Master Thesis

Using Deep Learning to Improve Proof-Number Search in Two-Player Board Games

M.B. Hort

Master Thesis DKE-19-27

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Department of Data Science and Knowledge Engineering
of the Maastricht University

Thesis Committee:

Dr. M.H.M. Winands
Dr. C.B. Browne

Maastricht University
Faculty of Science and Engineering
Department of Data Science and Knowledge Engineering

June 25, 2019

Preface

This thesis was written at the Department of Data Science and Knowledge Engineering of the Maastricht University. The thesis describes my research on Proof-Number search and ways to improve the performance. At first, I would like to thank Dr. Mark Winands, not only for supervising my thesis, but also for providing me with much information on Proof-Number search and the game Lines of Action. Additionally, I would like to thank Chao Gao for providing me with expert games for the game of Hex. Finally, I would like to thank my family for supporting me.

Max Hort
Maastricht, June 2019

Abstract

Game playing is an important application for testing algorithms and approaches from Artificial Intelligence. Board games can be used to test search algorithms, which treat games as trees. Next to game playing, which is concerned is concerned with predicting the best move in a given position, game solving is concerned with assigning a game-theoretic value to position, e.g. to determine a win or loss. Proof-Number Search (PNS) is a search method that can be used to solve games and endgame positions.

In this thesis, it is investigated how PNS can be improved by introducing a new search variant, two-level df-pn, and search enhancements. Proposed search enhancements include move frequency, move availability and deep-learning. Improvements are evaluated on the games Lines of Action (LOA), Hex, Othello and Connect6. Therefore, the performance of enhancements and two-level df-pn is compared to existing approaches. Even though no improvements could be achieved on Connect6 and Othello, two-level df-pn solves a set of positions faster than other two-level methods for LOA and Hex.

Next, the frequency and availability enhancements are proposed in LOA. They reduce the number of evaluated nodes and execution time by up to 90% over a search without enhancements. Existing enhancements are outperformed by a short execution time of approximately a third, while maintaining a comparable number of nodes evaluated. Similarly, frequency and availability enhancements achieve reduction of almost 90% in execution time and nodes evaluated for Hex. Using deep learning to estimate the proof and disproof number of leaf nodes reduces the number of nodes evaluated by 80% in LOA and more than 50% in Hex. In both games, the deep-learning enhancement does not perform better than move frequency and availability enhancements. Instead of reducing the execution time, the deep learning enhancement increases it by a factor up to 20 in LOA and 40 in Hex, due to the computational overhead of making predictions with the network and preprocessing positions in the correct shape.

Finally, all enhancements have been applied to a LOA version with a smaller board, where the frequency and availability reduced the amount of nodes evaluated by 75% over a search without enhancements.

Contents

1 Introduction **1**
1.1 Artificial Intelligence and Games 1
1.2 Search and Games 2
1.3 Machine Learning and Games 3
1.4 Problem Statement and Research Questions 3
1.5 Thesis Structure 4

2 Proof-Number Search **5**
2.1 AND/OR Tree 5
2.2 Proof-Number Search 6
2.3 PN* 8
2.4 Proof-Number and Disproof-Number Search 8
2.5 Depth-First Proof-Number Search 9
2.6 Two-Level Search 9
2.6.1 Motivation 9
2.6.2 First Level 10
2.6.3 Second Level 10
2.6.4 PN^2 11
2.6.5 PDS-PN 11
2.6.6 dfpn-pn 11
2.7 Enhancements 14
2.7.1 Update Most Proving Node 14
2.7.2 Deleting Solved Subtrees 15
2.7.3 Mobility 15
2.7.4 ϵ-trick 15
2.7.5 Monte-Carlo Evaluation 16
2.8 Graph-History Interaction Problem 16
2.9 Other Variants 17
2.9.1 PDS* and df-pn$^+$ 18
2.9.2 Focused df-pn 18
2.9.3 Deep df-pn 18
2.9.4 Parallel Search 18

3 Game Domains **19**
3.1 Terms and Definitions . . . 19
3.1.1 Properties . . . 19
3.1.2 Solvability . . . 20
3.1.3 Complexity . . . 20
3.2 Lines of Action . . . 21
3.3 Hex . . . 23
3.4 Othello . . . 24
3.5 Connect6 . . . 25

