefield & A. Nayak (Eds.), *AI 2013: Advances in Artificial Intelligence* (pp. 246–258). Cham: Springer International Publishing.

Jiang, G., Zhang, D., Perrussel, L. & Zhang, H. (2016). Epistemic GDL: A logic for representing and reasoning about imperfect information games. In *25th International Joint Conference on Artificial Intelligence (IJCAI 2016)* (pp. 1138–1144).

Kingma, D. P. & Ba, J. (2015). Adam: A method for stochastic optimization. In Y. Bengio & Y. LeCun (Eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Retrieved from http://arxiv.org/abs/1412.6980

Kipf, T. N. & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*.

Knuth, D. E. & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, *6*(4), 293–326.

Kocsis, L. & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *European Conference on Machine Learning* (pp. 282–293).

Koriche, F., Lagrue, S., Piette, É. & Tabary, S. (2017). WoodStock: un programme-joueur générique dirigé par les contraintes stochastiques. *Revue d'intelligence artificielle–no*, *307*, 336.

Kowalski, J., Mika, M., Sutowicz, J. & Szykuła, M. (2019). Regular boardgames. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 33, pp. 1699–1706).

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90.

Kuhlmann, G. & Stone, P. (2007). Graph-based domain mapping for transfer learning in general games. In *European Conference on Machine Learning* (pp. 188–200).

Kuhn, H. W. (1950). Extensive games. *Proceedings of the National Academy of Sciences*, *36*(10), 570–576.

Lanctot, M., Lisỳ, V. & Winands, M. H. (2013). Monte Carlo tree search in simultaneous move games with applications to Goofspiel. In *Workshop on Computer Games* (pp. 28–43).

LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W. & Jackel, L. (1989). Handwritten digit recognition with a back-propagation network. *Advances in Neural Information Processing Systems*, *2*.

Lee, C.-S., Wang, M.-H., Chaslot, G., Hoock, J.-B., Rimmel, A., Teytaud, O., ... Hong, T.-P. (2009). The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, *1*(1), 73–89.

Levine, J., Congdon, C. B., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., ... Thompson, T. (2013). General video game playing. *Artificial and Computational Intelligence in Games*, 77-84.

Li, Y., Tarlow, D., Brockschmidt, M. & Zemel, R. S. (2016). Gated graph sequence neural networks. In Y. Bengio & Y. LeCun (Eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings.* Retrieved from http://arxiv.org/abs/1511.05493

Lin, Y., Wang, A. S., Undersander, E. & Rai, A. (2022). Efficient and interpretable robot manipulation with graph neural networks. *IEEE Robotics and Automation Letters*.

Love, N., Hinrichs, T., Haley, D., Schkufza, E. & Genesereth, M. (2008). *General game playing: Game description language specification*. Stanford Logic Group, Computer Science Department, Stanford University.

McCulloch, W. S. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

McEwan, C. & Thielscher, M. (2022). Knowledge transfer for deep reinforcement agents in general game playing. In *Australasian Joint Conference on Artificial Intelligence* (pp. 53–66).

Méhat, J. & Cazenave, T. (2010). Combining UCT and nested Monte Carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(4), 271–277.

Michie, D. & Chambers, R. A. (1968). BOXES: An experiment in adaptive control. *Machine Intelligence*, *2*(2), 137–152.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... others (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Neumann, J. v. (1928). Zur theorie der gesellschaftsspiele. *Mathematische annalen*, *100*(1), 295–320.

Olšák, M., Kaliszyk, C. & Urban, J. (2019). Property invariant embedding for automated reasoning. *arXiv preprint arXiv:1911.12073*.

Paliwal, A., Loos, S., Rabe, M., Bansal, K. & Szegedy, C. (2020). Graph representations for higher-order logic and theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 34(03), pp. 2967–2974).

Piette, E., Soemers, D. J., Stephenson, M., Sironi, C. F., Winands, M. H. & Browne, C. (2019). Ludii–the ludemic general game system. *arXiv preprint arXiv:1905.05013*.

Raina, R., Madhavan, A. & Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 873–880).

Rawson, M. & Reger, G. (2020). Directed graph networks for logical reasoning. In *PAAR+ SC$^2$@ IJCAR* (pp. 109–119).

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386.

Rosin, C. D. (2011). Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, *61*(3), 203–230.

Ruan, J. & Thielscher, M. (2011). The epistemic logic behind the game description language. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI, San Francisco, California, USA* (p. 840-845).

Ruan, J. & Thielscher, M. (2012). Strategic and epistemic reasoning for the game description language GDL-II. In L. D. Raedt et al. (Eds.), *ECAI* (Vol. 242, p. 696-701). IOS Press.

