AN IMPROVED GNN-REASONER FOR GAME DESCRIPTION LANGUAGE

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING

Supervisors

Dr. Ji Ruan

March 2024

By

Weizuo Chen

School of Engineering, Computer and Mathematical Sciences

Copyright

Declaration

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

Acknowledgements

I would like to express my profound gratitude to Dr. Ruan for his invaluable guidance and Dr.Gunawan for his help. With their assistance, my research progress has significantly accelerated.

Abstract

The intersection of Artificial Intelligence (AI) with computer game playing has yielded significant breakthroughs, exemplified by Deep Blue's victory over a world chess champion and AlphaGo's triumph in Go through self-learning. The focus on General Game Playing (GGP) [17] seeks to develop AI systems capable of playing multiple games based solely on their rules, aiming for a broader intelligence application. A recent development in GGP introduced AlphaGo-style neural network learning approach in game playing [7, 10]. In these works, reasoning of game rules in the Game Description Language (GDL) is handeled by a logical reasoner based on Prolog or Propnet. The introduction of the GNN-Reasoner [9] shows a promising approach of using neural networks by employing a graph-based approach for game rules and state representation, though it faced challenges such as dataset feature design and inference task completion. Addressing these limitations, this thesis introduces an enhanced model, GNN-Reasoner-V2. Our contributions include a more efficient graph structure, a method for flattening negation rules, an improved neural network architecture that reduces the demand of hardware resource, a faster and modular dataset generation process, and a suite of analytical tools for comprehensive model evaluation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the ever-evolving landscape of artificial intelligence (AI), the intersection with computer game playing has been a captivating domain, marked by significant milestones and breakthroughs. Deep Blue's historic victory over a world chess champion in the last century stands as a testament to AI's prowess in gaming, while AlphaGo's [28] victory in the game of Go in 2016, achieved through self-learning, further underscored AI's ascendancy in the computer game playing.

A crucial aspect of AI gaming research recently is General Game Playing (GGP) [17]. It aims to create intelligent systems capable of playing various games with only the rules given, with the goal of achieving artificial intelligence with more general capabilities. However, most GGP systems developed so far have relied on algorithms developed by humans, rather than learning autonomously.

Although studies by Goldwaser and Togelius [7] and Alvaro et al. [10] showcase the formidable potential of reinforcement learning and Monte Carlo Tree Search (MCTS)[3] in the GGP domain, they share a common limitation—a lack of a universal game state representation. For instance, the convolutional neural networks utilized by these systems require distinct matrices tailored for each game.

To address this challenge, Alvaro et al. [9] introduced a novel graph-based approach

for representing game rules and states and migrated it into a Graph Neural Network known as the GNN-Reasoner. The GNN-Reasoner not only provides a more general representation of game rules and states but also showcases the potential of neural networks in logical reasoning. However, this approach faces limitations stemming from incomplete dataset feature design and imperfect inference tasks. Presently, the GNN-Reasoner is limited to predicting legal and next fluents in games, with further extensions to terminal and goal prediction tasks remaining elusive. Furthermore, the feature design of the dataset presents challenges in handling complex tasks, such as the negation problem. Additionally, both the graph structure and model architecture used by GNN-Reasoner are not capable of predicting all four tasks (legal, next, terminal, goal) simultaneously due to limitations in computational resources.

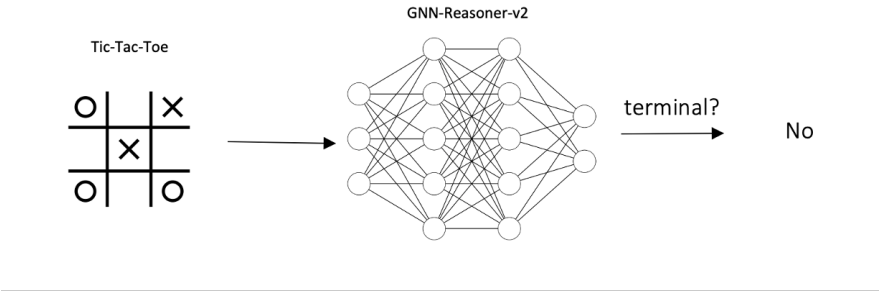

Figure 1: GNN-Reasoner-v2 (capable of inferring tasks)

To overcome the limitations of the GNN-Reasoner, we present an enhanced version called GNN-Reasoner-v2. This upgraded Graph Neural Network based reasoner excels in accurately predicting key reasoning tasks within General Game Playing (GGP), including Terminal, goal, legal, and next states. Our key contributions include: (i) A graph structure that consumes fewer hardware resources while offering more features, (ii) a graph-based method for flattening negation rules, (iii) a new neural network architecture that achieves comparable performance with reduced hardware resource utilization compared to the GNN-Reasoner, (iv) a more comprehensive and faster approach for dataset generation, (v) A suite of redesigned analytical tools to facilitate

thorough analysis and evaluation of the model's performance, (vi) A more manageable open-source code framework.

In the following chapters, we will first discuss the prior work relevant to our research, followed by a detailed description of the method of our work, including dataset generation, model architecture, analysis tools, etc. Finally, we will present a series of test data for our research, along with explanations and comparisons.

Chapter 2

Literature Review

2.1 Introduction

Before delving into the intricacies of our research, it's important to lay the groundwork by reviewing previous studies in the field. In the following sections, we'll start by exploring the history of computer games and their relationship with artificial intelligence. This historical perspective will shed light on how AI has been integrated into gaming over time, from its early stages to more sophisticated approaches.

Following that, we'll delve into General Game Playing (GGP) [17], a concept that allows AI to play a wide variety of games without needing to be specifically programmed for each one. We'll discuss its history and key components, offering insights into how AI can adapt to different gaming environments.

Lastly, we'll examine prior research on neural networks. We'll focus on a key previous research called the GNN-Reasoner [9] and other neural network frameworks used in our study.

2.2 History of Computer Game and AI

In 1944, the publish of "Theory of Games and Economic Behavior" [40] marks the inception of modern game theory. This seminal work elaborated on the foundational concepts of game theory, including zero-sum games, strategic form games, and the concept of equilibrium in games. Their contributions to game theory extended beyond mathematical modeling, impacting fields such as economics, social science, and even biology. Significantly, their work played a crucial role in the development of computer games and artificial intelligence, aiding in decision-making processes, search optimization, and opponent modeling, thereby advancing AI technology.

The intersection of computer games and artificial intelligence (AI) traces back to the 1950s when scientists began exploring the use of computer programs to simulate and execute gaming activities. Initial efforts focused on chess, as its clear rules made it an ideal testbed for the decision-making and learning capabilities of early AI under limited computational resources. In 1949, Claude Shannon [26] proposed the concept of a computer playing chess, applying one of the core algorithms of game theory, Minimax. This algorithm aimed to find the optimal decision by maximizing the player's potential benefits, laying the groundwork for future AI in gaming.

The term "artificial intelligence" was first introduced at the Dartmouth Conference in 1956, attended by university mathematicians and researchers, including Claude Shannon, and two engineers from IBM. Arthur Samuel from IBM developed a checkers program [24], marking the first game AI to run on a computer. This program was capable of learning by playing against its previous plays, a mechanism that would become a key component of the 21st-century AlphaGo [28]. In 1958, Newell, Shaw, and Simon collaborated on a chess program that improved upon the Minimax algorithm with Alpha-beta pruning, known as the NSS Chess [21] . This modification allowed the algorithm to avoid simulating unnecessary moves, saving computational resources

for better moves. This adaptation made Alpha-beta pruning a standard for future chess programs.

Over time, artificial intelligence (AI) has made significant progress in the field of computer gaming, particularly in chess. In the 1970s, computer scientists began developing chess programs capable of competing against human players. A landmark achievement of this era was IBM's Deep Blue [20] defeating world chess champion Garry Kasparov in 1997, marking a significant victory for AI in complex games. However, the majority of these significant achievements relied heavily on the computational resources. Systems including Deep Blue primarily employed force AI techniques, such as Alpha-beta tree searches based on evaluation functions of various features, databases of grandmaster opening moves, and databases of all possible chess positions and moves. The success of these AI programs was more dependent on the evaluation functions designed by researchers than on the learning capabilities of the programs themselves.

In 1986, Geoffrey Hinton et al. proposed "Learning representations by back-propagating errors" [23] sparked a great interest among researchers in experimenting with neural networks for solving various problems. This led to the creation of Neurogammon [34], designed to eliminate the need for researchers to encode human intuition (evaluation functions) when programming solutions to problems. Neurogammon beat all other programs in the first Computer Olympiad in 1989. Although it was not yet as powerful as the best human players, the principles behind it were seen by many researchers as having extraordinary potential.

By 1995, the introduction of TD-Gammon [33] marked a world wide milestone, incorporating features from Neurogammon and based on the reinforcement learning theory by Richard Sutton [31]. TD-Gammon achieved remarkable success in backgammon, laying the groundwork for numerous 21st-century applications based on reinforcement learning, like AlphaGo. Reinforcement learning focuses on finding the optimal choice under different scenarios, and supervised neural network approximated

functions for computing evaluation functions through specific types of input and output. Despite TD-Gammon's initial popularity, interest in it gradually faded due to the limited computing resources available at the time.

During the 80s and 90s, AI achieved remarkable feats in chess and checkers through force AI, but it is still a challenge in Go. The complexity of Go still be challenged even into the 21st century. In 1993, Bernd Brügmann introduced a method called "Monte Carlo Go" [18]. The Monte Carlo algorithm, a technique based on random sampling, is especially useful for solving complex problems that are difficult to solve precisely with traditional mathematical methods. By simulating a large number of random games, it estimates the best moves. This approach meant the program didn't have to search deeply until the end of the game, significantly reducing the computational load. Brügmann's method opened up new possibilities for AI to eventually beat human masters in Go.

In 2006, Rémi Coulom enhanced the Monte Carlo algorithm by developing the Monte Carlo Tree Search (MCTS) [3]. MCTS assesses various game states through extensive random simulations and dynamically builds a search tree based on these outcomes, effectively guiding decision-making processes. By 2012, Sylvain Gelly and his research team merged and refined this approach with prior studies. The program they proposed, MoGo [5], quickly becomed as the most powerful artificial intelligence for computer Go at the time. Nonetheless, to further advance AI in the field of Go, another breakthrough was necessary—deep learning.

The significant increase in global computational resources after 2010 has made the application of deep neural networks to complex games a reality. In 2013, DeepMind published a groundbreaking paper titled "Playing Atari Games with Deep Reinforcement Learning" [19], which introduced a novel approach to combining deep neural networks with reinforcement learning to train AI to play Atari games. This work demonstrated the potential of deep learning in the field of video gaming, opening new direction for development within the AI field.

Following this, in 2016, DeepMind published AlphaGo [28], an AI based on deep neural networks, reinforcement learning algorithms, and Monte Carlo Tree Search (MCTS). AlphaGo's victory over world champion Lee Sedol in the game of Go was not just a technological breakthrough but also sparked global interest and discussions, highlighting the immense potential of AI in solving complex strategic games.

By 2017, DeepMind introduced another innovative technology, Alpha Zero [29], an advanced AI system capable of achieving expert-level play in Go as well as in other board games. Unlike its predecessors, Alpha Zero learned by playing games against itself, without relying on human game data.

In 2019, DeepMind's AlphaStar [38] project developed an artificial intelligence that defeated several professional players in the game StarCraft II, showcasing the powerful performance of AI in real-time strategy games.

This advancement further demonstrated the powerful capabilities and potential applications of deep learning and self-improvement in AI research. These developments marked significant milestones in the fields of deep learning and reinforcement learning, showcasing the efficient learning and decision-making capabilities of computer programs in games and other complex tasks. DeepMind's achievements not only pushed the boundaries of artificial intelligence research but also inspired widespread exploration of the future possibilities of AI applications.

2.3 General Game Playing

General Game Playing (GGP) is a significant research domain in the field of artificial intelligence aimed at developing intelligent systems capable of autonomously playing various types of games without the need for specialized design for each game. The goal of GGP is to enable computers to comprehend game rules and formulate effective strategies based on these rules to perform well in games. Research in GGP spans

multiple aspects including game theory, logical reasoning, search algorithms, planning, and game theory. Its core concept involves the use of a General Game Description Language (GDL) to describe the rules and behaviors of games. GDL is a logical language that describes game states transitions, legal actions, termination conditions, and goal conditions through logical rules. GGP systems parse and understand game rules described in GDL, generate game trees, and formulate optimal action strategies for the next steps based on search algorithms [17].

The formal introduction of GGP can be traced back to the publication of "General Game Playing: Overview of the AAAI Competition" by Michael Genesereth, Barbulescu, and Thielscher in 2005 [6], which not only introduced the concept of GGP but also initiated competitions based on it. Subsequently, in 2008, "General Game Playing: Game Description Language Specification" [17] was published, further solidifying the concept by introducing a standardized language for describing games. This specification, known as the Game Description Language (GDL), became the foundation for numerous GGP agents and facilitated the multiple GGP competitions in subsequent years . This has attracted researchers and students from all over the world to participate. The competition evaluates the performance and effectiveness of different GGP systems through competitive gameplay across various games, driving continuous development and advancement in the field of General Game Playing.

GGP comprises four primary components: Description Language (GDL), GDL Reasoner, Game Manager, and Game Player. GDL serves as a logical language for depicting the state of a game world through true facts and employs logical rules to define transitions between states, legal moves, termination conditions, and goal conditions. The GDL Reasoner takes GDL as input and generates pertinent information about the game, such as legal moves, subsequent states, terminal conditions, and utility values. Users develop Game Players and execute their algorithms on GGP games, utilizing facts provided by the reasoner and adhering to communication protocols. Central to the

process, the Game Manager oversees all game-related operations, including updating the game state based on player actions and recording goal values upon reaching terminal states [17].

As this study only focuses on the Game Description Language (GDL) and GDL Reasoner within the GGP, the following subsections will provide detailed explanations of these two components.

2.3.1 Game Description Language (GDL)

GDL providing a set of formalized syntax and semantics, allows for the precise description of game rules, objectives, player actions, and conditions for game termination. This description enables AI programs to autonomously parse game rules and generate and optimize strategies without human intervention. Furthermore, GDL supports multiplayer, turn-based games, as well as games with random elements and asymmetric information [17]

GDL is a logic-based language designed to express the complete ruleset of turn-based games with perfect information. The language uses a set of keywords (predefined predicates) and logical constructs to describe game states, possible moves, and outcomes. The foundational keywords include `role`, `init`, `true`, `does`, `next`, `goal`, `legal`, and `terminal`, each serving to outline the structure of game-playing from initialization to termination [17].

At the core of GDL's expressiveness lie its logical constructs and predicates. Facts describe the current state of the game, while rules define how the game state changes in response to player actions. Custom predicates enable the representation of complex, game-specific conditions, such as winning configurations. The use of variables, negation, and conjunctions allows for a concise yet powerful way to describe game dynamics [17].

The development of GDL has significantly impacted AI research in game playing, enabling the creation of GGP competitions and fostering advancements in AI strategy, decision-making, and learning. GDL has not only facilitated the benchmarking of AI performance across a broad spectrum of games but also contributed to the understanding of AI's generalizability and adaptability [25].

```
1  (role xplayer)
2  (role oplayer)
3
4  (init (cell 1 1 b))
5  (init (cell 1 2 b))
6  (init (cell 1 3 b))
7  (init (cell 2 1 b))
8  (init (cell 2 2 b))
9  (init (cell 2 3 b))
10 (init (cell 3 1 b))
11 (init (cell 3 2 b))
12 (init (cell 3 3 b))
13 (init (control xplayer))
14
15 (<= (next (cell ?m ?n x))
16 (does xplayer (mark ?m ?n))
17 (true (cell ?m ?n b)))
18
19 (<= (next (cell ?m ?n o))
20 (does oplayer (mark ?m ?n))
21 (true (cell ?m ?n b)))
22
23 (<= (next (cell ?m ?n ?w))
24 (true (cell ?m ?n ?w))
25 (distinct ?w b))
26
27 (<= (next (cell ?m ?n b))
28 (does ?w (mark ?j ?k))
29 (true (cell ?m ?n b))
30 (or (distinct ?m ?j) (distinct ?n ?k)))
31
32 (<= (next (control xplayer))
33 (true (control oplayer)))
34
35 (<= (next (control oplayer))
36 (true (control xplayer)))
37
38 (<= (legal ?w (mark ?x ?y))
39 (true (cell ?x ?y b))
40 (true (control ?w)))
41
42 (<= (legal xplayer noop)
43 (true (control oplayer)))
44
45 (<= (legal oplayer noop)
46 (true (control xplayer)))
47
48 (<= open
49 (true (cell ?m ?n b)))
50
51 (<= (line ?x) (row ?m ?x))
52 (<= (line ?x) (column ?m ?x))
53 (<= (line ?x) (diagonal ?x))
54
55 (<= (row ?m ?x)
56 (true (cell ?m 1 ?x))
57 (true (cell ?m 2 ?x))
58 (true (cell ?m 3 ?x)))
59
60 (<= (column ?n ?x)
61 (true (cell 1 ?n ?x))
62 (true (cell 2 ?n ?x))
63 (true (cell 3 ?n ?x)))
64
65 (<= (diagonal ?x)
66 (true (cell 1 1 ?x))
67 (true (cell 2 2 ?x))
68 (true (cell 3 3 ?x)))
69
70 (<= (diagonal ?x)
71 (true (cell 1 3 ?x))
72 (true (cell 2 2 ?x))
73 (true (cell 3 1 ?x)))
74
75 (<= (goal xplayer 100)
76 (line x))
77
78 (<= (goal xplayer 50)
79 (not (line x))
80 (not (line o))
81 (not open))
82
83 (<= (goal xplayer 0)
84 (line o))
85
86 (<= (goal oplayer 100)
87 (line o))
88
89 (<= (goal oplayer 50)
90 (not (line x))
91 (not (line o))
92 (not open))
93
94 (<= (goal oplayer 0)
95 (line x))
96
97 (<= terminal
98 (line x))
99
100 (<= terminal
101 (line o))
102
103 (<= terminal
104 (not open))
```

Figure 2: GDL for Tic-Tac-Toe, retrieved from GGP.org [13]

Figure 20 shows the Game Description Language (GDL) corresponding to Tic-Tac-Toe. Lines 1-2 define two roles: `xplayer` and `oplayer`, representing players using `X` and `O` markers, respectively. Lines 4-13 define the initial state of a 3x3 board, where

each cell is initially marked as blank, and the game control is initially with `xplayer`, meaning the `X` player goes first. For example, line 4 states that the cell at `(1,1)` is initialized to blank at the start of the game.

State transitions in the game are defined by the `next` keyword, describing the rules for moving from the current state to the next state. Lines 15-36 describe that if it's the `X` player's turn, marking an empty cell will result in that cell being marked as `X` in the next state.

Lines 38-45 define the `legal` actions a player can take in a given state. For instance, in lines 38-40, if a cell is `blank` and the current `control` is with a player, that player can mark themself in that cell.

Lines 51-73 present the user-defined predicates, playing key roles in the game logic. These predicates help define the conditions for winning, legal moves, and the end state of the game. The predicates `row`, `column`, `diagonal` are used to determine if a player has successfully formed a line on a row, column, or diagonal, which are key conditions for winning. The predicate `line` is a more generalized predicate, used to determine if a player has successfully formed a line on any row, column, or diagonal of the board. If any of the row, column, or diagonal predicates hold true, then the line predicate also holds true, directly relating to the game's win/loss determination. The predicate `open` indicates that there are still empty spaces on the board. If at least one cell's status is `blank`, the game is not over, and players can still make moves. This predicate is crucial in determining whether the game has ended.

Lines 75-95 define the win/loss conditions through the `goal` keyword, indicating the end conditions of the game and the scores for the players.

- If a player completes a line (row, column, or diagonal), that player scores 100 points, and the opponent scores 0 points.
- If the board is filled and no player wins, both players score 50 points, indicating a

draw.

Lines 97-104 describe the termination conditions of the game, defining when the game ends. In Tic-Tac-Toe, the determination of win/loss conditions depends on the `line` and `open` predicates:

- If a player forms a `line`, that player wins.
- If the board is filled and no player wins (no `line` and `open` predicate holds), the game is a draw.