4 Machine Learning **27**
4.1 Introduction to Machine Learning . . . 27
4.1.1 Reinforcement Learning . . . 28
4.1.2 Tasks . . . 28
4.2 Artificial Neural Network . . . 28
4.3 Convolutional Neural Network . . . 29
4.3.1 Convolution . . . 29
4.3.2 Subsampling (Pooling) . . . 30
4.4 Learning in Game Playing . . . 30

5 Proof and Disproof Number Estimation **31**
5.1 Move Frequency and Availability . . . 31
5.1.1 Data . . . 31
5.1.2 Implementation . . . 32
5.2 Deep Learning . . . 32
5.2.1 Data . . . 32
5.2.2 Input . . . 33
5.2.3 Core . . . 33
5.2.4 Output . . . 34
5.2.5 Training . . . 34

6 Experiments **37**
6.1 Experimental Setup . . . 37
6.2 Data . . . 38
6.2.1 Lines of Action . . . 38
6.2.2 Hex . . . 38
6.2.3 Othello . . . 39
6.2.4 Connect6 . . . 39
6.3 Base Performance . . . 40
6.4 Move Frequency and Availability . . . 42
6.5 Deep Learning - Training . . . 44
6.6 Deep Learning - Enhancement Performance . . . 46
6.7 Application to 5×5 Lines of Action . . . 48
6.8 Results and Discussion . . . 49
6.8.1 Lines of Action and Hex . . . 49
6.8.2 Othello and Connect6 . . . 50

6.8.3 Deep Learning 51

7 Conclusion **52**
7.1 Research Questions 52
7.2 Problem Statement 53
7.3 Future Research 53

Appendices **62**

A PNS-Pseudocode **63**

B Endgame Positions Hex **65**

C Data Augmentation **68**
C.1 LOA 68
C.2 Hex 68
C.3 Othello and Connect6 68

D Deep Learning: Output Visualization **69**
D.1 LOA 69
D.2 Hex 71
D.3 Othello 73
D.4 Connect6 75

List of Figures

2.1 Example of an AND/OR tree. OR nodes shown as squares, AND nodes as circles. 6
2.2 Example of PNS in an AND/OR tree. OR nodes shown as squares, AND nodes as circles, *pn* above *dpn* next to nodes. . . . 7
2.3 Example of df-pn. OR nodes shown as squares, AND nodes as circles, pn above dpn next to nodes, thresholds are given on the left. 10
2.4 Example of the GHI problem. 17

3.1 Classification of state space and search space complexity. Search-space complexity is increasing from left to right, state-space complexity is increasing from bottom to top. 21
3.2 (a) The initial position of LOA. (b) Possible moves of piece *1*, marked with *X*. (c) A terminal position, black wins. 22
3.3 (a) The initial position of LOA. (b) Possible moves of piece *1*, marked with *X*. (c) A terminal position, black wins. 24
3.4 (a) The initial position of Othello, possible moves for black are marked with *X*. (b) Terminal position in Othello, black wins with 42 pieces over white with 22 pieces. 25
3.5 (a) Example opening of Connect6 on an 8×8 board. (b) Terminal position in Connect, black wins with 6 connected pieces, marked with *X*. 26

4.1 Example of an artificial neural network with 1 hidden layer. . . . 29
4.2 Visualization of the convolution process. A kernel is placed over the input vector and applies a weighted sum. Taken from O'Shea and Nash (2015). 30

5.1 Example of a network model for 8×8 board games. Game has two results (win and loss). 35

6.1 Training loss of four different network architectures on each game. 45
6.2 Division of an 8×8 LOA board into four 5×5 boards. 48

D.1 Outcome Loss in % for NW2 on LOA. 69

D.2 Distance Loss in moves for NW2 on LOA. 70
D.3 Outcome Loss in % for NW1 on Hex. 71
D.4 Distance Loss in moves for NW1 on Hex. 72
D.5 Outcome Loss in % for NW1 on Othello. 73
D.6 Distance Loss in moves for NW1 on Othello. 74
D.7 Outcome Loss in % for NW4 on Connect6. 75
D.8 Distance Loss in moves for NW4 on Connect6. 76