Ruan, J. & Thielscher, M. (2014). Logical-epistemic foundations of general game descriptions. *Studia Logica: An International Journal for Symbolic Logic*, *102*(2), 321–338. Retrieved 2023-07-07, from http://www.jstor.org/stable/43651944

Saffidine, A., Finnsson, H. & Buro, M. (2012). Alpha-beta pruning for games with simultaneous moves. In *Twenty-sixth AAAI Conference on Artificial Intelligence*.

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. ii—recent progress. *IBM Journal of research and development*, *11*(6), 601–617.

Schaeffer, J., Culberson, J., Treloar, N., Knight, B., Lu, P. & Szafron, D. (1992). A world championship caliber checkers program. *Artificial Intelligence*, *53*(2-3), 273–289.

Schaul, T. (2013). A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (pp. 1–8).

Schiffel, S. (2010). Symmetry detection in general game playing. In *Twenty-fourth AAAI Conference on Artificial Intelligence*.

Schiffel, S. & Björnsson, Y. (2014). Efficiency of GDL reasoners. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(4), 343–354.

Schiffel, S. & Thielscher, M. (2007). Fluxplayer: A successful general game player. In *AAAI* (Vol. 7, pp. 1191–1196).

Schiffel, S. & Thielscher, M. (2014). Representing and reasoning about the rules of general games with imperfect information. *Journal of Artificial Intelligence*

*Research*, *49*, 171–206.

Schkufza, E., Love, N. & Genesereth, M. (2008). Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In *Australasian Joint Conference on Artificial Intelligence* (pp. 56–66).

Schlichtkrull, M., Kipf, T. N., Bloem, P., Berg, R. v. d., Titov, I. & Welling, M. (2018). Modeling relational data with graph convolutional networks. In *European Semantic Web Conference* (pp. 593–607).

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85–117.

Schreiber, S. & Landau, A. (2013). *The general game playing base package.* Retrieved from https://github.com/ggp-org/ggp-base

Shannon, C. E. (1950). XXII. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *41*(314), 256–275.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., ... others (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... others (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... others (2017). Mastering the game of Go without human knowledge. *Nature*, *550*(7676), 354.

Silver, T., Chitnis, R., Curtis, A., Tenenbaum, J., Lozano-Perez, T. & Kaelbling, L. P. (2020). Planning with learned object importance in large problem instances using graph neural networks. *arXiv preprint arXiv:2009.05613*.

Sugiyama, M. (2015). *Statistical reinforcement learning: modern machine learning approaches*. CRC Press.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*(1), 9–44.

Sutton, R. S. & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, R. S., McAllester, D., Singh, S. & Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, *12*.

Tekol, Y. & contributors. (2020). *PySwip v0.2.10.* Retrieved from https://github.com/yuce/pyswip

Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Communications of the ACM*, *38*(3), 58–68.

The Stockfish developers (see AUTHORS file). (n.d.). *Stockfish.* Retrieved from https://github.com/official-stockfish/Stockfish

Thielscher, M. (2010). A general game description language for incomplete information games. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 24, pp. 994–999).

Thielscher, M. (2017). GDL-III: A description language for epistemic general game playing. In *The IJCAI-16 workshop on general game playing* (p. 31).

Thost, V. & Chen, J. (2021). Directed acyclic graph neural networks. In *International Conference on Learning Representations.* Retrieved from https://openreview.net/forum?id=JbuYF437WB6

Turing, A. M. (1953). Digital computers applied to games. *Faster than thought*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P. & Bengio, Y. (2018). Graph Attention Networks. *International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=rJXMpikCZ

Vittaut, J.-N. & Méhat, J. (2014a). Efficient grounding of game descriptions with tabling. In *Workshop on Computer Games* (pp. 105–118).

Vittaut, J.-N. & Méhat, J. (2014b). Fast instantiation of GGP game descriptions using prolog with tabling. In *ECAI* (Vol. 14, pp. 1121–1122).

Wang, H., Emmerich, M. & Plaat, A. (2018). Monte Carlo Q-learning for general game playing. *CoRR*, *abs/1802.05944*. Retrieved from http://arxiv.org/abs/1802.05944

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (PhD thesis). King's College, Cambridge, United Kingdom.

Werbos, P. (1974). *Beyond regression: new tools for prediction and analysis in the behavioral sciences* (PhD thesis).

Wielemaker, J., Schrijvers, T., Triska, M. & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, *12*(1-2), 67–96.

Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K.-i. & Jegelka, S. (2018). Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning* (pp. 5453–5462).

Zhang, D. & Thielscher, M. (2015). Representing and reasoning about game strategies. *Journal of Philosophical Logic*, *44*, 203–236.

Zhao, L. & Akoglu, L. (2020). Pairnorm: Tackling oversmoothing in GNNs. In *International Conference on Learning Representations.* Retrieved from https://openreview.net/forum?id=rkecl1rtwB