2.3.1.1 Game Description Language (GDL) Extentions

Due to the inefficiency of reasoning inherent in the Stanford specification of GDL, which limits its ability to handle certain types of games, among other issues, more research has been proposed to enhance its value in the research domain. The "Game Description Language for Incomplete Information" (GDL-II) was introduced by researchers in 2010 [35]. Its primary aim is to improve upon GDL so that it can describe games where players do not have complete information, meaning that certain game states or players' actions are not visible to other players. Typical examples of such games include poker and bridge, where players cannot see their opponents' cards.

GDL-II extends the capability of handling incomplete information games by introducing concepts such as observability, information sets, randomness, and private actions. Through specific syntax such as "sees" and "random," it allows for the description of players' limited observations of information, groups of game states with similar characteristics, random events in the game, and actions kept private from other players, thereby enhancing the ability to simulate player decision-making under limited information.

Building upon GDL-II, the same year saw the introduction of GDL-III (Game Description Language for Incomplete Information and Introspection) [36]. It further

expands this capability by allowing the description of the visibility of partial actions, indistinguishable game states for players, randomness factors within the game, as well as incorporating players' cognition and speculation about the game environment and other players' knowledge, thus providing a more accurate simulation of player decision-making processes under incomplete information.

Since we are currently only experimenting with board games, opting for the simplest initial version of GDL would be the best choice.

2.3.2 GDL Reasoners

In General Game Playing (GGP), the two most popular Game Description Language (GDL) reasoners are Prolog-based reasoner and proposition network-based reasoner.

The Prolog-based reasoner employs the Prolog (a programming language) to parse and execute game rules, utilizing logical reasoning to handle game states and actions. Since GDL is based on Datalog, the logical structure of GDL is similar to Prolog, allowing for direct interpretation and inference of GDL rules in Prolog. For instance, game rules in GDL can be represented as predicates in Prolog, and game state transitions can be expressed as rules in Prolog.

On the other hand, the proposition network-based reasoner represents game rules as a set of propositions and their logical relationships, utilizing a proposition network for inference and execution of the game. A proposition network is a way of representing knowledge as propositions and the relationships between them. Each proposition is a statement or assertion, usually represented in a formal logical language. These propositions are connected by relationships to form a network, with relationships being logical (such as "and", "or") or semantic (such as "belongs to", "contains"). The reasoner based on proposition networks uses this structure for reasoning, traversing the network of propositions and relationships to infer new information. This reasoning

can be based on logic or semantics. Such reasoners can be applied in various domains such as knowledge graph construction and querying, natural language understanding, recommendation systems, etc [30].

Both types of reasoners have their own characteristics and advantages, suitable for different types of games and application scenarios. The Prolog-based reasoner possesses strong logical expression capabilities and flexibility, suitable for handling complex logical rules and constraints, while the proposition network-based reasoner emphasizes intuitiveness and efficiency, having certain advantages in dealing with simpler and more intuitive game rules. Given that our research only requires GDL reasoner for dataset generation, we choose to use the Prolog-based reasoner.

2.3.3 Prolog, Swi-prolog and Pyswip

Prolog, short for "Programming in Logic," is a logic programming language primarily aimed at describing and solving problems using logical rules. In Prolog, programs consist of a series of logical facts and rules, which are referred to as predicates. Prolog programmers can query the system by posing queries, and the Prolog system attempts to match these queries with known facts and rules to determine their truth or falsehood [39].

```
% facts
gender(alice, female).
gender(bob, male).
gender(carol, female).
gender(dave, male).

% rules
brother(X, Y) :- gender(X, male), gender(Y, male), X \= Y.

% query
?- brother(X, Y).
```

Figure 3: An expample of Prolog segements

In this example, we first define some facts about people's genders and then define a rule to describe the relationship of brotherhood. The rule `brother(X, Y)` indicates that if `X` and `Y` are `males` and not the same person, then `X` is the `brother` of `Y`. Finally, a query `?- brother(X, Y)` is posed to find all `brother` relationships.

After running this query, the Prolog system attempts to match rules and facts to find the results that satisfy the query. For example, it may return `X = bob, Y = dave`, indicating that `Bob` and `Dave` are `brothers`.

SWI-Prolog is an open-source implementation of the Prolog programming language. It is a powerful and widely-used Prolog system that provides rich libraries and tools suitable for various application scenarios. SWI-Prolog is also a cross-platform system, capable of running on multiple operating systems, such as Windows, MacOS, and Linux [39].

PySWIP is a Python library for interacting with SWI-Prolog. It allows Python programmers to call Prolog programs from Python code and integrate Prolog's results into the Python environment, achieving seamless integration between Python and Prolog [32].

In this study, we primarily utilize Python and PySWIP to communicate with SWI-Prolog and conduct game reasoning, thereby generating datasets.

2.4 Machine Learning Framework

In this chapter, we will delve into various aspects of deep neural networks that are pertinent to our research. Firstly, we provide an overview of Neural Networks(NNs) and Fully Connected Neural Networks. Subsequently, we explore Graph Neural Networks (GNNs) and their variant models. Finally, we discuss some activation functions, loss functions, and optimization functions that we utilized in our research endeavors.

2.4.1 Nerual Networks (NNs)

Neural networks (NNs) consist of neurons, organized into layers. Each neuron receives input, performs a computation, and produces output, which may serve as input for other neurons. A typical NNs includes an input layer, one or more hidden layers, and an output layer. Figure 4 illustrates a simple neural network structure, including an input layer $x1, x2$, a hidden layer $h1, h2$, and an output layer y. Each arrow represents a connection from one neuron to another, corresponding to the weights in the network. The input layer $x1, x2$ receives the data. The hidden layer $h1, h2$ processes the data, transforming it via activation functions. The output layer y generates the final prediction result. Forward propagation and backpropagation are crucial for neural networks to learn from various datasets.

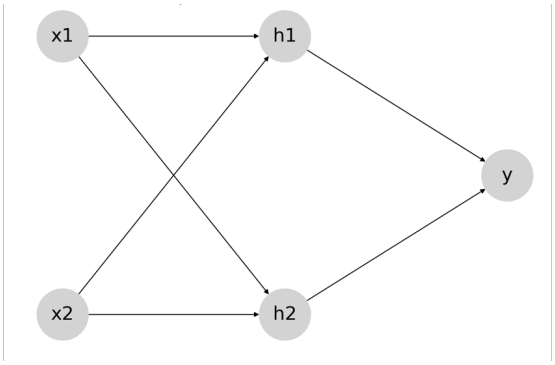

Figure 4: An expample Nerual Networks (NNs)

Forward propagation refers to the process in which data propagates through the network from the input layer to the output layer. At each neuron, the input values are multiplied by their corresponding weights, and the products are summed, possibly with the addition of a bias value. Finally, an activation function is applied to the sum to determine the neuron's output. The mathematical expression represented by Equation 1 illustrates this process, where $a^{(l+1)}$ represents the activation of the $l+1$ layer, σ denotes the activation function, and $W^{(l)}$ and $b^{(l)}$ are the weights and biases of the lth layer, respectively [8].

$$a^{(l+1)} = \sigma\left(W^{(l)}a^{(l)} + b^{(l)}\right) \tag{1}$$

Backpropagation is a fundamental algorithm for training neural networks. it adjusts the weights and biases within the network by following the gradient of the loss function. Initially, it computes the error at the output layer, represented as mathmatics 2, where C is the loss function, $\nabla_a C$ denotes the gradient of the loss function with respect to the activation, and σ' is the derivative of the activation function [8].

$$\delta^{(L)} = \nabla_a C \odot \sigma'\left(z^{(L)}\right) \tag{2}$$

This error is then backpropagated through the network, calculating the error for each layer as mathmatics 3.

$$\delta^{(l)} = \left(\left(W^{(l+1)}\right)^T \delta^{(l+1)}\right) \odot \sigma'\left(z^{(l)}\right) \tag{3}$$

Finally, the weights and biases are updated using an optimization algorithm like gradient descent, with updates described by mathmatics 4 and 5, where η represents the learning rate. This iterative process is crucial for the learning ability of the neural network [8].

$$W^{(l)} = W^{(l)} - \eta \nabla_{W^{(l)}} C \tag{4}$$

$$b^{(l)} = b^{(l)} - \eta \nabla_{b^{(l)}} C \tag{5}$$

Back to the figure 4, during forward propagation, the dataset from the input layer through the hidden layer to the output layer. Each neuron, upon receiving its input, computes its output based on its weights and biases and passes it through an activation

function to the next layer. In backpropagation, the error from the output layer is used to update the weights and biases throughout the network via gradient descent, minimizing the loss function.

This conceptual framework lays the groundwork for building more complex neural networks and deep learning models.

2.4.2 Fully-Connected Neural Networks

Fully-connected Neural Networks, also known as dense neural networks or dense networks, are fundamental architectures in deep learning. Each neuron in a dense layer receives inputs from all neurons in the previous layer and produces a single output value by computing the weighted sum of these inputs, followed by applying an activation function. The weights and biases associated with each connection are learned during the training process using optimization algorithms such as gradient descent.

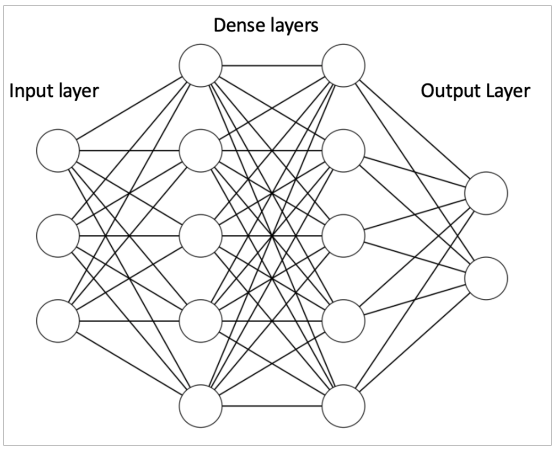

Figure 5: An expample of two dense layers Fully-connected Neural Networks

According to Deep Learning [8], The mathematical expression of Fully-connected Neural Networks typically involves two main aspects: the computation of weights and biases, and the application of activation functions.

1. Computation of weights and biases: Let $y_j^{(l)}$ denote the output of the j-th neuron

in layer l, the computation of this neuron can be expressed as:

$$y_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)} + b_j^{(l)} \tag{6}$$

Where $N^{(l-1)}$ is the number of neurons in the $(l-1)$-th layer, $w_{ij}^{(l)}$ is the weight between the j-th neuron in layer l and the i-th neuron in layer $(l-1)$, $x_i^{(l-1)}$ is the output of the i-th neuron in layer $(l-1)$, and $b_j^{(l)}$ is the bias of the j-th neuron in layer l.

2. Application of activation functions: For each neuron $y_j^{(l)}$ in the fully connected layer, its input is typically passed through an activation function $f(\cdot)$ to generate the final output $a_j^{(l)}$:

$$a_j^{(l)} = f(y_j^{(l)}) \tag{7}$$

Commonly used activation functions include ReLU, Sigmoid, Tanh, etc.

2.4.3 Rectified Linear Unit (ReLU)

ReLU (Rectified Linear Unit) is a commonly used activation function employed in the hidden layers of neural networks. It is defined as follows [8]:

$$f(x) = \max(0, x) \tag{8}$$

The distinctive feature of the ReLU function is that it outputs the same value as the input when the input is greater than zero, and outputs zero when the input is less than or equal to zero. This non-linear transformation enables neural networks to better learn complex non-linear relationships.

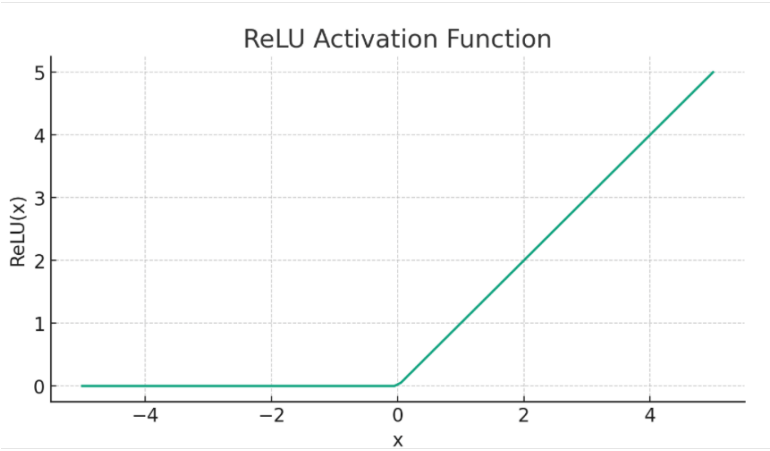

Figure 6: Rectified Linear Unit (ReLU)

2.4.4 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs), abbreviated as GNN, are deep learning models designed to process graph data. In contrast to traditional neural networks, they exhibit the capability to effectively handle unstructured and irregular data.

According to Wu et al. [43] and Kipf & Welling [12], the input to a graph neural network typically consists of a data structure composed of nodes and edges. Nodes represent entities or objects in the graph, while edges represent relationships or connections between nodes. Graphs can be categorized as directed or undirected, depending on whether edges have directionality. A graph is usually denoted as $G = (V, E)$, where V is the set of nodes and E is the set of edges. In graph structures, nodes and edges can have different attributes or features. For instance, in the context of GGP, each predicate can be seen as a node, while the relationships between predicates can be transformed into edges.

The principle of Graph Neural Networks (GNNs) is primarily based on the concept of message passing. It refers to the process where nodes exchange information through edges in the graph structure. In this process, the representation $\mathbf{h}_i^{(l)}$ of node v_i in the $l - th$ layer can be updated using the following formula:

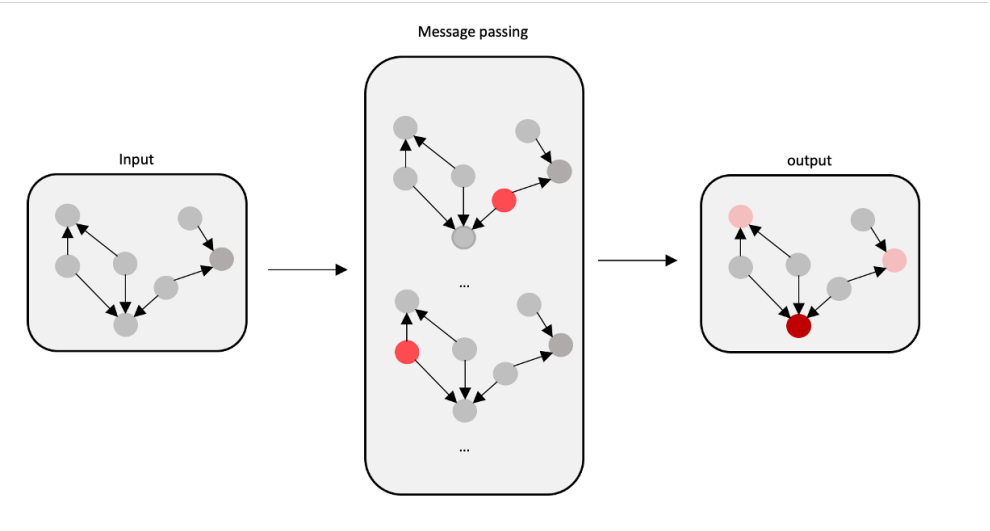

Figure 7: An example of message-passing GNN layer

$$\mathbf{h}_i^{(l)} = \text{Update}\left(\mathbf{h}_i^{(l-1)}, \text{ Aggregate}\left(\{\mathbf{h}_j^{(l-1)} \mid v_j \in \mathcal{N}(v_i)\}\right)\right) \tag{9}$$

$Aggregate$ is the aggregation function for neighboring node features, such as sum, average, or concatenation; $Update$ is the update function, which can be a fully connected layer, attention mechanism, etc. After learning the node representations, the entire graph representation can be obtained by aggregating all node representations. A common approach is to take the average of all node representations or assign different importance to different nodes using attention mechanisms. Finally, the learned graph representation can be utilized for various tasks, such as node classification, graph classification, link prediction, etc.

Figure 7 illustrates the message passing process of a GNN layer, where the red nodes are aggregating information along the edges' directions.

2.4.5 Graph Attention Neural Networks (GAT)

The Graph Attention Network (GAT) [37] is a type of graph neural network model designed to handle graph-structured data, where relationships between nodes can be

arbitrary and complex. GAT utilizes an attention mechanism to learn relationships between nodes and weights nodes based on their importance in the graph, enabling more flexible and precise learning of graph structure representations.

The main principle of GAT is to adaptively assign different attention weights to each node to aggregate information from neighboring nodes. Given a graph $G = (V, E)$, for a node v_i, with feature representation $\mathbf{h}_i$, the attention mechanism used by GAT is defined as follows:

$$e_{ij} = \text{LeakyReLU}(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]) \tag{10}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})} \tag{11}$$

$$\mathbf{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{W}\mathbf{h}_j \right) \tag{12}$$

e_{ij} represents the relevance between node v_i and its neighbor v_j, α_{ij} is the attention weight normalized by the softmax function, $\mathcal{N}(i)$ denotes the neighbor set of node v_i, $\mathbf{a}$ is the learnable parameter of the attention mechanism, $\mathbf{W}$ is the weight matrix, $\|$ denotes the vector concatenation operation, and σ is the activation function, typically ReLU. Finally, each e_{ij} is used as an input to the Update function in Equation (9) to aggregate information from neighboring nodes.

2.4.6 Graph Attention Neural Networks V2 (GATv2)

Graph Attention Neural Networks V2 (GATv2) [2] is an improved version of the original Graph Attention Network (GAT), designed to enhance the performance and efficiency of the model by refining the attention mechanism.

GATv2 introduces a sparse attention mechanism to address the attention computation issue in large-scale graphs. Traditional GAT requires computation for all nodes in the graph when calculating attention weights, which can lead to significant computational and storage overheads in large-scale graphs. However, GATv2 addresses this by selecting a small subset of nodes as candidate sets and computing attention weights only for these nodes and the target node, thus reducing the computational and storage overhead.

Moreover, GATv2 adopts an adaptive attention head selection mechanism, dynamically selecting the most appropriate attention heads based on the features of different nodes and the graph structure. This enhances the model's generalization capability and adaptability. These improvements make GATv2 advantageous in handling large-scale graph data and improving model performance.

2.4.7 Transformer-based GNNs (TransformerConv)

TransformerConv [27] employs the attention mechanism of Transformer to handle graph-structured data. In traditional GNNs, nodes typically interact only with their neighboring nodes, whereas Transformer-based GNNs allow each node to directly interact with all other nodes in the graph, thereby capturing more global graph information. The computational complexity of Transformer-based GNNs is higher because they allow each node to interact with all other nodes, which may lead to performance issues on large graphs. Therefore, they are typically more suitable for tasks that require consideration of the global graph structure.

Given an input feature matrix X representing the node features and an adjacency matrix A representing the graph structure, the output feature matrix H of the TransformerConv layer can be calculated as:

$$H = \text{softmax}\left(\frac{X \cdot \text{diag}(A) \cdot X^T}{\sqrt{d}}\right) \cdot X \cdot W \tag{13}$$

where $\cdot$ denotes matrix multiplication, $\text{diag}(A)$ represents the diagonal matrix of A, X^T is the transpose of X, W is a learnable weight matrix, d is the dimension of the feature space.

This formula captures the attention mechanism in TransformerConv, where each node attends to all other nodes based on their pairwise similarity computed through the attention mechanism, and then aggregates the attended features weighted by the attention scores to produce the output feature matrix H.

2.4.8 Jumping Knowledge

Jumping Knowledge [41] is a mechanism used in graph neural networks (GNNs) to propagate information within graph-structured data and enhance model performance. The main idea behind Jumping Knowledge is to utilize information from intermediate nodes to update the representation of target nodes, thereby capturing global information within the graph more effectively. Specifically, Jumping Knowledge combines the representations of intermediate nodes with that of target nodes to generate new representations for the target nodes, enriching their information. This mechanism assists the model in better understanding graph-structured data and improving its performance in tasks such as node classification and graph classification. The mathematical formulation of Jumping Knowledge can be represented as:

$$h_v^{(l+1)} = \sigma\left(W_{\text{jump}} \cdot \text{CONCAT}\left(h_v^{(l)}, \frac{1}{|N(v)|}\sum_{u \in N(v)} h_u^{(l)}\right)\right) \tag{14}$$

where $h_v^{(l)}$ is the representation of target node v at layer l, $N(v)$ is the set of neighboring nodes of node v, σ is the activation function, and W_{jump} is the weight

matrix.