List of Tables

5.1 Description of training data for deep learning after applying augmentation. . . . 33
5.2 Description of 4 network models for 8×8 board games. Nodes of consecutive convolutional layers are given. . . . 34
5.3 Description of 4 network models for 19×19 board games. Nodes of consecutive convolutional layers are given with filter size in braces. . . . 34
5.4 Descriptions of Layer that predict Outcome and Number of Moves until a terminal position is reached. . . . 36

6.1 Comparison of nodes evaluated and execution time for PNS variants on LOA positions. . . . 40
6.2 Comparison of standard deviation for a selection of PNS variants. 41
6.3 Comparison of nodes evaluated and execution time for PNS variants on Hex positions. . . . 41
6.4 Comparison of nodes evaluated and execution time for PNS variants on Othello positions. . . . 42
6.5 Comparison of nodes evaluated and execution time for move frequency and availability enhancements on LOA positions. . . . 42
6.6 Comparison of standard deviation for PNS with enhancements. . 43
6.7 Comparison of nodes evaluated and execution time for move frequency and availability enhancements on Hex positions. . . . 44
6.8 Comparison of nodes evaluated and execution time for move frequency and availability enhancements on Othello positions. . . . 44
6.9 Comparison of nodes evaluated and execution time in seconds for deep learning enhancements on LOA positions. . . . 46
6.10 Comparison of nodes evaluated and execution time in seconds for deep learning enhancements on Hex positions. . . . 47
6.11 Comparison of nodes evaluated and execution time in seconds for deep learning enhancements on Othello positions. . . . 47
6.12 Comparison of PN^2 to solve LOA on a 5×5 board. . . . 49
6.13 Comparison of nodes evaluated and execution time for search enhancements on LOA positions. . . . 49
6.14 Comparison of nodes evaluated and execution for search enhancements on Hex positions. . . . 50

Chapter 1

Introduction

In this thesis, Proof-Number search is investigated in order to find potential improvements. This chapter discusses the current state of game playing in Artificial Intelligence (AI), in Section 1.1. Following, Section 1.2 outlines the application of search in games and Section 1.3 describes the application of machine learning in games. Lastly, the problem statement of this thesis and three research questions are formulated in Section 1.4.

1.1 Artificial Intelligence and Games

For thousands of years, board games have been an important part of leisure activities. Often in a two-player setting, board games offer an environment to compete and compare skill. Humans use intuition to play games, whereas computers rely solely on algorithms to play. Board games offer a suitable environment for testing algorithms. The space of possible actions is well defined and observable. Additionally, the outcome of a game can easily be specified. Game playing is an important application for testing algorithms and approaches from AI. Programs and game engines are tested by playing against humans, and performance can be evaluated in competitions. This gives a framework for improvement for search algorithms. In 1953, advances on AI in game playing have been made by Turing, who described how computers could play chess (Bowden, 1953). Game engines have been using AI methods for the past decades. An example for this is Deep Blue (Campbell et al., 2002), the first chess engine to beat the world champion. Development and advances in AI in games are further encouraged by competitions such as the Computer Olympiad organized by the International Computer Games Association (ICGA).[1] The Computer Olympiad is a competition for engines in various games, including chess, Go, Checkers and Hex. In order for computers to play games, search techniques and machine learning are applied. Search techniques focus on investigating various

[1] https://icga.org/

moves, while machine learning is focused on learning to play from existing data, such as played games.

1.2 Search and Games

Search in games can be used to evaluate game positions, and for instance estimate the best move in a given position. Therefore, the game is interpreted as a tree where board positions are represented by nodes. The Minimax algorithm (von Neumann et al., 1944) is an early game search algorithm. A heuristic evaluation function is applied to assign scores to game positions. Two players compete against each other, whereas the first player tries to maximize his score and the second player minimizes it. Nodes are be evaluated in a depth-first manner to determine the score of a position. A significant improvement was achieved by an enhancement of the Minimax algorithm, the $\alpha\beta$ algorithm (Knuth and Moore, 1975). $\alpha\beta$ search evaluates nodes of a game tree in a depth-first manner, keeping track of the lower bound α and upper bound β that can be achieved, which enables a pruning of moves. Enhancements including move ordering (Schaeffer, 1989) and iterative deepening (Frey, 1983) lead to a further improvement of the performance. Another enhancement to $\alpha\beta$ search is Conspiracy Number Search(CNS) (McAllester, 1988). In CNS, the reliability of the Minimax score is considered, which can help to determine irrelevant search paths. A different, heuristic search method is Monte Carlo Tree Search (MCTS) (Coulom, 2007). Contrary to $\alpha\beta$ search, no evaluation function is needed, as nodes are evaluated based on the outcome of simulated games.

Another field of research is the solving of games. Contrary to game playing, which is concerned with predicting the best move in a given position, game solvers are concerned with assigning a theoretic value to position, determining a win or loss. Theoretical proofs are applied to games such as Nim (Bouton, 1901). In theory, with sufficient memory and time, all game positions can be solved by methods such as $\alpha\beta$ search, by creating every possible playout position until a terminal position is reached. Instead of evaluating game positions and assigning an estimated guess, only terminal positions are given a definite value. Thereby simple games such as Tic-Tac-Toe could be solved (Allis, 1994). Such an approach is likely to fail in practice, as narrow and deep search paths are frequently required to solve endgame positions. Proof-Number search (PNS) (Allis et al., 1994) is a search method, which originates from CNS, that can be used to solve endgame positions. Instead of following a depth-first approach, as in $\alpha\beta$ search, PNS is a best-first search algorithm that guides the search process to investigate the simplest node to prove, allowing for narrow search paths. Due to its efficiency, PNS has been applied to solve games including Checkers (Schaeffer et al., 2007), Connect-Four (Allis, 1988) and Fanorano (Schadd, 2011).

1.3 Machine Learning and Games

Deep learning has achieved popularity in multiple domains, ranging from robotics, image recognition to game playing. For board games, learning techniques can be used to analyze positions or played games to make predictions. While MCTS had difficulties in playing Go on a 19×19 board (Browne et al., 2012), recent media attention has been drawn to AlphaGo (Silver et al., 2016), which won a best of 5 series against the world champion in Go. AlphaGo enhanced MCTS search with deep learning to guide the search process. A base of expert games was used to train AlphaGo. Furthermore, AlphaZero has reached "superhuman performance" (Silver et al., 2017) without any human generated data to learn from. Not only does AlphaZero achieve superhuman performance in the game of Go, but the same approach has been transferred to Chess and Shogi, where it outperformed existing engines likewise.
This shows that deep learning methods can be used to improve the search performance in board games. Other search methods such as $\alpha\beta$ search (David et al., 2017) and PNS (Gao et al., 2017) can utilize deep learning to improve the search process. However, applications of learning techniques for estimating the value of board positions have not been investigated in PNS yet.

1.4 Problem Statement and Research Questions

When it comes to solving games or endgame positions, PNS is the most promising search technique to use. However, with problems and game domains of increasing complexity, performance improvements are required to allow for further application, advances and success of PNS. Performance improvements can be achieved in the nodes required to solve a position or the required time to solve them. To achieve improvements, previous work introduced PNS variants, search enhancements and applied domain knowledge.

Machine learning and deep learning offer techniques that can be applied to board games and achieved tremendous improvements in MCTS across multiple games. Learning techniques have not been fully evaluated for applicability for PNS. An analysis of different enhancements for PNS, especially learning techniques, can achieve improvements in performance, independent of game domain. Therefore, the following problem statement serves as a basis for this thesis:

How to improve Proof-Number search for two-player board games?

To address this problem statement, three research questions are investigated:

1. *How do the different PNS variants perform?*

The first research question is two-fold. On one hand, current PNS variants are evaluated on different board games in order to establish a baseline performance. On the other hand, a new PNS variant, two-level df-pn, is introduced and evaluated accordingly.

2. *How can proof and disproof number of leaf nodes be initialized to improve the search procedure, independent of game domain?*

Performance improvements and speedups can be achieved with extensions that initialize proof and disproof number. Often times, such extensions rely on domain knowledge to justify their application and are not easily transferable to other domains. In order to answer this question, a domain-independent initialization method is introduced and compared to the base performance (RQ1).