2.4.9 Global Attention pooling

Li, Tarlow, Brockschmidt, and Zemel [14], proposed a mechanism used to aggregate information from all nodes in the graph into a single representation termed as Global Attention pooling. This pooling operation is typically applied after the message passing phase in GNNs and helps in capturing global graph-level information for downstream tasks. One common formulation of global attention pooling is as follows:

$$Z = \sigma\left(\sum_{i=1}^{N} \alpha_i h_i\right) \tag{15}$$

Where, Z is the output representation of the pooled graph.σ is an activation function. N is the number of nodes in the graph. h_i is the representation of node i. α_i is the attention weight assigned to node i, typically computed using a learned attention mechanism.

2.4.10 Gaussian Error Linear Unit (GeLU)

Gaussian Error Linear Unit (GeLU) [11] is an activation function designed to provide smoother non-linear transformations for the hidden layers of neural networks. The definition of the GELU function is as follows:

$$\text{GELU}(x) = x \cdot \Phi(x) = \frac{1}{2}x \cdot \left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \tag{16}$$

$\Phi(x)$ represents the aggregated distribution function of the standard normal distribution, and $\text{erf}(x)$ is the error function. The GeLU function approximates a linear transformation near $x = 0$ and possesses a soft saturation characteristic, enabling more

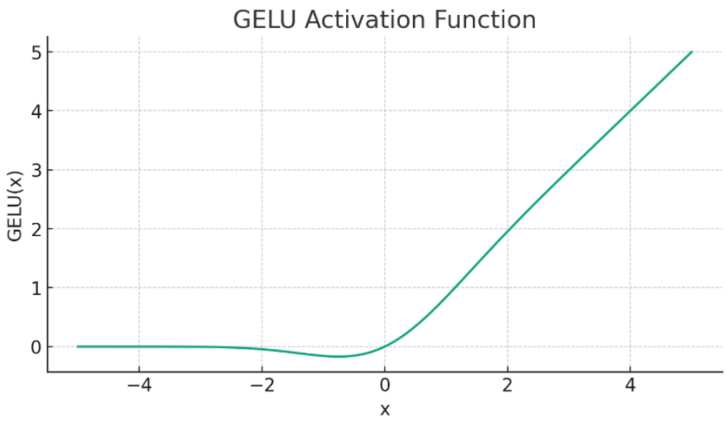

Figure 8: Gaussian Error Linear Unit (GeLU)

effective gradient propagation during the training of deep neural networks. When training deep neural networks, particularly graph neural networks, vanishing or exploding gradients are common issues. The smooth nature of GeLU means that gradients are preserved even near small input values, rather than abruptly dropping to zero as with ReLU. This helps in propagating gradients more effectively, thereby improving the training process of the network.

GeLU has been widely used in the field of deep learning, particularly in tasks such as natural language processing and image processing. It has been shown to outperform common activation functions like ReLU in certain cases, leading to improvements in the performance and convergence speed of neural networks.

2.4.11 Focal Loss

Focal Loss is a loss function proposed by Tsung-Yi Lin et al. [15] in 2017 to address class imbalance issues. When dealing with class imbalance problems, traditional cross-entropy loss may lead the model to overly focus on easily classified samples, thus ignoring those that are harder to classify. Focal Loss reduces the weight of easily classified samples to make the model pay more attention to difficult-to-classify samples, thereby improving the model's performance. The definition of Focal Loss is as follows:

$$\mathrm{FL}(p_t) = -(1 - p_t)^{\gamma} \log(p_t) \tag{17}$$

Here, p_t represents the probability assigned by the model to a sample belonging to the positive class, and γ is an adjustable parameter typically ranging from 0 to 5. When a sample is correctly classified, p_t approaches 1, causing the loss function to be suppressed by $(1 - p_t)^{\gamma}$, reducing the weight of easily classified samples. Conversely, when a sample is misclassified, the loss function increases.

Focal Loss has shown significant effectiveness in tasks such as object detection and image classification, and it plays an important role in addressing class imbalance and hard sample learning issues.

2.4.12 Gradient Explosion and Vanishing

Gradient explosion and vanishing gradient are common issues in deep learning that significantly impact the training stability and convergence speed of neural networks. Gradient explosion refers to gradients becoming extremely large during backpropagation, leading to overly aggressive weight updates that hinder effective model training. Conversely, vanishing gradient occurs when gradients diminish to very small values or even approach zero as the network deepens, preventing parameter updates and hindering the learning of meaningful feature representations. Both of these problems can result in training failure or very slow convergence. To address these issues, researchers have proposed various methods, including gradient clipping, the use of specific activation functions (such as ReLU), appropriate parameter initialization, among others, to alleviate gradient explosion and vanishing gradient problems [22].

2.5 GNN-Reasoner

The GNN-Reasoner, introduced by Alvaro et al. [9] in 2022, aims to redefine how game rules and states are represented for reinforcement learning, along with the associated inferencing tasks. Leveraging the capabilities of Graph Neural Networks (GNNs), the GNN-Reasoner facilitates a deeper understanding of the game environment, allowing it to learn and infer game states more accurately and robustly when transferring knowledge across various gaming contexts.

The GNN-Reasoner introduced a graph-based representation method called instantiated rule graphs (IRGs) for describing game states in the Game Description Language (GDL). This representation method effectively captures the complex relationships between game states and rules. Additionally, it presents a novel approach for generating samples and datasets, framing the GDL inference task as a neural network-based machine learning problem.

In this chapter, we will first describe the main components of the GNN-Reasoner, including Rule Graphs, Instantiated Rule Graphs, vector embeddings, and GNNs structure. We will finally discuss the current limitations of the GNN-Reasoner. As this research is based on the GNN-Reasoner, we provide only an introduction to avoid repetition.

2.5.1 Rule Graph

The Rule Graph [9] serves as a visual representation of GDL rules, encoding their relationships and structures for reasoning purposes. Nodes within the graph generally correspond to segments of complete rules, while edges signify connections and dependencies among these rules. By using the rule graph, the GNN-Reasoner can understand complex interactions between different rules, enabling informed decision-making or predictions based on them.

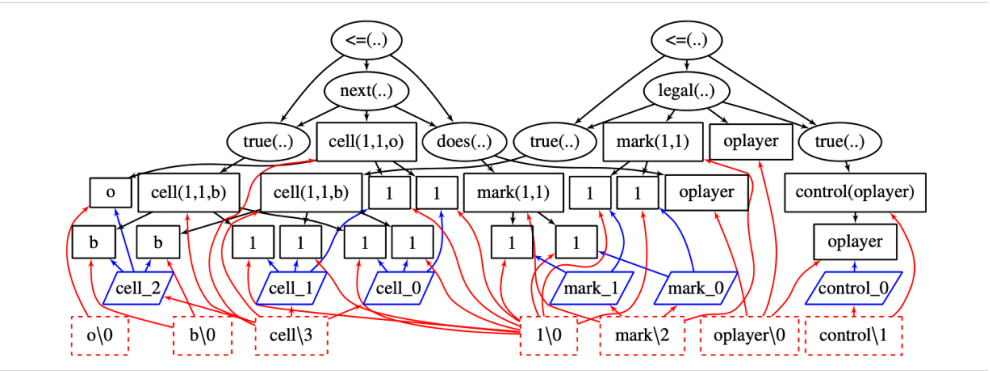

Figure 9: An expample of Rule Graph, Retrieved from GNN-Reasoner [9]

The Rule Graph consists of keyword nodes, expression nodes, symbol nodes, and symbol argument nodes. Keyword nodes represent logical and relational sentences based on predefined GDL keywords like legal, next, does, etc., with each keyword corresponding to a specific node type. Expression nodes denote sentences containing non-keyword nodes, such as custom-defined predicates or functions for the game. Symbol nodes serve to identify expressions consistently throughout the graph without explicit lexical labels, linked to instances of the expressions they represent. Symbol argument nodes encode positional information within expressions, establishing connections between argument instances and their positions in the graph. Edges in the graph construct syntax trees by connecting expressions with their arguments. The part of the Rule Graph generated from the dataset Tic-Tac-Toe is depicted in figure 9.

2.5.2 Instantiated Rule Graphs (IRGs) and Vector Embedding

While GDL offers a robust framework for game rules, it lacks a universal state representation. The Instantiated Rule Graph (IRG) [9] extends the concept of rule graphs by adding the information of games states directly into the node embedding.

In order to facilitate the Instantiated Rule Graphs (IRGs) in neural network models, GNN-Reasoner adopt a vector embedding [9] approach to encode both node types and their corresponding labels. Each node n in the instantiated rule graph $I = (V, E, T, L)$

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

Figure 10: Vector embedding value, Retrieved from GNN-Reasoner [9]

is represented by a vector embedding $\vec{v}_n \in 0, 1^{19}$, ensuring a uniform representation across various node types and labels. Table 10 shows the node feature that each value $\vec{v}_{1...19}$ corresponds to.

2.5.3 GNNs Structure within GNN-Reasoner

The neural network architecture of GNN-Reasoner consists fully connected input embedding layers, Jumping knowledge, GAT Block layers employing Graph Attention Networks (GAT) [37], Global Attention Pooling [14] and final fully connected output layers. This GNN framework as shown in figure 11, operates on input instantiated rule graphs (IRGs), employing node vector embeddings to iteratively update node representations through message propagation within the GAT Blocks. Utilizing Jumping Knowledge and global attention mechanisms, the model aggregates final node embeddings into a graph-level representation, facilitating the prediction of node probabilities.

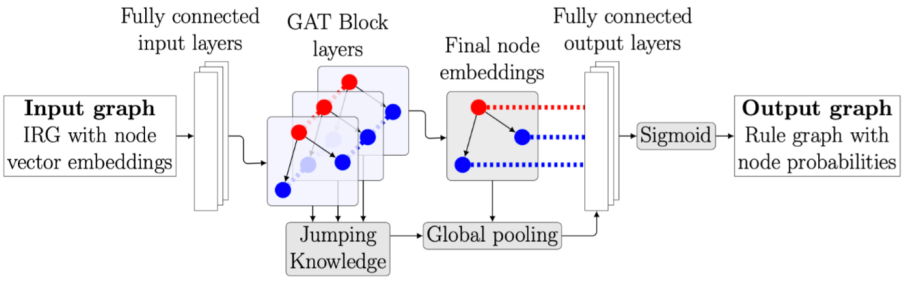

Figure 11: GNN Structure, Retrieved from GNN-Reasoner [9]

The structure of the GAT Blocks, illustrated in Figure 12, incorporates bidirectional edge layers alongside skip connections and ReLU activation. The Graph Attention Networks (GAT) [37] was applied by their balance between memory usage and performance.

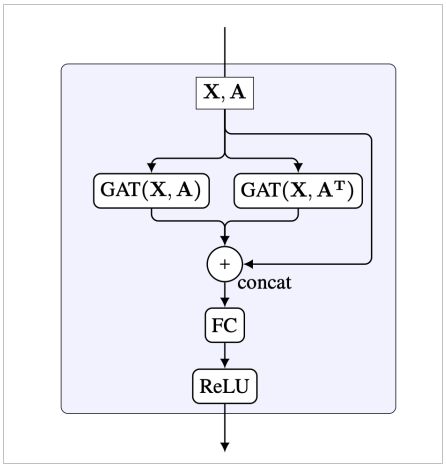

Figure 12: GAT Structure, Retrieved from GNN-Reasoner [9]

2.5.4 Limitations of GNN-Reasoner

Although the GNN-Reasoner excels in predicting Legal actions and next fluents, it lacks the complete functionality of a standard GGP-Reasoner, such as handling terminal states and goals. Moreover, its Rule Graph follows the same order as GDL rules, preventing the model from effectively learning dataset features. To address this issue, the GNN-Reasoner employs a method using dual GAT blocks as described in section 2.5.3. This approach utilizes two different GAT blocks to handle default Rule Graphs and Rule Graphs with reverse edges. While this allows the GNN-Reasoner to function, it incurs significant computational overhead, restricting its usability to games with relatively small state spaces.

The GNNs architecture, particularly with the addition of terminal and goal predictions, faces further challenges. Firstly, the increased complexity of the dataset due to

these additional predictions hampers the model's performance with smaller batch sizes. Secondly, the computational demands arising from the dual GAT blocks architecture, along with the intricacies of the dataset, require training on a machine with a minimum of 40GB of GPU memory. However, the neural network can only training for the most basic Tic-Tac-Toe. Even with larger GPU memory, such as A100 with 80GB, only a small number of simple games can be used for neural network training.

Beyond these challenges, the insufficient of dataset features prevents the model from effectively learning terminal and goal predictions, resulting in poor accuracy across most GGP games. Additionally, the dataset suffers from issues related to long-range dependencies and negation problems, making it difficult for the model to accurately predict terminal states and goal values. Finally, the Rule flattening method used by the GNN-Reasoner fails to adequately address negation problems. This limitation makes the neural network can't classify those rules.

In conclusion, the GNN-Reasoner requires a enhanced, smaller-sized graph structure, a GNN architecture that minimizes hardware resource consumption without compromising performance, and a more effective and efficient Rule flattening method.

2.6 Conclusion

In this chapter, we provide an overview of previous research relevant to our study. We begin by discussing the history of computer games and AI. Following this, we delve into the history and components of General Game Playing (GGP), with a detailed focus on the Game Description Language (GDL) and Logic Reasoners, which are essential to our research. Additionally, we introduced a range of neural networks and architectures from others previous research as a background, followed by a detailed exploration of the GNN-Reasoner [9]. The concepts presented in this chapter will be elaborated and expanded upon in subsequent chapters.

Chapter 3

Method

3.1 Introduction

This study builds upon the GNN-Reasoner proposed by Alvaro et al. [9] and introduces a series of improvements to create an enhanced version called: GGP-Reasoner-v2. These enhancements are primarily focused on four areas: optimization of graph structures, dataset generation process, analysis tools, and adjustments to the neural network architecture. This chapter will detail these improvements.

1. **Graph Structures:** We proposed a new Annotated Rule Graph, a new Reversed Instantiated Rule Graph labelling function, a new dataset vector embedding method, a novel Graph-Based Rule Flattening method, and a series of improvements related to enable Terminal and Goal prediction. The new graph structure facilitates the neural network in better learning dataset features.

2. **Dataset Generation:** We have re-implemented the code for the GNN-Reasoner [9] and introduced a new data structure to efficiently store information from the dataset. Additionally, leveraging the advantages of multi-core processors, we have accelerated the dataset generation speed 10 times faster. These modifications

not only speed up the dataset generation process but also facilitate our research. This allows us to analyze the impact of different graph structures on the neural network more quickly.

3. **Analysis Tools:** We have improved and created a series of new analysis and generation tools to make the experimental process smoother and more convenient. These tools enable our research to automate dataset generation, training, analysis, and saving of experimental data with a single click.

4. **Neural Network Architecture:** In response to the GNN-Reasoner's [9] limitations in processing specific types of graph data (specifically only targeting Legal and Next) and high computational resource consumption, we made adjustments and optimizations to the neural network architecture. These adjustments include modifying the configuration of network layers, using alternative activation functions and loss functions. These improvements aim to enhance the neural network's learning capabilities and improve its performance in complex tasks.

Through these improvements, we expect the GNN-Reaonser-v2 to be more effective in processing complex graph data, providing more accurate predictive results, and thus demonstrating greater application potential in the field of General Game Playing (GGP). In the following chapters, we first provide a detailed description of the dataset generation process, followed by an explanation of the neural network's architecture and training parameters. Lastly, we describe a series of neural network analysis tools.

3.2 An Overview of Dataset

Firstly, we present an overview of the dataset for GNN-Reasoner-v2. Figure 13 illustrates the process of dataset generation, using Tic-Tac-Toe as an example. The procedure starts with the program executing multiple distinct simulations that each stores a unique

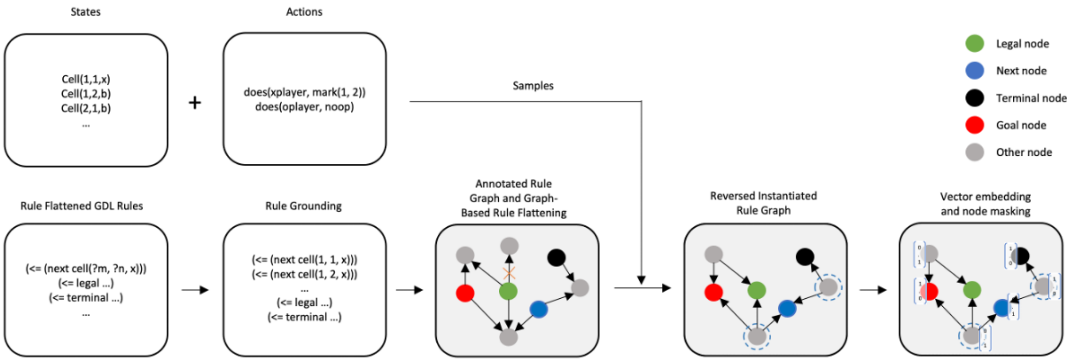

Figure 13: Dataset generation process

game state and the corresponding player move. During this phase, all rule flattened rules within the `tictactoe.kif` file are grounded through Prolog-based rule grounding. Subsequently, the grounded GDL rules are transformed into a directed graph termed as Annotated Rule Graph. Through Graph-Based Rule Flattening, unnecessary nodes are eliminated. The rule flattened Annotated Rule Graph is then labeled with instantiated labelling function, and all edges are reversed, resulting in the creation of Reversed Instantiated Rule Graph (R-IRG). Finally, the R-IRG undergo a vector embedding process to convert them into a format compatible with neural network input. Additionally, node masking is applied to assign a mask matrix to different node types. The details of the above process will be elaborated in the following sections.

3.2.1 Manual Rule Flattening

The majority of games described in the Game Description Language (GDL) are manually created, with their initial designs not specifically intended for use by neural networks. All original GDL files for the games in our research are sourced from the GitHub repository GGP.org [13]. These GDLs always involve a significant number of vocabularies after Rule Grounding, resulting in the generation of large graphs. We applied the same approach from the GNN-Reasoner [9], called Manual Rule Flattening. It directly

modify the GDL in order to address these challenges.

Figure 14 shows an example of the Rule Flattening. The original Terminal GDL rule includes a dependency on `line(x)`, which in turn depends on `row(?m, ?x)`, and `row(?m, ?x)` contains multiple `True` keywords. Those rules can be migrated to a single rule manually on the right side of Figure 14.

```
(<= Terminal
     line(x) )

(<= line (x)
    row (?m, ?x) )

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

```
(<= Terminal
    true (cell (?m, 1, ?x) )
    true (cell (?m, 2, ?x) )
    true (cell (?m, 3, ?x) ) )
```

Figure 14: Example of Rule Flattening (merge three rules into one single rule)

3.2.2 Samples Generation

The Samples Generation process is conducted using SWI-Prolog [39] and pyswip to perform the inferencing tasks [32]. Algorithm 1 presents our general implementation, where a game description G is utilized to produce d game samples S_v comprised of tuples of the form $S_{\text{legal}} = (p_0, l_0), \ldots, (p_n, l_n)$ representing legal actions for each player, S_{next} for the subsequent states, S_{terminal} for terminal states, and $S_{\text{goal}} = (p_0, g_0), \ldots, (p_n, g_n)$ for the score of each player.

This general algorithm is capable of generating all GDL game samples. For instance, in Tic-Tac-Toe, a game state might be represented as `(control(xplayer), cell(1,1,o), ... cell(3,3,b))`, indicating the current move belongs to xplayer and oplayer has positioned a piece at `(1,1)`. A joint action `does(xplayer, mark(1, 2))` indicates that xplayer placed a piece at `(1,2)` in the

current round. Next states `(control(oplayer), cell(1,1,o), cell(1,2,x)...)` refers to the next rounds's state. A Legal fluent `(<= (legal xplayer (mark 3 3))` refers to player x is legal to mark a position at `(3,3)`. Finally, at the end of the game, A score represented as `(goal oplayer 100), (<= (goal xplayer 0)` indicats that oplayer wins and signifies the game's end. Terminal state is reprensented as true or false.

Algorithm 1 Samples Generation algorithm

function GenerateSamples(Rule Flattened Game description G, number of examples d)
$S_v \leftarrow \varnothing$ ▷Initialise empty set of collected examples
$P \leftarrow p_0, \ldots, p_k$ = GETROLES(G)
while $|S_v| < d$ **do**
  $S \leftarrow f_0, \ldots, f_i$ = GETINITIAL(G)
  $End \leftarrow true$
  while End **do**
    if not GETTERMINAL(G, S) **then**
      $S_{\text{legal}} \leftarrow (p_0, a_0) \ldots (p_n, a_n)$ = GETLEGALACTIONS(G, P, S)
      $a \leftarrow_{\in R} S_{\text{legal}}$ ▷Randomly select a legal action
      $S_{\text{next}} \leftarrow$ GETNEXT(G, S, a)
      $S_{\text{terminal}} \leftarrow false$
      $S_{\text{goal}} \leftarrow$ GETGOAL(G, S, P)
      $S \leftarrow S_{\text{next}}$
    else
      $S_{\text{terminal}} \leftarrow true$
      $S_{\text{goal}} \leftarrow (p_0, g_0) \ldots (p_n, g_n)$ = GETGOAL(G, S, P)
      $End \leftarrow false$
    end if
    if $(S, a, S_{\text{legal}}, S_{\text{next}}, S_{\text{terminal}}, S_{\text{goal}}) \notin S_v$ **then**
      $S_v \leftarrow S_v \cup \{(S, a, S_{\text{legal}}, S_{\text{next}}, S_{\text{terminal}}, S_{\text{goal}})\}$
    end if
  end while
end while
return S_v

Due to the unbalanced distribution of different types of states in a single game, we have implemented a strategy to balance the distribution of states in our dataset. For instance, in Tic-Tac-Toe, the majority of turns occur in a playable state, but terminal

states are only relevant at the end of a round. Therefore, to achieve a balanced dataset, we duplicated the data associated with terminal and goal states to match the number of Legal actions and Next states.

3.2.3 Rule Grounding

A standard Game description language (GDL) includes a series of variables beginning with `?`. In General Game Playing (GGP), a logic-based reasoner (such as Prolog) automatically finds replacement values for these variables and facilitate reasoning tasks based on this replacement value. This process is referred to as rule grounding. For instance, in the first GDL statement on the right side of figure 15, `?w` is substituted with `oplayer`. In logic reasoning tasks based on neural networks, the output of the network is a matrix representing the probability distribution associated with various nodes. This implies that the dataset must ground all GDL rules before pass into the neural network.