3. *Can deep learning be used to improve the search procedure by initializing proof and disproof number of leaf nodes?*

To answer the last question, it is not only of importance whether proof and disproof number can be estimated with Deep Learning techniques, but also how and with which accuracy. Deep Learning models are proposed to estimate proof and disproof number. Those models are used in PNS variants and the performance is evaluated and compared to the base performance (RQ1) and new methods (RQ2).

1.5 Thesis Structure

Chapter 2 describes PNS and its variants. Furthermore, a new PNS variant, two-layer df-pn is introduced. Additional information on existing enhancements of PNS is given. In Chapter 3, game domains are introduced. This includes an analysis of the current state of the art. Chapter 4 gives an outline on machine learning techniques and their application in game playing. In Chapter 5, methods for predicting proof and disproof number are investigated and defined. Chapter 6 carries out experiments. Those include measures of the baseline performance of PNS variants over various game domains, with and without enhancements. The use of deep learning in PNS is evaluated. Furthermore, results are evaluated and discussed. The performance of the deep learning estimation is compared with baseline performances. Lastly, Chapter 7 outlines limitations and future fields of research. Appendices are added at the end of the thesis.

Chapter 2

Proof-Number Search

In this chapter, the basics of Proof-Number search are explained. Firstly, AND/OR trees are defined in Section 2.1. Following, various Proof-Number search variants are explained in Sections 2.2-2.5. Section 2.6 describes two-level search variants. Search enhancements are introduced in Section 2.7. Lastly, a description of the Graph-History-Interaction problem (Section 2.8) is given and other PNS variants are mentioned (Section 2.9).

2.1 AND/OR Tree

AND/OR trees can be used to represent two-player games. They are rooted trees, consisting of two types of nodes, OR and AND nodes. OR nodes represent the first player to move; AND nodes represent the second player to move. Each node represents a board position and the execution of a legal move results in the creation of a child node. A node with children is called internal node; nodes without children are called leaf nodes. Each node is assigned with one of three values, in regard to the first player: win, loss, unknown. *win* shows that a node is a proven win, *loss* represents a proven loss and *unknown* indicates that further evaluations have to be conducted to determine the value of the node. If a node is a terminal position, either *win* or *loss* is assigned as its value, otherwise it is treated as *unknown*. Terminal nodes are leaf nodes. For every internal node, values are determined as follows:

- OR node:
  - Win if any of its children is a win
  - Loss if all of its children are a loss
- AND node:
  - Win if all of its children are a win
  - Loss if any of its children is a loss

The value of a tree is determined by its root value. As AND/OR trees are only able to distinguish between a win or loss, draws are interpreted as losses, as the first player fails to win. Figure 2.1 shows an example of an AND/OR tree.

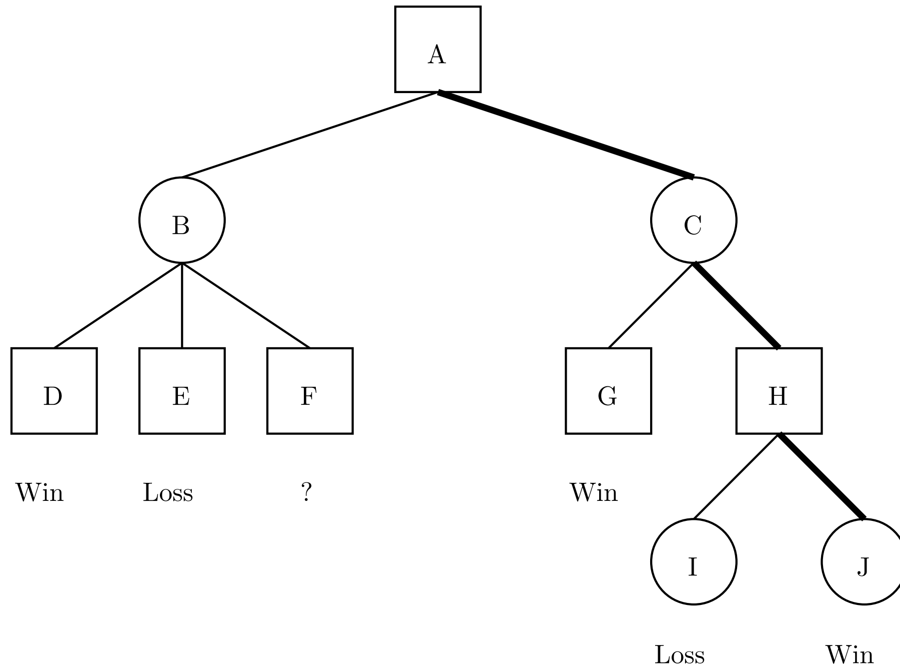

Figure 2.1: Example of an AND/OR tree. OR nodes shown as squares, AND nodes as circles.