```
(<= (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 15: Rule grounding

Figure 15 presents an example illustrating the comparison between a standard GDL (Game Description Language) rule that has been grounded. We adopted the same approach as GNN-Reasoner [9], which originally comes from a Prolog-based method with tabling [39] to efficiently ground these GDL rules, which also generates all possible fluents and legal actions.

3.2.4 HB Predicate Finding

In this research, we introduced the HB predicate (a predicate appearing in the head and body of different rules), a type of user-defined predicate extended from Game Description Language (GDL). It combines the idea of user-defined function predicates from GDL with the concept of functions commonly used in programming languages such as C. For instance, in the GDL rules `(<= terminal (not open))` and `(<= open (true(cell 1 1 b)))`, it contains an HB predicate `open`. The first `open` refers to the HB predicate (body), and the second `open` indicates HB predicate (head). The HB predicate (head) can be analogous to a function in C, whereas the HB predicate (body) refers to the portion of the program where the Head function is invoked.

Definition 1 (HB Predicate) *Given a game description G, and an empty set HB, for all rules $h_1 \leftarrow b_1^1 \wedge \ldots \wedge b_{k_1}^1, \ldots, h_n \leftarrow b_1^n \wedge \ldots \wedge b_{k_n}^n$ in G. If $h_i = b_l^j$ occurs, and $h_i \notin$ HB, then HB $\leftarrow$ HB $\cup \{h_i\}$. If a term $b_n(d_1, \ldots, d_n)$ occurs in a rule in G, the algorithm continues to recursively traverse $d_1, \ldots, d_n$ until completion.*

3.2.5 Annotated Rule Graph

In this study, we introduced a modified Rule Graph, termed as Annotated Rule Graph, which extends the methodology of the GNN-Reasoner [9] by incorporating additional node types, and some reversed edges. It is used to represent the relationships between various predicates in GDL. The process of constructing the Annotated Rule Graph is divided into two parts. Firstly, establishing the basic graph structure called Base Annotated Rule Graph, followed by utilizing Graph-Based Rule Flattening to construct the final Annotated Rule Graph.

The Annotated Rule Graph encompasses various types of nodes, including:

- Keyword nodes: Representing predefined terms K utilized in the Game Description Language (GDL).

- Expression nodes: Consisting of predicates and functions that do not include keywords.
- Constant nodes: Corresponding to constants generated after grounding.
- HB predicate nodes: A further type of predicate node HB, serving for Rule Flattening. It is also classified as HB head predicate nodes hb_head and HB body predicate nodes hb_body. This is to capture the predicates which can be removed after rule flattening.
- Symbol constant nodes: These nodes act as connectors between different predicate nodes, highlighting their distinctions.
- Symbol argument nodes: These nodes identify the positions of constants within predicate and keyword nodes.

$$K = \left\{ \begin{array}{c} \text{<=, legal, next, terminal,} \\ \text{goal, distinct, does, init, role,} \\ \text{true, input, base, not, or} \end{array} \right\} \tag{18}$$

Definition 2 (Base Annotated Rule Graph for a game description G) *Given a game description G, the Base Annotated Rule Graph of game G is a coloured, directed graph $R_A = (V, E, T)$. V is the set of nodes, E is the set of edges and $T : V \rightarrow K \cup$ {hb_body, hb_head, constant, expression, symbol_const, symbol_arg} 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 Σ is the set of ground terms present in G. V, E and T are defined as follows:*

1. ***Implication nodes(<=) [9]:*** *For rule $h \leftarrow b_1 \wedge \ldots \wedge b_n$ in G, we construct nodes $v, h, b_1, \ldots, b_n$ such that $F_G(v) = h \leftarrow b_1 \wedge \ldots \wedge b_n$, $F_G(h) = h$, $F_G(b_1) =$*

$b_1, \ldots, F_G(b_n) = b_n$, *and the edges* $(v, h), (v, b_1), \ldots, (v, b_n), (h, b_1), \ldots, (h, b_n)$ *in* E. *Node type* $T(v)$= *implication and the types of* $h, b_1, \ldots, b_n$ *will be given below.*

2. ***Keyword nodes [9]:*** *If a term* $k(t_1, \ldots, t_n)$ *occurs in a rule in* G *and* $k \in K$ *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* $T(k) = k$ *and lexical mappings* $F_G(t_1) = t_1, \ldots, F_G(t_n) = t_n$. *The node types of* $t_1, \ldots, t_n$ *are given by applying this definition recursively.*

3. ***Symbol nodes*** [1] *For every* n*-ary relation and function symbol* $p \notin K, HB$ *in* G, *we construct symbol nodes* $s, s^1, \ldots, s^n \in V$, *construct edges* $(s^1, s), \ldots, (s^n, s) \in E$, *assign node types* $T(s)$ = *symbol_const,* $T(s^i)$ = *symbol_arg and assign lexical mappings* $F_G(s) = p$ *and* $F_G(s^i) = p_i$.

4. ***Expression nodes:*** [2] *If a term* $p(t_1, \ldots, t_n)$ *occurs in a rule in* G *and* $p \notin K$, *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* $F_G(s) = p$ *and* $F_G(s^i) = p_i$, *we construct edges* $(p, t_1), \ldots, (p, t_n), (s, p), (t_1, s^1), \ldots, (t_n, s^n)$ *in* E. *We assign node types* $T(p)$ = *expression and lexical mappings* $F_G(p) = p(t_1, \ldots, t_n)$, $F_G(t_1) = t_1, \ldots, F_G(t_n) = t_n$. *The node types of* $t_1, \ldots, t_n$ *are given by applying this definition recursively.*

5. ***HB predicate nodes:*** *For all rules* $h_1 \leftarrow b_1^1 \wedge \ldots \wedge b_{k_1}^1, \ldots, h_n \leftarrow b_1^n \wedge \ldots \wedge b_{k_n}^n$ *in* G *with constructed nodes* $v_1, h_1, b_1^1, b_{k_1}^1, \ldots, v_n, h_n, b_1^n, b_{k_n}^n$, *if* $h_i = b_l^j$ *occurs and* $hi \in HB$ *or* $b_l^j \in HB$, *we redefine their node type as* $T(h_i)$ = *hb_head,* $T(b_l^j)$ = *hb_body. If a term* $b_n(d_1, \ldots, d_n)$ *occurs in a rule in* G, *the node type of* $(d_1, \ldots, d_n)$ *are given by applying this definition recursively.*

[1] Note that this is different from [9] on nodes that are part of HB and reversed edges
[2] Note that this is different from [9] on edges such as (t_1, s^1).

6. ***Constant nodes:*** *If a term* $p(t_1, \ldots, t_n)$ *occurs in a rule in* G *that node type* $T(t_i) \notin \{k, hb\_head, hb\_body\}$*, we redefine its node type as* $T(t_i)$ = *constant.*

All nodes are interconnected through dependency relationships as defined in GDL. The above definition also describes the algorithm for generating Base Annotated Rule Graph. It is important to prioritize the generation of expression nodes, followed by HB predicate nodes, and finally constant nodes.

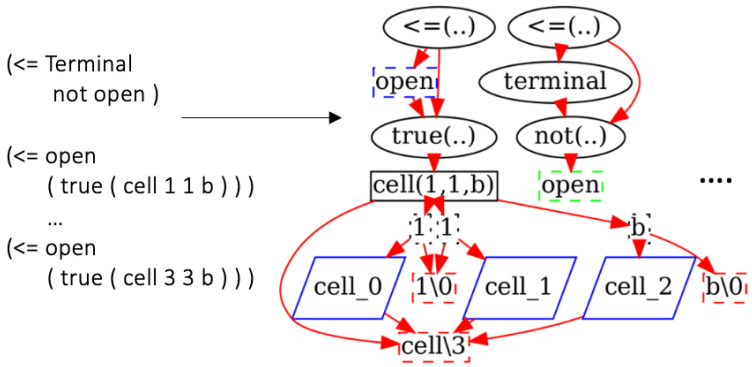

Figure 16: Example of Base Annotated Rule Graph

Figure 16 illustrates the Base Annotated Rule Graph derived from the transformation of `(<= terminal (not open))` and its related dependencies `(<= open (true(cell 1 1 b)))`. It's essential to emphasize that the lexical labels in Figure 16 are only for visualization purposes. Elliptical nodes represent keyword nodes, while rectangular nodes denote expression nodes, encompassing the expression node `cell`, the HB predicate `open`, and all constant nodes (dotted black) within the expression nodes. Red rectangular nodes represent symbol nodes. In this instance, two constant nodes `1` are linked to the symbol node `1\0`, indicating their identical node type. The node `open` (dotted blue) is further defined as a HB head predicate node. The node `open` (dotted green) is further defined as an HB body predicate node. Lastly, blue nodes denote symbol argument nodes. For instance, the constant node `1` on the left is

linked to the symbol argument node `cell_0`, signifying its position as the first variable in the expression node `cell(1,1,b)`.

3.2.5.1 Graph-Based Rule Flattening

The Manual Rule Flattening mentioned in section 3.2.1 is effective only for non-negation rules, as certain rules in Game description language (GDL) cannot be automatically flattened by Prolog. To address this issue, we proposed a new method for handling rules with negation problems: Graph-based Rule Flattening. This approach achieves Rule Flattening by directly modify the graph structure by removal and addition of nodes and edges. Figure 17 shows the final Annotated Rule Graph with a rule where `(<= terminal (not open))` depends on `(<= open (true (cell 1 1 b))),...,(<= open (true (cell 3 3 b)))`. The Graph-based rule flattening directly connects node `not` to nodes `(true(cell 1 1 b)), ...(true(cell 3 3 b))` without the need of HB predicates node `open`.

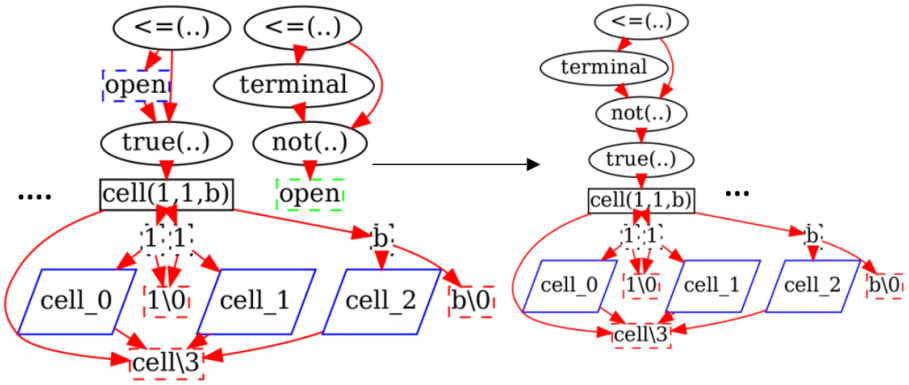

Figure 17: Base Annotated Rule Graph (left) and Annotated Rule Graph constructed by Graph-Based Rule Flattening (right)

Algorithm 2 outlines the process of Graph-based Rule Flattening. Given an Annotated rule graph $R_A = (V, E, T)$, it traverses all nodes in V representing HB predicates and eliminates any associated symbol constant nodes. Subsequently, it iteratively handles pairs of HB head and body predicate nodes. For each corresponding pair, it

Algorithm 2 Graph-Based Rule Flattening

function GraphBasedRuleFlattening(AnnotatedRuleGraph G_A))
$(V, E, T) \leftarrow G_A$
$edge \leftarrow \varnothing, nodes \leftarrow \varnothing$
$hb\_heads, hb\_bodies \leftarrow$ GETHBPREDICATENODES(V)
$constants \leftarrow$ GETCONSTANS(V)
REMOVESYMBOLCONSTS(hb_heads, hb_bodies)
for $head$ in hb_heads **do**
  for $body$ in hb_bodies **do**
    if $head$ = $body$ **then**
      $c \leftarrow$ FINDCHILDCONSTNODES($body$, $head$)
      $i \leftarrow$ FINDIMPLICATIONNODES$(head)$
      $parent \leftarrow$ FINDPARENT$(body)$
      $childs \leftarrow$ FINDCHILD$(head)$
      for $child$ in $childs$ **do**
        if $child$ not in $constans$ **then**
          $to \leftarrow to \cup$ GETID$(child)$
        end if
      end for
      $edges \leftarrow edges \cup \{($GETID$(parent), to)\}$
      $nodes \leftarrow nodes \cup \{c, i\}$
    end if
  end for
end for
$edges \leftarrow$ FINDEDGESCONNECTED$(edges)$
ADDEDGES$(edges)$
REMOVENODES$(nodes)$
return

identifies child constant nodes and implication nodes, which are removed at the end of the process. Additionally, it finds parents for each bodies and childs for each heads, then connect them. To prevent consecutive HB predicates in GDL rules, the algorithm utilizes the function FINDEDGESCONNECTED to concatenate their edges end-to-end.

3.2.6 Reversed Instantiated Rule Graph (R-IRG)

The Annotated Rule Graph mentioned in the above section is capable of providing a graph-based game rule representation for different games, but it cannot describe

the current state of the game. GNN-Reasoner [9] proposed Instantiated Rule Graph (IRG), which aims to instantiating rule graph with game current state and player moves. However, it only serves to predict legal actions and next states. We proposed a new type of Instantiated Rule Graph (IRG), termed as Reversed instantiated Rule Graph (R-IRG). We additionally added supports for the terminal predication, goal predication and negation problem on its basis. Most message passing mathematics in Graph Neural Networks (GNNs) propagate and aggregate information in fixed directions within directed graphs [43]. Therefore, we reversed all edges at the end to allow nodes requiring prediction to aggregate more information.

Definition 3 (Labelling functions for R-IRG) *Given game G, its Annotated Rule Graph $R_A = (V, E, T)$, a state $S = \{f_1, f_2, \ldots, f_n\}$ consisting of fluents f_i, actions $A = \{(p_1, a_1), \ldots, (p_n, a_n)\}$, and nodes N are node type $T(n_p)$ = not, an Reversed Instantiated Rule graph of state S and actions A is a graph $I = (V, E^T, T, L_S, L_A, L_N)$ where $L_S : V \to \{false, true\}$, $L_A : V \to \{false, true\}$ and $L_N : V \to \{false, true\}$ are three labelling function based on state S, action A and negation nodes N.*

If node n is node type $T(n)$ = expression and $FG(n) \in S$ or $FG(n) \in A$ and parent node n_p is node type $T(n_p)$ = true or $T(n_p)$ = does with edge $(n_p, n) \in E$, then $L_S(n)$ = true, $L_S(n_p)$ = true, $L_S(n_c)$ = true, $L_A(n)$ = true, $L_A(n_p)$ = true, $L_A(n_c)$ = true for all children nodes given $(n, n_c) \in E$. For all other nodes $n_o \neq n$, $L_A(n_o)$ = false, $L_S(n_o)$ = false.

If node n is node type $T(n)$ = expression and $FG(n) \in S$ or $FG(n) \in A$ and parent node n_p is node type $T(n_p)$ = not with edge $(n_p, n) \in E$, then $L_N(n)$ = true, $L_N(n_p)$ = true, $L_N(n_c)$ = true for all children nodes given $(n, n_c) \in E$. For all other nodes $n_o \neq n$, $L_N(n_o)$ = false.

At the end, we reverse all edges, i.e., for all $(v, v') \in E$ we have $(v', v) \in E^T$.

As defined in Definition 3, the labeling function labels nodes associated with the

current state, actions and nodes relate to negation problem. Furthermore, it also reversed all edges from Annotated Rule Graph. Figure 18 illustrates the Reversed Instantiated Rule Graph after Graph-Based Rule Flattening for the GDL rule `(<= terminal not open)`, `(<= open (true (cell 1 1 b)))`. Both the `true` node and its related expression nodes are labeled as true (blue). The `not` node labels itself and its child nodes below as true (yellow). The nodes shaded in yellow-blue gradient represent those nodes simultaneously affected by the labeling functions from both `not` and the current state.

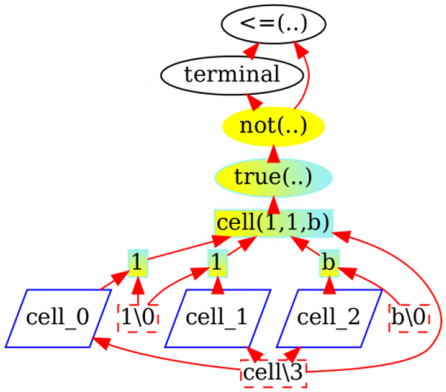

Figure 18: Example of Colored Reversed Instantiated Rule Graph

3.2.7 Node Embedding and Feature Extraction

The Reversed Instantiated Rule Graph (R-IRG) can effectively represent most games based on the Game Description Language (GDL) and current states, yet it is a form of lexical mapping.

The vector embedding method proposed by the GNN-Reasoner [9] has demonstrated effectiveness in predicting legal and next actions. It constructs a feature vector based on different keywords and characteristics in the game rules. However, it often encounters challenges when dealing with more complex predictions involving terminal and goal states. To address this limitation, we have extended the node vector embedding method

from the GNN-Reasoner, enriching our dataset with additional features. The key criteria for feature extraction is based on GDL kitself.

For each node $n \in V$ in the Reversed Instantiated Rule Graph (R-IRG) $I = (V, E, T, L_S, L_A, L_N)$, we generate a vector embedding $\vec{v}_n \in \{0,1\}^{22}$. As outlined in Table 1, the first index of the vector are derived from R-IRG subtree labeling function, which we discussed in the sction 3.2.6. The second index is dedicated solely to the keyword `goal`, indicating the current player score. The third index `not` is also derived from R-IRG subtree labeling. The value of the not node is determined by the length of $\vec{v}_n$ plus the number of dependencies connected to the not node. The eighth index indicates the number of dependencies of a expression node. For example, expression node `cell(1,1,b)` has three dependencies (childs). Its value is equal to the length of the $\vec{v}_n$ plus the number of constants. The remaining indexes represent different nodes and keywords within the Annotated Rule Graph. When true, they are also marked with the number `22`, reflecting the total length of the vector.

(a) Label and non-keywords

Index	Embedding
1	true in state
2	goal score
3	negation
4	symbol constant
5	symbol argument
6	expression
7	constant
8	dependencies

(b) Keywords

Index	Embedding
9	<=
10	not
11	or
12	distinct
13	does
14	goal
15	init

(b) Keywords

Index	Embedding
16	legal
17	next
18	role
19	terminal
20	true
21	input
22	base

Table 1: Vector embedding values

Figure 19 illustrates an example of vector embedding. Similar to the previous sections, it is generated by GDL rules: `(<= terminal not open), (<= open (true (cell 1 1 b))), ..., (<= open (true (cell 3 3 b)))`. The first node is 'terminal', resulting in $\vec{v}^n_{20} = 22$. The second node is the keyword

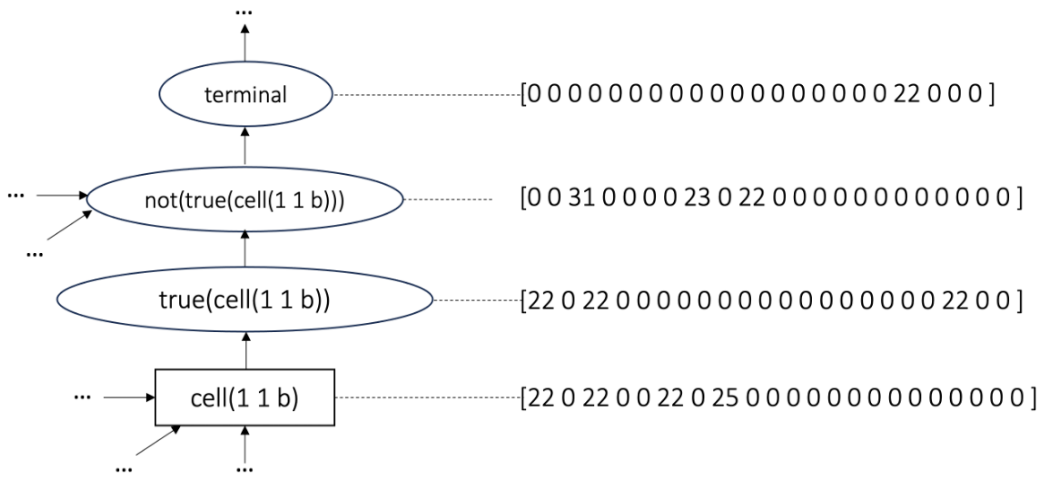

Figure 19: Example of Vector embedding

'not' node, which has 9 childs, hence $\vec{v}_3^n = 31$, $\vec{v}_8^n = 31$ and $\vec{v}_{10}^n = 31$. The third node is the keyword 'true' node, with $\vec{v}_0^n = 22$, $\vec{v}_2^n = 22$, and $\vec{v}_{20}^n = 22$. The last node is the expression node, with $\vec{v}_0^n = 22$, $\vec{v}_2^n = 22$, and $\vec{v}_6^n = 22$. Since it has 3 constants: `1, 1, b`, therefore $\vec{v}_8^n = 25$.

3.2.8 Target Labelling And Node Masking

In Graph Neural Networks (GNNs), the target matrix is a one-hot encoded matrix used for supervised learning, containing the true output values corresponding to the training dataset. Each row represents a data point, and each column represents a category, with the column value for the true category set to 1 and all other columns set to 0. This matrix is crucial as it is employed for error computation, back-propagation, and evaluation of the neural network. During the node embedding process, the predicted target matrix is labeled based on the current state of the game using Prolog.

Furthermore, node masking is applied to specify which data should be considered or ignored by the neural network during the training process. These include the Legal mask, Next mask, Terminal mask, and Goal mask. For example, if `(<= goal opalyer 50)` is true in the game's current state, nodes associated with this condition are marked

Algorithm 3 Labelling taget and masks

function LabellingGoalTargetAndMask(AnnotatedRuleGraph R_A, State S)
$\vec{g} \leftarrow \varnothing$, $\vec{g}_m \leftarrow \varnothing$
$V, E, T \leftarrow R_A$
$K \leftarrow$ GETKEYWORDNODES(V)
for k in K **do**
  $k_{id} \leftarrow$ GETID(k)
  if $T(k)$ = $goal$ **then**
    $rule \leftarrow$ GETRULE(k) $\triangleright$ {Find all childs and return a complete rule}
    if CHECKRULEBODY$(S, rule)$ **then**
      $\vec{g}(k_{id})$ = 1 $\triangleright$ {Check rule authenticity through Prolog}
    end if
    $\vec{g}_m(k_{id})$ = 1
  end if
end for
return $\vec{g}, \vec{g}_m$

with 1 in the Goal mask matrix, and Target matrix. Algorithm 3 presents pseudocode illustrating an example of labeling goal ground truth target and mask matrix.

3.2.9 Dataset Generation

The algorithm 4 shows the overall process for producing a dataset of vector embeddings along with associated target and mask labels from a given rule flattened game description G. Initially, it generates a set of samples S_v from G, based on a pre-defined number of examples d. Next, it grounds all Game Description Language (GDL) rules and convert them into an Annotated Rule Graph R_A. Next, we iterate through all sample S_v and utilize them to construct Reversed Instantiated Rule Graphs (R-IRGs) I and I_a, with and without a specified action a. This is due to the difference between predicting the next state and other prediction tasks, where predicting the next state necessitates knowledge of the current player's move. These Reversed Instantiated Rule Graphs (R-IRGs) are then pass through vector embeddings to obtain feature matrices X and X_a. The algorithm acquires an adjacency matrix A from the Annotated Rule Graph R_A and generates training target and mask matrices for legal, next, terminal, and goal nodes.

Algorithm 4 Dataset generation algorithm

function GenerateDataset(Rule Flattened Game description G, number of examples d)
$S_v \leftarrow \text{GENERATESAMPLES}(G, d)$
$R_G \leftarrow \text{RULEGROUNDING}(G)$
$HB \leftarrow \text{HBFINDING}(G)$
$R_A \leftarrow (V, E, T) = \text{TRANSFORMTOANNOTATEDRULEGRAPH}(R_G, HB)$
for all $(S, a, S_{\text{legal}}, S_{\text{next}}, S_{\text{terminal}}, S_{\text{goal}}) \in S_v$ **do**
  $E \leftarrow (S, a, S_{\text{legal}}, S_{\text{next}}, S_{\text{terminal}}, S_{\text{goal}})$
  $I \leftarrow (V, E, T, L) = \text{REVERSEDINSTANTIATERULEGRAPH}(R_A, S, \varnothing)$
  $I_a \leftarrow (V, E, T, L_a) = \text{REVERSEDINSTANTIATERULEGRAPH}(R_A, S, a)$
  $X \leftarrow \text{GETVECTOREMBEDDING}(I)$
  $X_a \leftarrow \text{GETVECTOREMBEDDING}(I_a)$
  $A \leftarrow \text{GETADJACENCYMATRIX}(R_A)$
  $\vec{l}, \vec{l_m} \leftarrow \text{LABELLEGALTARGETSANDMASK}(R_A, S)$
  $\vec{n}, \vec{n_m} \leftarrow \text{LABELNEXTTARGETSANDMASK}(R_A, S)$
  $\vec{t}, \vec{t_m} \leftarrow \text{LABELTERMINALTARGETSANDMASK}(R_A, S)$
  $\vec{g}, \vec{g_m} \leftarrow \text{LABELGOALTARGETSANDMASK}(R_A, S)$
  $D \leftarrow D \cup \{E, ((X, A), \vec{l}, \vec{t}, \vec{g}), ((X_a, A), \vec{n}), \vec{l_m}, \vec{n_m}, \vec{t_m}, \vec{g_m}\}$
end for
return D

Finally, these feature, adjacency, target, and mask matrices are all incorporated into the dataset D. The completed dataset is then outputted for use in training or evaluating the neural networks.

3.3 Dataset Generation Process Enhancement

While GNN-Reasoner[9] provided the foundational code for dataset generation and some analysis tools, its utility was hindered by slow data processing speeds, an unclear data structure. Consequently, we undertook a comprehensive rewriting of the code, significantly increasing the speed of dataset generation by at least eight times. This enhancement was primarily accomplished through the following strategies:

1. **Parallel Processing:** By leveraging the multi-core, multi-threading capabilities of CPUs, we enabled the parallel generation of the dataset. This approach

significantly reduced the time required for dataset compilation.

2. **Code Optimization:** We optimized code, such as for loops, to reduce computational complexity and execution time. Additionally, we redesigned a series of new data structures for storing various information of R-IRG. This facilitated our rapid experimentation with more graph structures.

3. **Modularization and Management:** We divided the dataset generation code into distinct, clear modules. This reorganization not only made the code more comprehensible but also allowed for more efficient debugging and maintenance. Furthermore, we employed a shell script to unify and manage these modules, ensuring a cohesive and systematic approach to dataset generation.

Game	GNN-Reasoner-v2	GNN-Reasoner
kightstrour	5.51	46.27
hamilton	6.69	94.36
hanoi6disks	7.83	88.33
tictactoe	3.75	36.00
tictactoelarge	5.43	91.42
blocker	2.53	143.41

Table 2: Dataset generation time (min)

Table 2 presents a comparative analysis for dataset generation speeds between the GNN-Reasoner-v2 and GNN-Reasoner [9]. All generations are conducted on an Intel Xeon Silver 4210 CPU with 96 GB of memory which runs Ubuntu 20.04.2 LTS. It shows that the original dataset creation method used by GNN-Reasoner for more complex games takes longer. Our new method is significantly faster in comparison. The increased speed of dataset generation not only greatly improves the efficiency of our experimental processes but also enables us to explore more graph structures by saving a large amount of time.

3.4 Machine Learning Framework

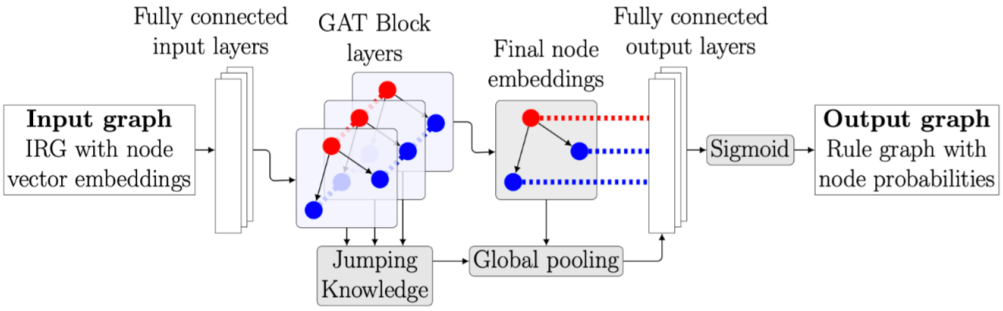

Figure 20: Graph neural networks architecture, Retrieved from GNN-Reasoner [9]

We proposed a new neural network, GNN-Reasoner-v2, which is based on the GNN-Reasoner [9] but with modifications to its GAT blocks. Since the data is based on graphs (with numerous nodes and edges), we continue to use graph neural networks. Hyper-parameters are chosen by multiple experiments. The overall structure of the neural network remains unchanged, but the GAT blocks have been altered. This new neural network architecture significantly reduces the neural network's demand for computational resources without compromising performance, enabling it to train on a wider range of more complex games.

The datasets generated are initially passed through a fully-connected layer to standardize the dimensions of all IRG samples. Subsequently, these samples undergo message passing within a GAT Block, resulting in updated node embeddings. Each layer of the GAT Block incorporates Jumping Knowledge [41], which is then integrated with the node embeddings from the final layer.

For Global Pooling [14], the final node embeddings are utilized to obtain a graph-level embedding. This embedding is then fed into a fully connected output layer. Finally, a sigmoid function is applied to derive probabilities for each node in the Reversed Instantiated Rule Graph.

An example of goal value prediction pseudocode for the aforementioned process

Algorithm 5 Neural Reasoner Prediction

function InferGoalStates(Rule Flattened Game Description G, state S, Goal-Threshold t_{goal})
$goal \leftarrow \varnothing$
$R_A \leftarrow$ ANNOTATEDRULEGRAPH(G)
$I \leftarrow (V, E^T, T, L_S, L_A, L_N)$ = REVERSEDINSTANTIATERULEDGRAPH(R_A, S)
$K \leftarrow$ GETKEYWORDNODES(V)
$X \leftarrow$ GETVECTOREMBEDDING(I)
$A \leftarrow$ GETADJCENCYMATRIX(I)
$X' \leftarrow$ GNN(X, A) {X' is the output with node probabilities}
for all $k \in K$ **do**
  $k_{id} \leftarrow getID(k)$
  if $T(k) = goal$ **and** $X'(k_{id}) > t_{goal}$ **then**
    $player \leftarrow$ GETPLAYER(k)
    $score \leftarrow$ GETSCORE(k)
    $goal \leftarrow goal \cup \{player : score\}$
  end if
end for
return $goal$

is provided in Algorithm 5. It predicts current player score by transforming a game description G and current state S to a Reversed Instantiated Rule Graph (R-IRG) I and imploying a Graph Neural Network GNN. Iterating through keyword nodes K, it checks if they correspond to the goal node and have a probability exceeding the goal threshold. Finally, it returns the predicted goal values corresponding to specific player. In all four prediction tasks, only the prediction method for Next fluents differs slightly from the aforementioned approach. This is because it requires the current Reversed Instantiated Rule Graphs (R-IRGs) to include not only the current state but also the move made by the player, whereas the other three prediction tasks only require the current game state.

We utilized a different GAT (Graph Attention Network) Block compared to GNN-Reasoner [9]. We employed a one-layer GAT Block without residual connection, aiming to balance high performance with reduced hardware consumption. As illustrated in Figure 21, `X` represents the node feature matrix, and `A` refers to the adjacency matrix.

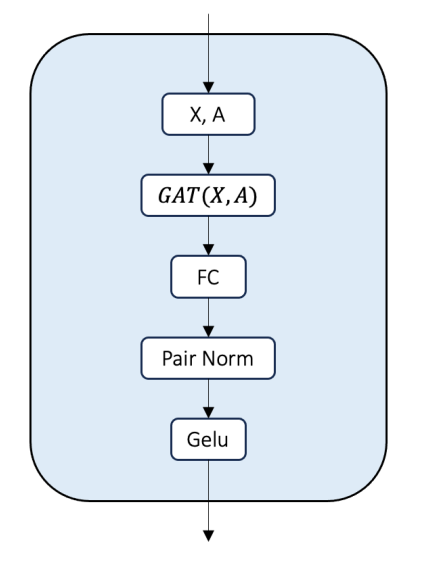

Figure 21: Inside of GAT Block

We still chose to use the GATConv neural network [37] due to its lower memory consumption and higher performance. Subsequently, the output passes through a PairNorm layer [42] and then enters a GeLU activation function [11].

3.4.1 Training Details

Dealing with the disproportionately small representation of Terminal and Goal nodes in the Reversed Instantiated Rule Graph presented a challenge, particularly given the significant class imbalance in a binary classification context within the dataset. To address this issue, we employed Focal Loss [15], which effectively improves neural network convergence by prioritizing the learning of more complex instances. Additionally, we incorporated Early Stopping to optimize the number of training epochs, aiming to prevent over-fitting by halting training when the training loss ceases to decrease over a predefined number of epochs. We also utilized Gradient Clipping [22] techniques to mitigate the risk of gradient explosion. Following the methodology from GNN-Reasoner [9], we employed the AdamW optimizer [16], renowned for its adaptive learning rate adjustment and effective weight decay mechanisms. These features are instrumental in

enhancing neural network convergence and mitigating the risks of over-fitting.

It's important to note that our dataset generation process generated two R-IRGs: I and I_a. R-IRG I contains the information of current game states. R-IRG I_a contains the information of current game states and player moves. Since next prediction relies on extra information about player actions (I_a), whereas the other three prediction tasks do not (I). the neural network training process involves separate forward propagation for next prediction and the combined other three prediction tasks. Subsequently, backward propagation is conducted jointly for all tasks.

Parameter	Value
Attention heads	4
Batch size	$16 - 32$
Dropout	10^{-1}
Early patience	5
FC size	64
GAT layers	$2 - 5$
Learning rate	10^{-4}
Weight decay	10^{-3}

Table 3: Trainning hyper-parameters

Table 3 provides a comprehensive overview of the hyper-parameters used in the training process. `Attention Heads` refers to the number of attention heads in the multi-head attention mechanism of GAT (Graph Attention Network) layers. `Batch Size` indicates the number of samples processed by the training process simultaneously. It's important to note that due to hardware limitation, different games require the selection of different batch sizes to ensure smooth progression of neural network training. The batch sizes we've chosen are based on extensive experimentation, ensuring a balance between neural network performance and consumption of computer hardware resources. `Dropout` denotes the proportion of data randomly omitted in each layer of the GAT during training. `Early Patience` represents the number of epochs without performance improvement before training is stopped. `FC Size` determines

the dimensionality of the output from the fully connected layer. `GAT Layers`, also can be seen as hidden layers in our neural network, indicate the number of GAT blocks employed. Lastly, `Weight Decay` is a parameter in the AdamW optimizer, crucial for preventing the neural network training doesn't lead to over-fitting.

3.4.2 Threshold Finding

In Figure 20, the final output of the neural network during the prediction process is the probability distribution of all nodes in the reversed-Instantiated rule graph (R-IRGs). We applied a modified threshold finding method derived from GNN-Reasoner [9], capable of determining thresholds for the neural network's output probability distribution.

Algorithm 6 Find Thresholds

function FindThresholds(validation_dataset D_v))
for i in range(0, 1, 0.05) **do**
{Get mask and target matrix}
for d in D_v **do**
$\{((X, A), \vec{l}, \vec{t}, \vec{g}), ((X_a, A), \vec{n}), \vec{l_m}, \vec{n_m}, \vec{t_m}, \vec{g_m}\} \leftarrow d$
$x \leftarrow \text{GNN}(X, A)$
$x' \leftarrow \text{GNN}(X_a, A)$
$t_{out} \leftarrow \text{WHERE}(\text{SIGMOID}(x) > i, 1, 0)$
$g_{out} \leftarrow \text{WHERE}(\text{SIGMOID}(x) > i, 1, 0)$
$l_{out} \leftarrow \text{WHERE}(\text{SIGMOID}(x) > i, 1, 0)$
$n_{out} \leftarrow \text{WHERE}(\text{SIGMOID}(x') > i, 1, 0)$
$e_t \leftarrow \text{MSELOSS}(\vec{t}_m * t_{out}, \vec{t}_m * \vec{t}, SUM)/\text{NONEZERO}(\vec{t}_m)$
$e_g \leftarrow \text{MSELOSS}(\vec{g}_m * g_{out}, \vec{g}_m * \vec{g}, SUM)/\text{NONEZERO}(\vec{g}_m)$
$e_l \leftarrow \text{MSELOSS}(\vec{l}_m * l_{out}, \vec{l}_m * \vec{l}, SUM)/\text{NONEZERO}(\vec{l}_m)$
$e_n \leftarrow \text{MSELOSS}(\vec{n}_m * n_{out}, \vec{n}_m * \vec{n}, SUM)/\text{NONEZERO}(\vec{n}_m)$
$t_t, t_g, t_l, t_n \leftarrow \text{UPDATEBESTTHRESHOLD}(i, e_t, e_g, e_l, e_n)$
end for
end for
return t_t, t_g, t_l, t_n

Algorithm 6 illustrates an example of the process for determining terminal thresholds. It aims to find a set of thresholds to minimize certain loss functions on a given validation dataset D_v. It iterates over a range of possible threshold values from 0 to 1 with a

step size of 0.05, while also iterating over each sample in the validation dataset. For each sample, it extracts various tuples representing graph features matrices, adjacency matrices labels, masks, and targets. It then pass a Graph Neural Network (GNN) to get a possibility distribution based on both the original Reversed Instantiation graph (R-IRG) and graph with R-IRG action labelling function, converting these predictions into binary outputs using a threshold determined by the current iteration. It subsequently computes the mean square error loss for each predictable node, taking into account only the valid labels using masks, and adjusts the optimal thresholds accordingly. Finally, it returns the discovered optimal thresholds t_t, t_g, t_l, and t_n.

Due to limited research time, we had to rely on a quick method for searching thresholds. If we could further narrow down the search range, it would undoubtedly enhance the final performance and accuracy of the GNN-Reasoner-v2, albeit at the cost of significantly increasing training time.