Squares represent OR nodes and circles represent AND nodes. The root of the tree is an OR node. Node B is a loss for the first player, as a loss (E) can be forced by the second player. Therefore, F does not need to be evaluated anymore. The first player can force a win, indicated by the bold line, by opting for C. The second player can only choose to go for H, as G is an immediate win for the first player. However, H is a win for the first player as well, as one of its child nodes is a *win*. The value of the AND/OR tree is a *win* for the first player.

2.2 Proof-Number Search

Proof-Number search (PNS) is a best-first tree search algorithm applied to determine the definite value of AND/OR trees (Allis et al., 1994). No domain knowledge is required for this purpose, only terminal positions need to be recognized.
To guide the search process, PNS introduces two values that are used to describe nodes: proof number (*pn*) and disproof number (*dpn*). While *pn* represents the

minimum number of leaf nodes that have to be proven, as a win, for the node to be proven, *dpn* represents the minimum number of leaf nodes that have to be disproven, as a loss, for the node to be disproven. Non-terminal positions with unknown value are assigned with *pn* and *dpn* of 1. If a position is lost, it has a *pn* of ∞ and *dpn* of 0; if a position is won, it has a *pn* of 0 and *dpn* of ∞. *pn* and *dpn* of internal nodes are determined as follows:

- OR node:
  - *pn*: minimum *pn* of children
  - *dpn*: sum *dpn* of children
- AND node:
  - *pn*: sum *pn* of children
  - *dpn*: minimum *dpn* of children

PNS utilises a most-proving node (MPN) (Allis, 1994) to determine the next node for evaluation in a best-first manner. Therefore, two criteria are being considered, the shape of the tree as well as *pn* and *dpn* of nodes. Starting at the root, the MPN is selected by following a path that selects child nodes with minimum *pn* at OR nodes and child nodes with minimum *dpn* at AND nodes. The application of PNS can be investigated in an AND/OR tree as shown in Figure 2.2. It can be seen that nodes with unknown values are initialized with

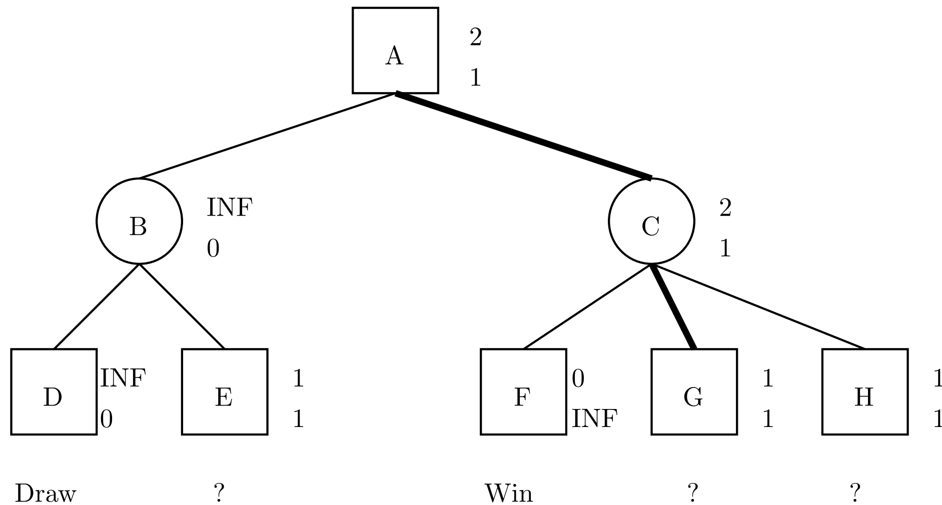

Figure 2.2: Example of PNS in an AND/OR tree. OR nodes shown as squares, AND nodes as circles, *pn* above *dpn* next to nodes.

a *pn* and *dpn* of 1 (E, G, H). The value of D is equal to a loss, as draws are interpreted as losses for the first player. The MPN is G, the path to the MPN

is highlighted bold. H could also be chosen as a MPN, depending on how ties are broken. The general PNS procedure is as follows:

1. Select MPN
2. Generate children (Expand)
3. Evaluate child nodes
4. Backpropagate *pn* and *dpn*

The given steps are repeated until the *pn* at the root is either 0 (win) or ∞ (loss). Pseudocode of the PNS algorithm can be found in Appendix A.