3.5 Machine Learning Engineering

Neural network research, due to its unique demands, often requires researchers to spend extensive time training a variety of neural networks and generating diverse datasets to pursue different research objectives. This highlights the crucial need for a set of tools designed to streamline and standardize these testing and research processes. Moreover, a modular programming framework is essential for researchers to experiment with more crazy ideas. After reviewing the initial code from GNN-Reasoner [9], we decided to completely redesign the structure of our research project and tools. We not only extensively rewrote the code to improve the efficiency of the training processing, but also added more analysis tools. This chapter offers a brief introduction to these tools. It should be noted that although GNN-Reasoner had already provided some of these tools, due to its complex operation, they have been reprogrammed and new functionalities

have been added. These implementations are written in Python, utilizing PySwip [32] and the PyTorch Geometric library [4].

The `End to End Start Up Tool` enables researchers to train neural networks with a single click that learn directly from raw input (GDL) to desired output, and conduct evaluation. It saves various data throughout the process, including dataset data (graph matrix), training data (loss, weights, and biases), and prediction accuracy, PCA analysis, etc. It is a collection of tools that we will describe in subsequent sections of this chapter.

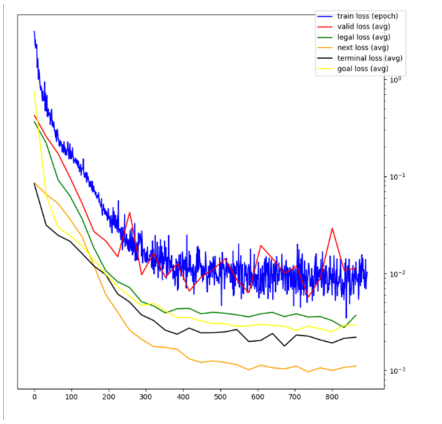

Figure 22: An example of loss curve viewer

Our `Loss Curve Viewer` provides a dynamic loss curve in real time with just one command during training. This tool comes with features for automatic saving and historical data review. It also enables the segregation and examination of individual loss values for each predictable node. Furthermore, this tool can dynamically displays the historical training loss curve based on the current training neural network, facilitating comparative analysis for researchers.

`Dataset Generation Tool` enables researchers to automatically generate large number of different game datasets, through pre-set parameters. This tool enables researchers to make the most of their resting hours without the need to write or modify project code.

The `Neural network training Tool` allows researchers to automatically train a multitude of different neural networks by setting parameters in advance. It not only auto-saves the currently trained neural network but also keeps a record of the thresholds for predictable nodes and all the training parameters.

The `Neural network Evaluation Tool` empowers researchers to automatically evaluate a large number of neural networks by pre-setting parameters. Additionally, these test results are automatically stored locally on the computer for convenient access by researchers.

The `Gradient Histogram Viewer` is developed based on TensorBoard [1], aimed at visualizing the distribution of gradient values for each parameter in neural networks. Due to the frequent occurrences of gradient vanishing and exploding in datasets with long-range dependency issues, this tool assists researchers in quickly identifying such occurrences and taking measures to address them.

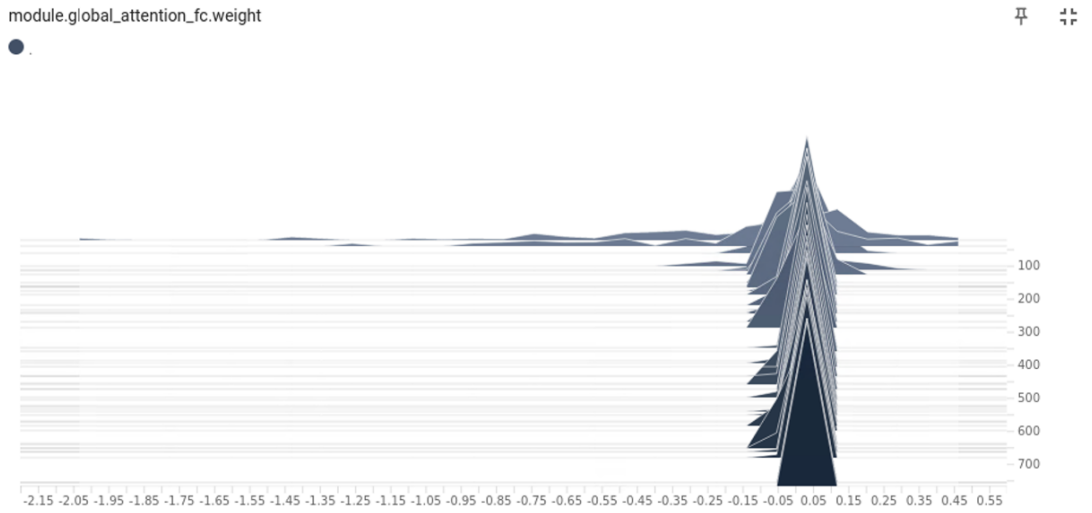

Figure 23: Gradient histogram viewer

The `Dataset Viewer` allows developers to visualize the Annotated Rule Graph, which is instrumental in analyzing the dataset. The Reversed Instantiated Rule Graph Analysis tool not only visualizes the R-IRG but also provides detailed R-IRG data, such as the number of nodes, edges, mean path lengths and more. The key difference

between this tool and the Annotated Rule Graph Viewer is in their efficiency and use cases. The Annotated Rule Graph Viewer runs with high efficiency and provides only graph visualization. On the other hand, while the Instantiated Rule Graph Analysis tool runs slower, it offers more detailed insights into the graph structure, making it better suited for in-depth data analysis.

```
Error: Terminal incorrect
predict: False, inferred: True
current state:
xxx
  o
oox
```

Figure 24: An example of Error Log tool

The `Error Log tool`, while being a part of the Neural network Evaluation Tool, can also function independently. It automatically records errors encountered during the neural network evaluation process and provides the correct outcomes for researchers to analyze the cause of the issues. For instance, if the neural network incorrectly predicts the Terminal, the tool saves the neural network's predicted outcome, a visualization of the current game state, and the correct result for comparison.

3.6 Conclusion

In this chapter, we provide a comprehensive overview of the data generation methods, neural network architectures, and analysis tools. In the following chapters, we will conduct a series of evaluations on the aforementioned methods.

Chapter 4

Analysis

4.1 Introduction

To demonstrate the effectiveness of the GNN-Reasoner-v2 inference, we conducted various training and testing approaches, including: training based on individual games, transferred and mixed training, principal component analysis (PCA), training based on different graph structures, and training based on different neural network architectures.

Individual game training encompassed six distinct games, as shown in Table 4, which presents their corresponding Reversed Instantiated Rule Graph (R-IRG) matrices, providing insights into their complexity and characteristics. The criteria for our selection of these games is their R-IGR complexity in Table 4. The higher the complexity of the game, the more hardware resources are required for training. The complexity of different games varies widely, with the number of nodes ranging from a minimum of 6,332 to a maximum of 19,672, and the number of edges ranges 9,769 to 31,516. The correctness and completeness of these game datasets are validated by their trained neural network. These game datesets are used to train neural networks in Chapter 4.2. Transfer learning experiments are conducted to evaluate the neural network's ability to transfer its learned knowledge to new games that it has not been trained on. Results

are shown is Chapter 4.4. Mixed training involved randomly combining datasets from multiple games for training, evaluating the neural network's generalizability and its ability to transfer learned knowledge to different games. Section 4.5 presents its results. Training based on different graph structures in Section 4.7 aimed to evaluate the impact of varying graph structures on neural network performance. we also explored various neural network architectures.

Consequently, we conducted a neural network ablation study to investigate the influence of different neural network structures on neural network accuracy.

Game	R-IRG size		Outdegree	
	Nodes	Edges	Max	Mean
kightstrour	12291	30154	1393	4.91
hamilton	14963	31516	1601	4.21
tictactoe	6332	14572	759	4.60
blocker	3870	9769	547	5.04
blocks2player	12505	26490	1577	4.24
connecfour	19672	47221	2249	4.80

Table 4: Graph metrics of six game datasets

The evaluation method used by GNN-Reasoner [9] involves simulating game playing, which can result in a non-trivial overlap between the test dataset and the training dataset. To address this issue, we made modifications. For each game, we generated 12,000 Instantiated Rule Graphs (IRG), with 85% used for training, 5% for validation, and 10% for testing.

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 with 44 GB of memory.

4.2 Evaluation on Individual Games

Firstly, we present the evaluation results of GNN-Reasoner-V2 for individual games measured by F-1 score. The neural network was separately trained on six games for predicting terminal states, goal scores, legal actions, and next fluents. The dataset for these games underwent manual Rule Flattening and Graph-based Rule Flattening. Among these games, Knight's Tour and Hamilton are special that their GDL do not contain a large number of HB predicates, and thus, the manual Rule Flattening was not applied.

The F1-score is used to evaluate the performance of a classification model, which combines the model's precision and recall. Precision measures the proportion of true positive predictions among all samples predicted as positive. It is calculated as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{19}$$

Recall represents the proportion of true positive predictions among all actual positive samples. It is computed as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{20}$$

The F-1 score was measured by the harmonic mean of precision and recall, given by:

$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{21}$$

The prediction of GNN-Reasoner-v2 involves both node classification and binary classification. The neural network first classifies the types of nodes (legal, next, terminal and goal, etc) before making True or False predictions. For instance, during the next prediction, it first classifies all nodes related to next state among all nodes, then label

each next node as true or false. Finally, it consider all nodes labeled as True as the predicted result for the final output.

We categorized the output results of the neural network's four prediction tasks into three types: correct nodes, incorrect nodes, and missing nodes. Correct nodes are equivalent to true positives in F1-score calculation, while incorrect nodes correspond to false positives. The missing nodes indicates the nodes that the neural network's output fails to identify compared to the actual correct results, which is also relevant to false negatives.

It is noted that the terminal prediction is different with other three prediction tasks as it needs to consider both terminal and non-terminal states. However, legal and next prediction only need to consider the non-terminal state while goal prediction needs to consider terminal state.

Game	Terminal			Goal			Legal			Next		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
tictactoe	1.00	0.96	**0.99**	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
knightstour	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
hamilton	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
blocker	1.00	1.00	**1.00**	0.97	1.00	**0.98**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
blocks2player	1.00	1.00	**1.00**	0.98	0.98	**0.98**	0.98	0.99	**0.98**	1.00	0.95	**0.97**
connectfour	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 5: GNN-Reasoner-v2 accuracy on individual games

Table 5 shows the accuracy across six games in all four predication tasks. The bold F1-Scores indicate high performance, with a score higher than 0.95 representing very high precision and recall. All games achieve high accuracy across the four different prediction tasks. Due to the random shuffling of the training dataset in each training process of the neural network, it's understandable that the training results may fluctuate to some extent.

4.3 Rule Flattening

In order to investigate the impact of long-range dependencies and the imbalance of nodes requiring prediction on neural network performance, we conducted an experiment to train several neural networks under Annotated Rule Graph (same with section 4.2) and non-flattened Rule Graph proposed by GNN-Reasoner [9]. Notably, the neural network trained on non-flattened Rule Graph only employs the Rule Graph construction method from the GNN-Reasoner, while other configurations such as vector embedding methods remain the same as described in chapter 3. Figure 25 illustrates the difference between the two graph structures mentioned above. The non-flattened Rule Graph exhibits a higher path length compared to the Annotated Rule Graph.

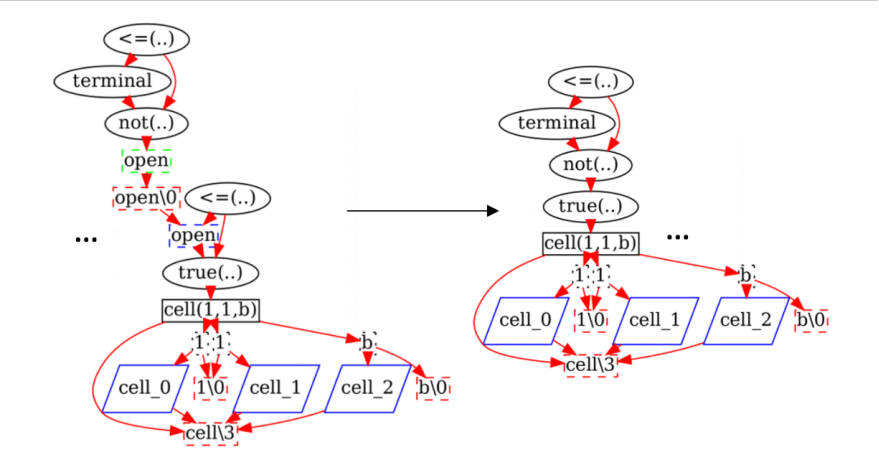

Figure 25: The difference of non-flattened Rule Graph [9] (left) and Annotated Rule Graph (right) in Tic-Tac-Toe

Table 6 shows the GNN-Reasoner-v2 accuracy of three games: Tic-Tac-Toe, Connect-Four, and Blocker under two graphs mentioned above. It shows all three games without Rule Flattening exhibit very poor performance especially, in terminal and goal prediction. This result can be analyzed using the following data, including: mean path length, the distribution of node types, and the training loss curve.

Table 7 presents the mean path length of each predictable keyword node. Given a

Game	Description	Terminal			Goal			Legal			Next		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
tictactoe	Annotated	1.00	0.96	**0.98**	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	non-flat	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	**1.00**	1.00	1.00	**1.00**
connectfour	Annotated	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	non-flat	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.13	0.23	1.00	0.99	**0.99**
blocker	Annotated	1.00	1.00	**1.00**	**0.97**	1.00	0.98	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	non-flat	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 6: GNN-Reasoner-v2 accuracy between Annotated Rule Graph and non-flattened Rule Graph proposed by GNN-Reasoner on tictactoe

Game	Description	Terminal	Goal	Legal	Next
tictactoe	Annotated	2.15	2.54	1.68	1.66
	non-flat	**6.24**	**6.56**	1.68	1.66
connectfour	Annotated	1.55	1.72	1.60	1.62
	non-flat	**7.28**	**7.73**	**3.93**	**3.69**
blocker	Annotated	2.06	2.51	1.50	1.87
	non-flat	**4.64**	**5.11**	1.50	1.87

Table 7: Graph comparison of Annotated Rule Graph and non-flat Rule Graph proposed by GNN-Reasoner [9] (mean path length)

directed graph, we computed the average path length from each keyword node requiring prediction to the lowest child predicate node. The data indicates that the mean path length of the non-flattened Rule Graph generally exceeds that of the Annotated Rule Graph, especially with respect to terminal and goal nodes.

For instance, in non-flatted Rule Graph, the terminal mean path length in Tic-Tac-Toe is three times higher than the Annotated Rule Graph, whereas in larger games like Connect-Four, this difference approaches five times. Furthermore, the Rule Graph without Rule Flattening exhibit an obvious imbalance in mean path length, with the mean path length of terminal and goal nodes being at least double that of legal and next nodes. These issues may also impact the neural network's accuracy.

The performance of the neural networks mentioned above can also be analysed through the number of predictable keywords. Table 8 shows the number of nodes for four keywords (terminal, goal, legal and next) in each game. The number of nodes for both Tic-Tac-Toe, Connect-Four and Blocker significantly increases after Rule

Game	Description	Terminal	Goal	Legal	Next
tictactoe	Annotated	**17**	**34**	20	242
	non-flat	3	6	20	242
connectfour	Annotated	**139**	**280**	**156**	**786**
	non-flat	3	8	86	716
blocker	Annotated	**69**	**138**	32	80
	non-flat	2	4	32	80

Table 8: Graph comparison of Annotated Rule Graph and non-flattened Rule Graph (nodes)

flattening, leading to a noticeable improvement in accuracy. Furthermore, there is a severe imbalance in the distribution of predictable nodes in the dataset. This explains why terminal state and goal scores are often more challenging to predict than legal actions and next fluents.

The figure 26 illustrates the comparison of training loss curves between the Annotated Rule Graph and the non-flattened Rule Graph [9] for Tic-Tac-Toe. The Annotated Rule Graph exhibits lower loss values compared to the non-flattened Rule Graph, indicating that the neural network can better learn from this graph structure. The loss value of terminal and goal is lower than that of legal and next from the beginning. Furthermore, during the training of non-flattened Rule Graph, the loss values for terminals and goals cease to decrease early in training.

In conclusion, based on the above results, we observed that neural networks are sensitive to both long mean path length caused by long-range dependencies and significant imbalances in mean path length across different prediction tasks, as well as significant imbalances in the number of nodes across different prediction tasks. Addressing the balance between computational resource consumption and handling deeper graph structures has long been a significant challenge in the field of graph neural network research.

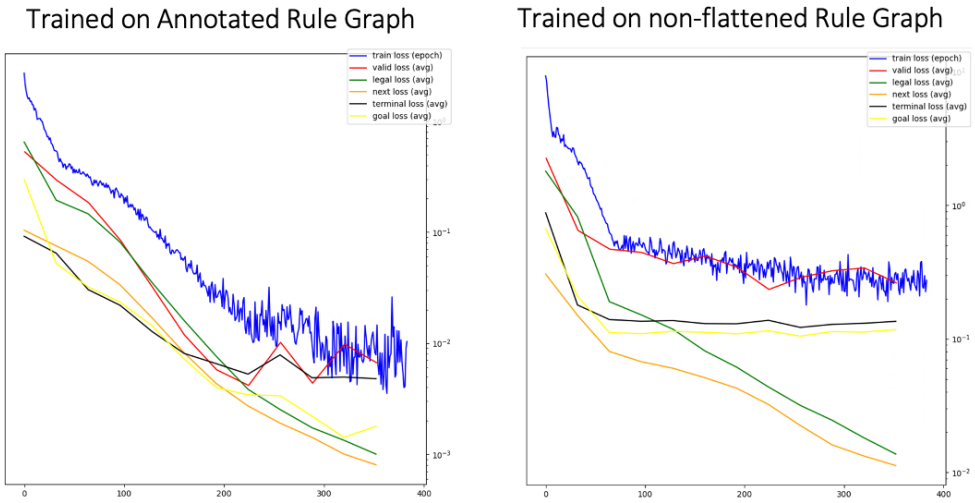

Figure 26: Training loss curves for Annotated Rule Graph (left) and non-flattened Rule Graph (right) on Tic-Tac-Toe

4.4 Transfer Learning

In addition to the individual testing of games mentioned earlier, the transfer learning capability of the GNN-Reasoner-v2 is also crucial in our research. The GNN-Reasoner-v2 needs to possess good transfer learning ability to adapt across different domains. We conducted transfer learning experiments on four games: Hamilton, Knight's Tour, Tic-Tac-Toe, and Connect-Four, respectively, to evaluate its transfer learning capability.

Table 9 shows the accuracy of each type of trained networks for games it was not trained on. It indicates that the neural networks generally perform exceptionally well when evaluated on the game they were trained on. However, there are some variations when neural networks are applied to other games. The neural network trained on Hamilton shows very high scores for all states when tested on its self, but has a lower performance on the terminal state of Knight's Tour, and varying performance on Tic-Tac-Toe and Connect-Four. The neural network trained on knight's Tour excels on its game and shows good transferability to other games, especially on legal actions and next fluents. The neural networks trained on Tic-Tac-Toe and Connect-Four also show

Game	Description	hamilton			knightstour			tictactoe			connectfour		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
hamilton	Terminal	1.00	1.00	**1.00**	0.01	1.00	0.18	0.16	0.88	0.27	0.94	0.59	0.73
	Goal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	0.44	0.08	0.14	0.00	0.00	0.00
	Legal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	0.79	0.98	0.88	0.42	0.70	0.53
	Next	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
knightstour	Terminal	1.00	0.78	0.88	1.00	1.00	**1.00**	1.00	0.96	**0.98**	1.00	1.00	**1.00**
	Goal	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	0.95	**0.98**	1.00	1.00	**1.00**
	Legal	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	0.92	**0.96**	1.00	1.00	**1.00**
	Next	0.96	0.58	0.72	1.00	1.00	**1.00**	0.98	0.84	0.90	1.00	0.97	**0.99**
tictactoe	Terminal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	0.99	**0.99**	1.00	1.00	**1.00**
	Goal	1.00	0.29	0.46	1.00	1.00	**1.00**	1.00	0.97	**0.98**	0.91	0.67	0.77
	Legal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	Next	0.99	0.82	0.89	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
connectfour	Terminal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	0.96	**0.98**	1.00	1.00	**1.00**
	Goal	1.00	0.29	0.46	1.00	0.91	**0.95**	1.00	0.57	0.72	1.00	0.97	**0.98**
	Legal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	Next	1.00	0.95	**0.97**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 9: Transfer Learning (individual)

strong performance on their respective games and decent transfer learning capability on other games.

4.5 Mix Training

Game	Description	hamilton			knightstour			tictactoe			connectfour		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
mix6	Terminal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	Goal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	0.97	**0.98**	0.98	0.99	**0.98**
	Legal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	Next	0.95	1.00	**0.97**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
mix6-non-flat	Terminal	1.00	0.87	0.93	1.00	1.00	**1.00**	0.00	0.00	0.00	0.00	0.00	0.00
	Goal	1.00	0.90	0.93	1.00	1.00	**1.00**	0.00	0.00	0.00	0.00	0.00	0.00
	Legal	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	1.00	0.26	1.00	0.41
	Next	0.95	0.90	0.93	1.00	1.00	**1.00**	1.00	1.00	**1.00**	0.96	1.00	**0.98**

Table 10: GNN-Reasoner-v2 accuracy on mix training

To test the neural network's multi-domain adaptability, we conducted mix training experiment. Mix training can improve neural network‘s adaptability across different domains. By the mixing training datasets, the neural network can learn more diverse feature representations, thus performing better in various domains. Table 10 details the GNN-Reasoner-v2 accuracy on two different mix-trained neural networks. The mix6

neural network is trained on a mix of six games, including Hamilton, Knight's Tour, Tic-Tac-Toe, Connect-Four, Blocker and Blocks2player. The mix6-non-flat neural network is trained on the same games but without Manual Rule Flattening and Graph-based rule flattening. The purpose of this extra neural network is to attempt to address the issues mentioned in chapter 4.3 of the dataset by using mix training.

As shown in the table, the mix6 achieves excellent performance in all four games' prediction tasks. However, the mix6-non-flat performs poorly in the prediction tasks of terminal and goal in Tic-Tac-Toe and Connect-Four. Figure 27 illustrates the loss curves for the two neural networks mentioned above. The downward trend of the loss curves for mix6-non-flat neural network demonstrates that the neural network fails to properly learn knowledge about terminal and goal, leading to their loss curves converging early during training.

In summary, while the above conclusion indicates that mix training fails to address the issue of long-range dependencies, it does possess powerful multi-domain learning capabilities. If provided with a larger mixed dataset, it may achieve high accuracy predictions on the vast majority of unknown games.

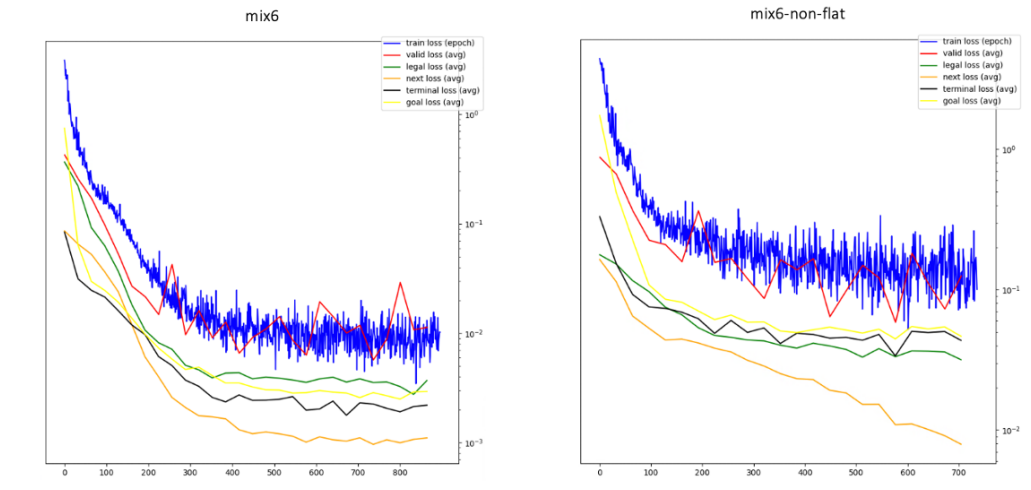

Figure 27: Loss curves for mix6 (left) and mix6-non-flat (right)

4.6 Principal Component Analysis on Transferred and Mixed Networks

To better understand the transferability of GNN-Reasoner-v2 across various games, we applied Principal Component Analysis (PCA). PCA is a statistical method utilized for reducing dimensionality and visualizing data. It aids in visualizing high-dimensional data by projecting it onto a lower-dimensional space. Representing data in two or three dimensions via PCA facilitates improved comprehension and interpretation of the inherent structure and relationships within the dataset. Figure 28 depicts the Principal Component Analysis (PCA) results of four datasets used in section 4.4, each containing 100 samples per game.

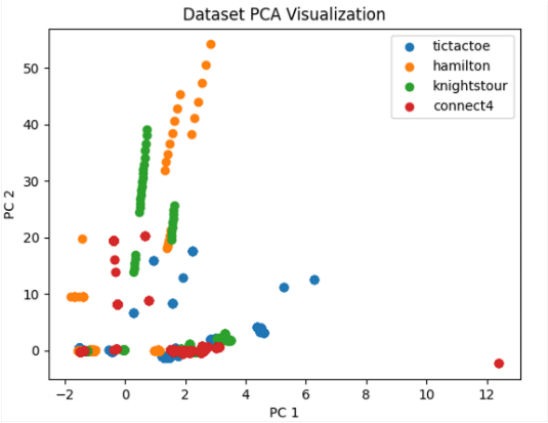

Figure 28: Principal component analysis on four individual games

The data points are primarily concentrated in the bottom-left corner of the plot, but there is also some distribution in the middle-upper region of the plot, indicating a degree of similarity in certain features. However, the data points exhibit a scattered trend along the directions of Principal Component 1 (PC1) and Principal Component 2 (PC2), revealing significant differences among them in other features. Among the four games, the distribution of Hamilton and Knight's Tour is particularly evident, mainly dispersed upward along PC2. In particular, for Hamilton, its data points have higher values along

PC2 compared to the other three games. Overall, In these games, although they have different game rules, they are all based on the Game Description Language (GDL). This leads to a certain degree of similarity in their data after undergoing Reversed Instantiated Rule Graph (R-IRG) subtree labeling and vector embedding.

Figure 29 illustrates the feature extraction capabilities of five different neural networks trained on four different game datasets including Knight's Tour, Hamilton, Connect-Four, and Tic-Tac-Toe, as well as a mix6 neural network trained on a mixed dataset. Through these PCA plots, we can observe the neural networks' understanding and differentiation abilities across different game datasets.

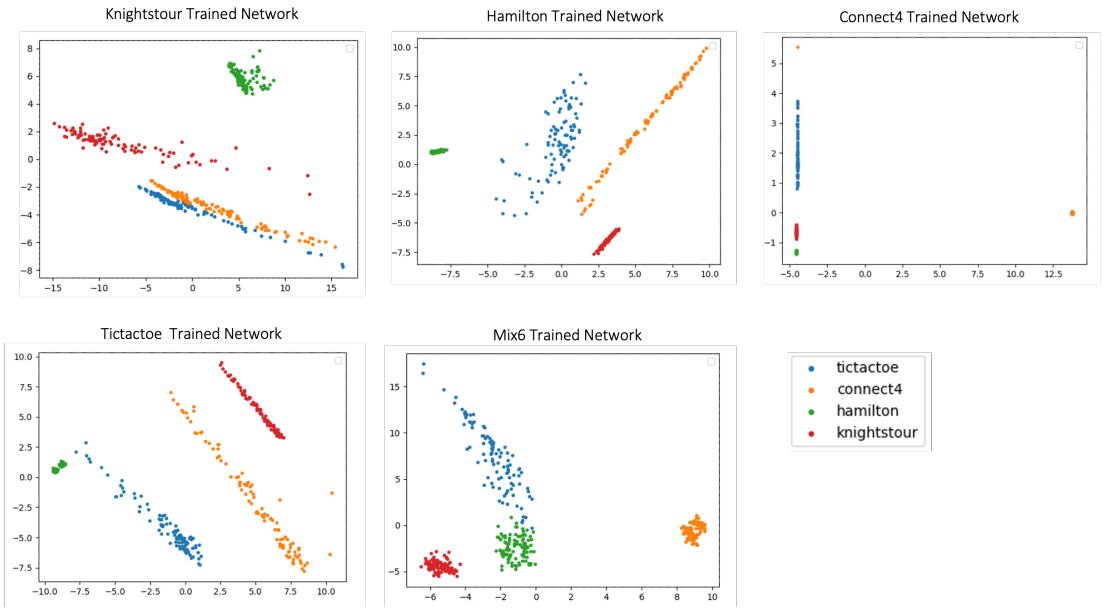

Figure 29: Principal component analysis of intermediate graph embedding generated by neural reasoner prior to output layers.

All trained network data points are relatively clear according to the game dataset, indicating their ability to identify different games based on the features of their own trained game data. One notable observation is the performance of the datasets in the Knight's Tour trained network and Hamilton trained network. The data points for Tic-Tac-Toe and Connect-Four are clustered to each other in the Knight's Tour trained network, suggesting that the neural network extracts common patterns from the

Tic-Tac-Toe and Connect-Four datasets.

Overall, by combining figure 28 with figure 29, we can observe that the datasets of these games exhibit a significant number of similar features. These shared characteristics contribute to the robust transfer learning capability of the models.

4.7 Graph Ablation Study

Due to the significant influence of various graph structures on neural network accuracy, we undertook a graph ablation study. This experiment is aimed at gaining deeper insights into the functioning of neural networks and exploring a more general method for accurately addressing long-range dependencies without utilizing Rule Flattening. Rule Flattening greatly reduces the mean path length of the Reversed Instantiated Rule Graph (R-IRG) and mitigates the severe imbalance among various nodes in the dataset that need to be predicted. This approach enables the neural network to better learn GDL-related knowledge. However, Rule Flattening typically requires manual modifications to the GDL, which is not a good solution. If we could automate operations directly on the graph without modifying the original GDL, perhaps we could address this issue more effectively. In this section, we will elaborate on the diverse graph structures we have investigated and provide corresponding data analysis.

The majority of experiments were conducted using Connect-Four and Tic-Tac-Toe. Connect-Four was chosen for experimentation because it exhibits greater complexity in its GDL compared to other games, which allows for a better assessment of the neural network's performance in complex environments. On the other hand, Tic-Tac-Toe was selected because its graph is easier to modify and ensures that the Labelling function in the Reversed Instantiated Rule Graph remains intact. Additionally, due to the substantial GPU memory consumption associated with the Reversed Instantiated Rule Graph (R-IRG) generated by Connect-Four, some experiments in this chapter apply Tic-Tac-Toe

as a substitute.

4.7.1 Reversed Instantiated Rule Graph without Graph-Based Rule Flattening

As mentioned in section 3.2.5.1, Graph-based Rule Flattening is primarily applied to problems related to flattening rules containing negation problem. The terminal and goal rules in Connect-Four game description language (GDL) contain a large number of rules associated with the negation problem. In this section, we will discuss the accuracy of two distinct neural networks trained on Connect-Four. The first neural network is trained on Reversed Instantiated Rule Graph (R-IRG) mentioned in section 4.2, and the second neural network is trained on the R-IRG that does not employ Graph-based Rule Flattening.

Game	Description	Terminal			Goal			Legal			Next		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
connectfour	R-IRG	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	non-graph-flat	0.89	0.73	0.80	0.70	0.43	0.53	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 11: GNN-Reasoner-v2 accuracy in Connect-Four trained on R-IRG and R-IRG without Graph-based Rule Flattening

Table 11 illustrates the difference in accuracy between two types of neural networks. Due to the significant instability observed during the training of the non-graph-flat neural network, the presented result is the average of the outcomes from training two separate neural networks. This instability does not always result in consistently poor neural network performance but can vary between highs and lows. Given that the terminal and goal GDL rules in Connect-Four involve numerous negation problems, it is understandable that the non-graph-flat neural network exhibits lower accuracy compared to the standard neural network.

4.7.2 Non-Reversed Instantiated Rule Graph

In Graph Neural Networks (GNNs) [43], most message passing algorithms is typically designed to aggregate information based on direction in directed graphs. In directed graphs, edges have directionality, indicating a causal or influential relationship between nodes, and the direction of edges is one of their features. In GNN-Reasoner [9], the Rule Graph is a directed graph, but its information aggregation direction is from the nodes to be predicted to the bottommost child nodes. The Rule Graph can often be seen as a collection of various tree structures. In the design of GNN-Reasoner, the root of the tree often aggregates more information, while the nodes that need to be predicted are aggregated with sufficient information. This can lead to poor neural network accuracy.

To address this issue, GNN-Reasoner [9] employs two types of GAT-blocks to calculate both the original edges and the reversed edges. This approach helps in predicting legal actions and next fluents but comes with additional computational resource consumption. Consequently, when experimenting with it, we found that even running Tic-Tac-Toe-Large is not feasible due to resource constraints.

Game	Description	Terminal			Goal			Legal			Next		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
connectfour	R-IRG	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	non-reversed	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 12: GNN-Reasoner-v2 accuracy in connectfour trained on Reversed Instantiated Rule Graph (R-IRG) and non-reversed IRG

In this test, we utilized a single-layer GAT block for training. Table 12 shows the accuracy of GNN-Reaoner-v2 trained on Reversed Instantiated Rule Graph (R-IRG) and non-reversed Instantiated Rule Graph in the context of Connect-Four. The data shows that the F1 scores for non-reversed IRG in Connect-Four are all zero, which further reinforces the earlier points we discussed and justifies our decision to utilize R-IRG in GNN-Reasoner-v2.

4.7.3 Undirected Instantiated Rule Graph

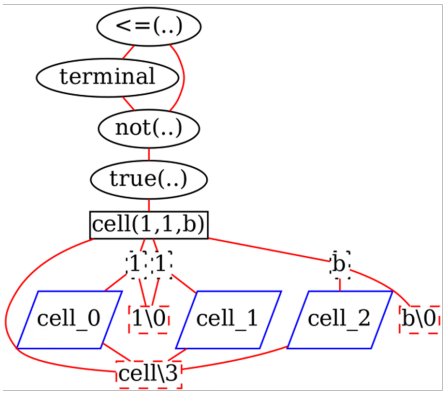

Figure 30: Undirected Rule Flattened Instantiated Rule Graph for "(<= terminal not open)" in Tic-Tac-Toe

In addition to using a single GAT block and a Reversed Instantiated Rule Graph (R-IRG) to reduce computational resource consumption, another approach is to use a single GAT block and an Undirected Instantiated Rule Graph for the same purpose. In this section, we primarily compare the accuracy difference between the Undirected IRG and R-IRG in Tic-Tac-Toe. Figure 30 illustrates the Undirected Rule Flattened Instantiated Rule Graph of the GDL rule `(<= terminal not open)`. When dealing with undirected graphs, Graph Neural Networks (GNNs) conduct message passing and information aggregation to all connected nodes of each node.

Game	Description	Terminal			Goal			Legal			Next		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
tictactoe	R-IRG	1.00	0.96	**0.98**	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	undirected	1.00	0.95	**0.97**	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 13: GNN-Reasoner-v2 accuracy in Tic-Tac-Toe trained on R-IRG and undirected IRG

Table 13 displays the accuracy of the GNN-Reasoner-v2 for Tic-Tac-Toe when trained on the two mentioned graph structures. As demonstrated, the variance between them is minimal. However, we did not choose the undirected IRG due to its potential necessity to compute more edges in undirected graphs

4.7.4 Minimal Instantiated Rule Graph

Aiming to determine whether the performance of the neural network depends more on the complexity of neural networks or the uniqueness of graph structures, we conducted an experiment on a new graph structure: Minimal Reversed Instantiated Rule Graph. This graph builds on the Rule Graph proposed by GNN-Reasoner[9] by removing duplicate nodes, while retaining the same labelling functions, edge direction and vector embedding as the Reversed Instantiated Rule Graph (R-IRG). As illustrated in Figure 31, repeated nodes, such as `open` and `1`, are consolidated into a single node. This reduction significantly decreases the performance overhead on the dataset during the training process, allowing the neural network to accommodate a larger parameter size.

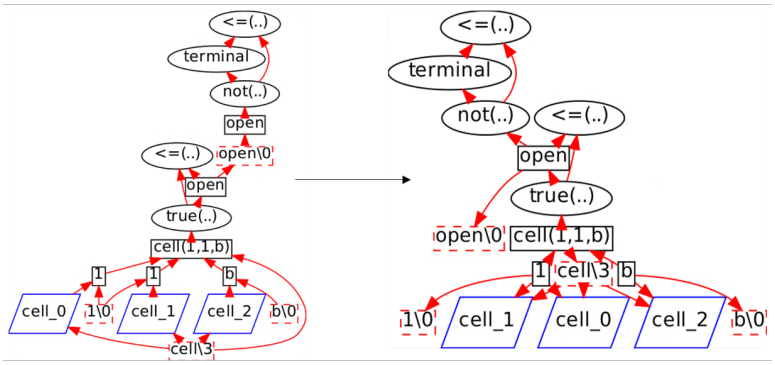

Figure 31: Difference between Rerversed GNN-Reasoner Instantiated Rule Graph [9] (left) and Minimal R-IRG(right)

Table 14 presents the neural network accuracy in Tic-Tac-Toe trained on the R-IRG, minimal R-IRG, and non-flattened Rule Graph. R-origin refers to the non-flattened Rule Graph mentioned in section 4.3 with reversed edges. Minimal-1 indicates that it utilized 1 hidden layer and 4 attention heads. Minimal-5 refers to it utilized 5 hidden layers and 6 attention heads. According to the data, both Minimal-5 and Minimal-1 demonstrate an accuracy of over 50% in predicting Terminal and Goal nodes. This suggests that the prediction accuracy of the neural network may not dependent on the number of hidden layers but rather on other components of the neural network or the

uniqueness of graph structures.

Game	Description	Terminal			Goal			Legal			Next		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
tictactoe	R-IRG	1.00	0.95	**0.97**	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	R-origin	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	minimal-5	0.62	0.78	0.69	0.89	0.66	0.76	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	minimal-1	0.74	0.73	0.73	0.88	0.43	0.58	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 14: GNN-Reasoner-v2 accuracy in R-IRG, minimal R-IRG and non-flattened Rule Graph proposed by GNN-Reasoner [9]

Table 15 presents the graph metrics for Tic-Tac-Toe across three types of graphs. The minimal R-IRG have a significantly smaller IRG size compared to the other two graphs, indicating their potential to accommodate more complex neural network architectures. However, minimal R-IRG exhibit a higher mean outdegree compared to the other graphs, which in turn may constrains their accuracy from another perspective.

Game	Description	IRG size		Outdegree	
		Nodes	Edges	Max	Mean
tictactoe	R-IRG	6332	14572	759	4.60
	R-origin	5328	12216	625	4.59
	minimal	**593**	**1857**	**52**	**8.26**

Table 15: Graph metrics for R-IRG, minimal R-IRG and non-flattened Rule Graph proposed by GNN-Reasoner [9]

Table 16 displays the mean path length of the four types of predictable nodes in Tic-Tac-Toe. The data indicates that the terminal and goal nodes of minimal R-IRG have a smaller mean path length compared to R-origin. The data demonstrates that reducing the mean path length of R-IRG is a highly effective method for improving the neural network's prediction accuracy.

The experiments conducted above indicate that the number of hidden layers or heads in the neural network does not significantly impact the neural network's prediction accuracy. Instead, reducing the mean path length of R-IRG is the key factor. Another valuable aspect of this experiment lies in demonstrating that there is still significant

Game	Description	Terminal	Goal	Legal	Next
tictactoe	R-IRG	2.15	2.54	1.68	1.66
	R-origin	**6.24**	**6.56**	1.68	1.66
	minimal	**3.16**	**3.33**	1.68	1.66

Table 16: Graph comparison of R-IRG, minimal R-IRG and non-flattened Rule Graph proposed by GNN-Reasoner (mean path length)

room for optimization in the Reversed Instantiated Rule Graph (R-IRG), particularly in terms of reducing the size of it.