2.3 PN*

PN* is a variant of PNS, which transforms the best-first approach to a depth-first search (Seo et al., 2001). PN* applies iterative deepening as a means to solve for memory problems that can arise in PNS, which keeps the entire tree in memory. PN* performs several searches, starting from the root. A single search procedure is bounded by thresholds and is carried out until the thresholds are exceeded. Afterwards, a new search procedure is carried out with updated thresholds, until a solution is found. In iterative deepening approaches, identical nodes can be expanded and evaluated multiple times. In order to reduce this overhead, results of evaluated nodes are stored in a transposition table (Greenblatt et al., 1967). Not only does PN* apply multiple iterative deepening at the root node but also at all interior OR nodes. At AND nodes, thresholds assigned to children OR nodes are iteratively increased, starting from 1 (Sakuta and Iida, 2001). PN* gives a threshold to the subtree it is searching and continues the search process until the threshold is reached. Each iteration of PN* increments the search threshold at the root. In contrast to PNS, only *pn* is regarded in PN*.

2.4 Proof-Number and Disproof-Number Search

Proof-number and Disproof-number search (PDS) is an extension to the PN* algorithm (Nagai, 1999). PDS is a depth-first multiple iterative-deepening algorithm and uses *pn* and *dpn* to set thresholds for the search process. By using thresholds for *dpn*, PDS extends PN* in not only performing multiple iterative deepening at OR nodes but also at AND nodes. *pn* and *dpn* thresholds are given to a node and the subtree is searched while *pn* and *dpn* are within the thresholds. Based on this, PDS is able to find disproof and proof solutions to subtrees (Nagai, 1998). As well as PN*, PDS uses a transposition table to store results and speed up the search process. Winands et al. (2004b) apply the replacement scheme TwoBig (Breuker et al., 1996).

2.5 Depth-First Proof-Number Search

Another depth-first variant of PNS is depth-first proof-number search (df-pn) (Nagai and Imai, 1999). Unlike in PN* and PDS, no iterative deepening is performed at the root node. df-pn uses thresholds for *pn* and *dpn*. According to van den Herik and Winands (2008), *pn* thresholds are called *pnt* and *dpn* thresholds *dnt*. df-pn initializes *pnt* and *dnt* with ∞ and searches subtrees within the thresholds. The search terminates when a solution to the tree has been found. If *pnt* is equal to ∞ after termination of the search, the tree is a win for the first player, otherwise it is a loss. The search is carried on as long as *pn* and *dpn* of subtrees are smaller than the thresholds. While the search continues, thresholds are distributed among subtrees. The selection of the most proving node proceeds according to the procedure in PNS. Given a node n, if n is an OR node the child c with lowest *pn* is selected, while at an AND node the child c with the lowest *dpn* is selected. Additionally, df-pn keeps track of the child c_2 with second lowest *pn* and *dpn*, respectively. This information is used to determine the threshold for c at an OR node as follows:

$$pnt_c = min(pnt_n, pn_{c_2} + 1) \tag{2.1}$$

$$dnt_c = dnt_n + dn_c - sum(children_{dn}) \tag{2.2}$$

Thresholds at an AND node are determined in the following way:

$$pnt_c = pnt_n + pn_c - sum(children_{pn}) \tag{2.3}$$

$$dnt_c = min(dnt_n, dn_{c_2} + 1)) \tag{2.4}$$

The behaviour of df-pn is illustrated in Figure 2.3, according to Kishimoto et al. (2012). Thresholds at the root A are initialized with ∞. The path to the MPN D is highlighted in bold. B is selected at first, because of a lower *pn* than C. Afterwards, the thresholds are adjusted to $pnt(B) = pn(C)+1 = 4$ and $dnt(B) = dnt(A)-dpn(C) = \infty-1$, as B is evaluated while it is more favourable than the second best child of A, C. Next, D is chosen, as the child with lowest *dpn* and thresholds are readjusted to $pnt(D) = pnt(B) - pn(E) = 4 - 1 = 3$ and $dnt(D) = min(dnt(B), dpn(E) + 1) = 3$. In contrast to PDS, which is a depth-first algorithm that almost behaves in a best-first manner (Nagai, 1998), df-pn always select the MPN as PNS (Nagai, 2001).