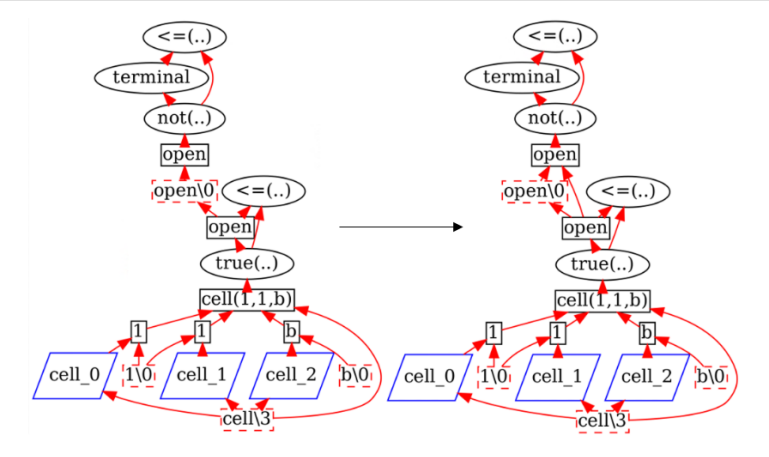

Figure 32: Difference between R-origin (left) and connected R-IRG (right)

4.7.5 Connected Reversed Instantiated Rule Graph

Connected Reversed Instantiated Rule Graph is also a graph that builds on the Rule Graph proposed by GNN-Reasoner[9] by connecting HB predicate nodes, while retaining the same labelling functions, edge direction and vector embedding as the Reversed Instantiated Rule Graph. As shown in figure Figure 32, it establish direct connections between all same `hb_head` nodes and `hb_body` nodes, without removing any nodes. In this experiment, we aim to observe whether the neural network's performance is affected by the shortest path length without significantly reducing the mean path length. Table 17 displays the differences in mean path length among the three types of R-IRGs.

We can observe that the mean path length of the connect R-IRG exhibits a certain degree of decline.

Game	Description	Terminal	Goal	Legal	Next
tictactoe	R-IRG	2.15	2.54	1.68	1.66
	R-origin	**6.24**	**6.56**	1.68	1.66
	connected	**5.43**	**5.74**	1.68	1.66

Table 17: Graph comparison of R-IRG, R-origin and connected R-IRG (mean path length)

The table 18 indicates that the neural network trained on connect R-IRG and R-origin both demonstrates significantly poor performance. This once again confirms the argument discussed earlier that neural networks are more sensitive to mean path length.

Game	Description	Terminal			Goal			Legal			Next		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
tictactoe	R-IRG	1.00	0.95	**0.97**	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	non-flat	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	connected	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 18: GNN-Reasoner-v2 accuracy in R-IRG, R-origin and connected R-IRG

4.7.6 No-Implication Reversed Instantiated Rule Graph

No-Implication R-IRG is a type of graph structure based on R-IRG, but it removes all `implication` nodes and the associated edges from the graph as shown in figure 33. In this graph structure, all nodes to be predicted become parent nodes, eliminating the need for GNNs to aggregate information towards implications.

Table 19 shows that no-implication R-IRG have accuracy similar to standard R-IRG, demonstrating that implications can be removed. However, implication nodes serve to establish relationships between various statements in GDL. To ensure that the R-IRG still retains this information, we decided not to remove implication nodes45.

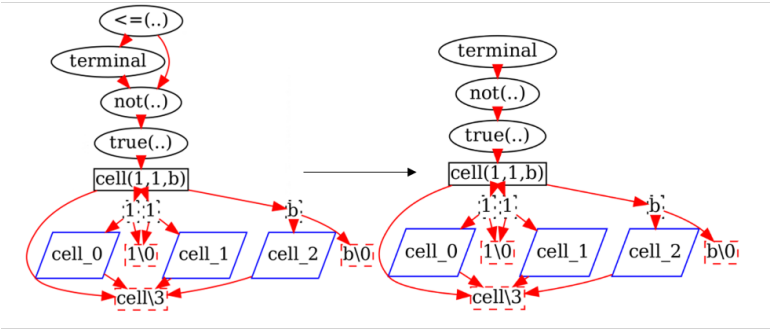

Figure 33: Difference between R-IRG (left) and no-implication (right) R-IRG

Game	Description	Terminal			Goal			Legal			Next		
		P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
tictactoe	R-IRG	1.00	0.95	**0.97**	1.00	0.97	**0.99**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
	no-implication	1.00	0.96	**0.98**	0.97	0.95	**0.96**	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 19: GNN-Reasoner-v2 accuracy in R-IRG and No-Implication R-IRG

4.8 Neural Networks Ablation Study

To evaluate the impact of different neural network architectures on the GNN-Reasoner-v2, we conducted neural network ablation studies. Table 20 presents eleven distinct neural networks, each with specific modifications or deletions outlined in the table. The remaining components are kept consistent with the default neural network mentioned in chapter 3. The notation `1-layer` signifies that the neural network utilizes only one hidden layer, while `1-head` indicates that the GAT neural network employs only one attention head.

Furthermore, `GATv2` signifies utilization of the GATv2Conv neural network [2] instead of the GATConv neural network [37], and `residual` indicates inclusion of residual connections in the neural network's GAT block, where the input X, A of the GAT block is directly passed to the fully-connected layer before GAT. Moreover, the `Transformer` neural network employs TransformerConv [27] instead of GAT, enabling analysis of whether predictions are based on global graph information or node relationships. Lastly, `non-pool` indicates removal of global attention pooling [14] from the neural network architecture.

For dataset features variations, The term `x-batch` denotes the batch size utilized by the neural network. `no-negation` indicates exclusion of negation labeling functions in the Reversed Instantiated Rule Graph (R-IRG) subtree labeling functions, while `no-labelling` implies absence of R-IRG subtree labeling altogether.

Game	Terminal			Goal			Legal			Next		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
default	1.00	0.99	**0.99**	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
1-layer	1.00	0.92	**0.96**	0.97	0.97	**0.97**	1.00	1.00	**1.00**	1.00	0.97	**0.98**
1-head	1.00	0.96	**0.98**	1.00	0.94	**0.97**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
no-pool	0.92	0.94	0.93	1.00	0.98	**0.99**	0.98	1.00	**0.99**	0.96	1.00	**0.98**
no-negation	1.00	0.96	**0.98**	0.87	0.52	0.65	1.00	0.97	**0.98**	1.00	1.00	**1.00**
no-labelling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8-batch	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16-batch	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GATv2	0.56	0.76	0.64	1.00	1.00	**1.00**	1.00	1.00	**1.00**	1.00	1.00	**1.00**
residual	0.87	0.96	0.91	0.88	0.97	0.92	0.76	1.00	0.86	1.00	1.00	**1.00**
transformer	1.00	0.96	**0.98**	1.00	0.97	**0.98**	1.00	1.00	**1.00**	1.00	1.00	**1.00**

Table 20: GNN-Reasoner-v2 accuracy on different neural networks

All experiments are based on Tic-Tac-Toe trained networks, due to transformer neural network require more computer memory resources, which we currently cannot meet. Figure 20 illustrates the accuracy of the aforementioned neural networks on the Tic-Tac-Toe. Neural networks `1-layer`, `1-head` and `no-pool` achieved similar accuracy to the default. These results once again demonstrate that the complexity of the model does not have a significant impact on its performance. Additionally, attention pooling does not significantly influence the performance of the neural network.

`8-batch` and `16-batch` attained notably poor accuracy. This can be attributed to the dataset's complexity, rendering the neural networks ineffective at handling small batch sizes. When using smaller batch sizes, the variance of computed gradients increases due to the smaller number of samples in each batch. This instability in gradient updates makes it more likely for gradient values to become outliers, leading to gradient explosion.

The `no-negation` neural network achieved decent accuracy across all prediction

tasks except for the goal prediction, where it performed poorly. This can be attributed to the presence of numerous goal rules with negation problem, leading to challenges in accurately capturing their representations. Without specific feature labeling for the `not` nodes in GDL, those goal rules may exhibit highly similar representations after vector embedding. Conversely, the `no-labelling` neural network demonstrated significantly poor accuracy, which is understandable as the dataset contained minimal features, making it challenging for the neural network to classify effectively.

The neural network model GATv2 demonstrated weak performance in predicting terminal states, possibly due to imbalances in the dataset. Initially, the addition of residual connections to the GAT block aimed to address the issues of gradient explosion. However, residual connections not only brought a decrease in accuracy, it also failed to resolve the problems of gradient explosion. Therefore, we choosed to remove them. The `transformer` neural network yielded similar accuracy to the default, but the significant additional computational overhead led us to abandon it. The `transformer` neural network may possess deeper potential. While its attention mechanism computes relationships between all nodes in the graph, limiting computations to only a half of the nodes at a time can significantly reduce hardware resource consumption.

4.9 Gradient Exploration Analysis

As described in chapter 4.8, when the neural network uses a batch size of 16 and 10, the neural network fails to train entirely. The process involves training for several epochs, after which the loss eventually becomes NaN, leading to training failure. To investigate the cause of this issue, we examined the gradient histograms when the batch sizes were 32 and 16.

Figure 34 displays the gradient histograms of the final fully connected layers of the GNN-Reasoner-v2 during the two training processes. In this figure, the horizontal axis

represents gradient values, while the vertical axis represents training epochs.

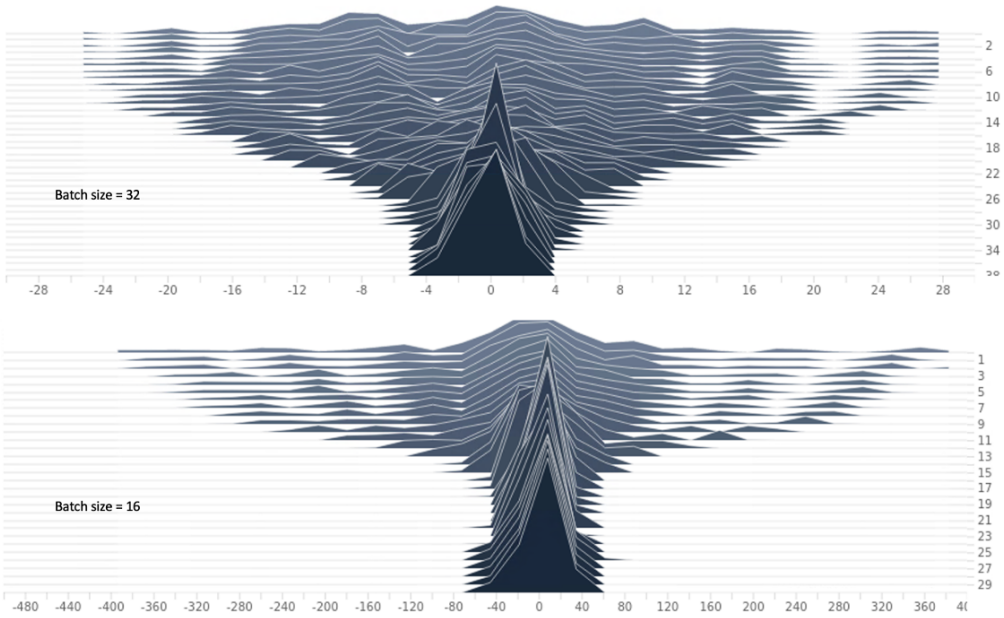

Figure 34: Gradient histograms in tictactoe when batch size = 32 (top) and 16 (bottom)

When the batch size is 32, the neural network converges smoothly. However, when the batch size is 16, the neural network fails at epoch 29, with the loss value becoming NaN. Upon observation, when the batch size is 32, the gradient values are distributed relatively evenly, with the majority clustered around zero, indicating effective training. When the batch size is 16, the gradient values are also evenly distributed, and the width of the peaks is rapidly decreasing. However, the range of gradients is extremely large (e.g., from -480 to +480), indicating a strong sign of gradient explosion. It makes sense that when gradients are large, the updates to neural network weights are also significant, potentially causing the neural network parameters to update to unreasonable values, resulting in the computation of NaN values for the loss function.

In addition to the experiment mentioned above, we trained two variants of the GNN-Reasoner-v2. One variant removed the prediction of goal values, while the other did not include the prediction of terminal states. Interestingly, both neural networks trained

smoothly with a batch size of 16 but encountered gradient explosion when the batch size was reduced to 8. This experiment highlights an important point: as the number of prediction tasks increases, the neural network struggles more to learn the dataset features effectively. As previously mentioned, their gradients exhibited exceptionally large magnitudes, which resulted in unstable updates and caused loss values to become NaN due to the excessively large gradient values.

4.10 Conclusion

In this chapter, we conducted a comprehensive analysis of GNN-Reasoner-v2. Firstly, we tested its prediction accuracy across various individual games, including terminal, goal, legal, and next predictions, demonstrating consistently high accuracy. These results were followed by ablation experiments on Rule Flattening, confirming its effectiveness in enhancing the neural network's performance.

Subsequently, we evaluated the transfer learning capabilities of GNN-Reasoner-v2, which revealed its ability to generalize knowledge from one game to others. Additionally, mix training experiments showed that the neural network's robustness increased with exposure to a diverse range of game rules.

Following these evaluations, we conducted ablation experiments on graph structures and neural network architectures to identify the optimal configurations and discern their impact on GNN-Reasoner-v2's performance. These experiments highlighted the mean path length, the balance of node types within the R-IRG and features design of the graph as a crucial factor influencing the neural network's performance, while indicating that the complexity of the neural network had minimal impact on its performance enhancement. Finally, we analyzed the impact of batch size on the neural network and concluded that smaller batch sizes are prone to gradient explosion issues.

Chapter 5

Discussion and Conclusion

In this thesis, we introduce an enhanced GGP Reasoner with high accuracy, GNN-Reasoner-v2, built upon the GNN-Resasoner [9]. Compared to GNN-Reasoner, GNN-Reasoner-v2 possesses the majority of functionalities required for a competent GGP-Reasoner. Furthermore, it requires significantly fewer hardware resources compared to GNN-Reasoner. Through transfer learning experiments, it showcases a notable transfer learning capabilities applicable to previously unlearned games. Furthermore, mixed-trained models further demonstrate the neural network's robustness when confronted with a mixture of GDL rules. It not only adapts to learning different games and performing reasoning tasks but also offers a more generalized data representation for the GGP and AI fields. The output of GNN-Reasoner-V2 is graph data, a more universal data representation that may aid in creating a more general deep reinforcement learning-based game agent.

In the analysis chapter, we explored a series of different graph structures aimed at addressing the longstanding challenge of long-range dependencies in GNNs directly at the graph level. These explorations encompass simplified graph architectures and a series of structures aimed at reducing the mean path length. Furthermore, we conduct ablation studies using various model architectures, including variations in GNNs types, pooling

methods, and model complexities. Through these experiments, we determine our current graph and neural network architecture. Finally, principal component analysis applied to the intermediate graph embedding visualizes the differences in learned inferences among trained networks, showing the model's robust transfer learning capabilities.

While GNN-Reasoner-v2 achieves a very high neural network performance, there remain several areas for improvement. Firstly, the current rule flattening method is manual, requiring researchers to rely on their own knowledge to perform rule flattening beforehand. Therefore, a fully automatic method for rule flattening could greatly accelerate research progress.

Furthermore, while Rule flattening can reduce the mean path length of R-IRG and mitigate the degree of imbalance in dataset distribution, it is still not a perfect solution for addressing long-range dependency. Additionally, there is still room for further enhancement in the design of dataset features. Furthermore, due to the grounded game description language (GDL) rules required by GNN-Reasoner-v2, the graph structures contain a large number of nodes and edges. These nodes and edges impose a significant demand on computational resources, thus becoming one of the primary obstacles in our research. A new graph structure or neural network architecture would be the correct approach to resolve this issue effectively. There are some potential solutions:

1. A graph structure only contains nodes related predicting nodes can significantly reduce the size of the graph, thereby reducing the demand for high-performance hardware.

2. By integrating techniques derived from natural language processing, such as leveraging transformer encoders, into the construction of dataset features, neural networks could potentially enhance their ability to capture inter-node features more effectively.

3. In both GNN-Reasoner [9] and GNN-Reasoner-v2, the graph structures often

further derive the parameters of a predicate into multiple sub-expression/constant nodes. These nodes constitute the largest portion of space in the current graph structure. Removing these nodes can provide a smaller graph structure.

4. Employ GNN during the dataset's feature extraction phase, followed by feeding these features into a Transformer. This method not only harnesses the strengths of GNN in capturing graph structural information but also utilizes the Transformer's enhanced ability to model global relationships and better handle long-range dependencies.

The GNN-Reasoner-v2 and GNN-Reasoner are same in that during the forward pass of training, the model performs two forward passes. One forward pass is for training legal, terminal, and goal states, while the other is for training next fluents. This is because predicting next fluents often requires additional information about the actions of the current player. Multiple forward passes imply computational costs far higher than a single forward pass. Reducing the number of forward passes to one could potentially greatly enhance the model's performance.

The aforementioned issues all have the potential for significant improvement with GNN-Reasoner-v2. We look forward to more researchers participating in these advancements.

References

[1] Tensorboard: A platform for visualizing, debugging, and monitoring deep learning models. [Accessed 4 March 2024].

[2] S. Brody, U. Alon, and E. Yahav. How attentive are graph attention networks?, 2022.

[3] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[4] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric, 2019. cite arxiv:1903.02428.

[5] S. Gelly and D. Silver. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM Press.

[6] M. R. Genesereth, N. Love, and B. Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62–72, 2005.

[7] A. Goldwaser and M. Thielscher. Deep reinforcement learning for general game playing. In *AAAI*, pages 1701–1708. AAAI Press, 2020.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] A. Gunawan, J. Ruan, and X. Huang. A Graph Neural Network Reasoner for Game Description Language. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, pages 443–452, 8 2022.

[10] A. Gunawan, J. Ruan, M. Thielscher, and A. Narayanan. Exploring a learning architecture for general game playing. In M. Gallagher, N. Moustafa, and E. Lakshika, editors, *Australasian Conference on Artificial Intelligence*, volume 12576 of *Lecture Notes in Computer Science*, pages 294–306. Springer, 2020.

[11] D. Hendrycks and K. Gimpel. Gaussian error linear units (gelus), 2023.

[12] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks, 2016. cite arxiv:1609.02907Comment: Published as a conference paper at ICLR 2017.

[13] A. Landau and sam schreiber. ggp.org, 2016.

[14] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks, 2015. cite arxiv:1511.05493Comment: Published as a conference paper in ICLR 2016. Fixed a typo.

[15] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection, 2018.

[16] I. Loshchilov and F. Hutter. Decoupled weight decay regularization, 2019.

[17] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification. *Stanford Logic Group Technical Report*, LG-2006-01, 2008.

[18] B. B. Max-Planck. Monte carlo go. 1993.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[20] M. Newborn. *Deep Blue - an artificial intelligence milestone.* Springer, 2003.

[21] A. Newell, J. C. Shaw, and H. A. Simon. Chess-playing programs and the problem of complexity. *IBM J. Res. Dev.*, 2(4):320–335, 1958.

[22] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks, 2012. cite arxiv:1211.5063Comment: Improved description of the exploding gradient problem and description and analysis of the vanishing gradient problem.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct. 1986.

[24] A. Samuel. Some studies in machine learning using the game of checkers (reprinted from journal of research and development, vol 3, 1959). *Ibm Journal of Research and Development*, 44:207–226, 01 2000.

[25] S. Schiffel and M. Thielscher. Reasoning about general games described in gdl-ii. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):846–851, Aug 2011.

[26] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.

[27] Y. Shi, Z. Huang, W. Wang, H. Zhong, S. Feng, and Y. Sun. Masked label prediction: Unified massage passing model for semi-supervised classification. *CoRR*, abs/2009.03509, 2020.

[28] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[29] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[30] C. F. Sironi and M. H. M. Winands. Optimizing propositional networks. In T. Cazenave, M. H. M. Winands, S. Edelkamp, S. Schiffel, M. Thielscher, and J. Togelius, editors, *CGW@IJCAI*, volume 705 of *Communications in Computer and Information Science*, pages 133–151, 2016.

[31] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[32] Y. Tekol and contributors. PySwip v0.2.10, 2020.

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

[34] G. Tesauro and T. J. Sejnowski. A parallel network that learns to play backgammon. Technical Report CCSR-88-2, Center for Complex Systems Research, University of Illinois, Urbana-Champaign, 1988.

[35] M. Thielscher. A general game description language for incomplete information games. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.

[36] M. Thielscher. A general game description language for incomplete information games. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2010.

[37] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks, 2018.

[38] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. P. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. P. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing. Starcraft ii: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[39] J.-N. Vittaut and J. Méhat. Fast instantiation of ggp game descriptions using prolog with tabling. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic*, pages 1121–1122, August 2014.

[40] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, New Jersey, second edition, 1947.

[41] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka. Representation Learning on Graphs with Jumping Knowledge Networks. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5453–5462, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[42] L. Zhao and L. Akoglu. Pairnorm: Tackling oversmoothing in gnns, 2020.

[43] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications, 2018. cite arxiv:1812.08434.