2.6 Two-Level Search

In addition to the variants introduced, two-level variants have been proposed, which combine two variants with each other.

2.6.1 Motivation

The general motivation behind a two-level search process is to allow for deeper search paths while maintaining fewer nodes in memory (Allis, 1994). At first,

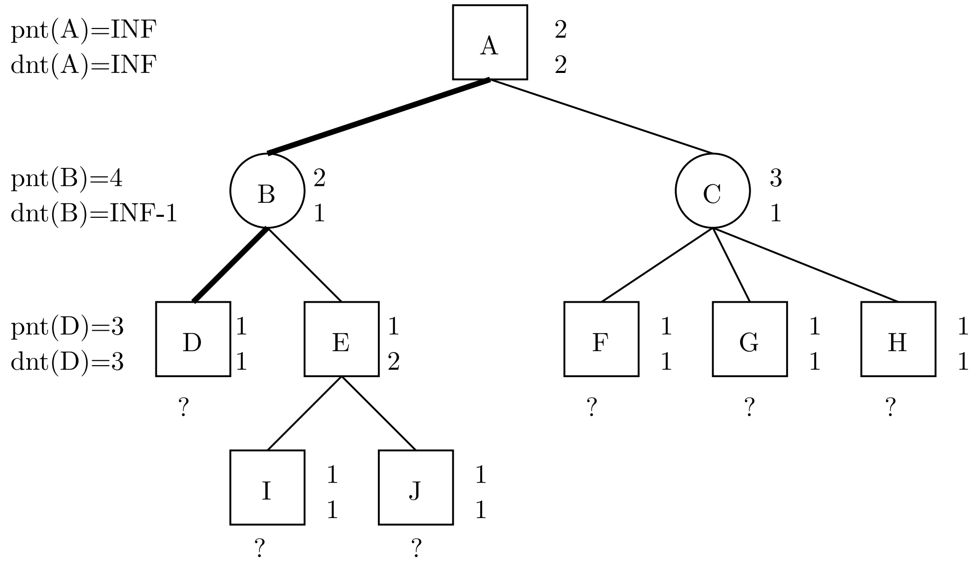

Figure 2.3: Example of df-pn. OR nodes shown as squares, AND nodes as circles, pn above dpn next to nodes, thresholds are given on the left.

a MPN is detected by the search on the first level. Afterwards, a second level search is started with the MPN as a root. *pn* and *dpn* of the MPN is set to the result of the second level search. After the search terminates, the second level search tree can be discarded as it is not needed for further evaluations, thereby reducing memory requirements.

2.6.2 First Level

The first-level search is kept in memory. The search is performed as usual with the only difference that the evaluation of a MPN initiates a search on the second level to determine *pn* and *dpn*. The first level applies a delayed evaluation of leaf nodes. This means that only leaf nodes that are selected as MPN are evaluated with a second level search.

2.6.3 Second Level

The second-level search starts with the MPN of the first level as its root. In this procedure, the size of the second level search is constrained by the size of the first level search tree. The initial approach to determine the second-level size was to restrain it to the size of the first level (Allis, 1994). Later, an approach according to the logistic-growth model (Berkey, 1988) has been proposed with a formula to determine the second level size (Breuker et al., 2001a):

$$f(x) = 1/(1 + \epsilon^{(a-x)/b}) \tag{2.5}$$

In this formula, x is the size of the first level search, a, b are hyper-parameters that can be tuned.

2.6.4 PN^2

The first two level PNS variant that has been proposed is PN^2 (Allis, 1994). The objective of PN^2 is to reduce the memory required by PNS, as the entire tree is stored in memory. The first level performs PNS to select a MPN, which is evaluated with another PNS search on the second level. As an improvement, the immediate children of the second-level root are added to PNS search tree on the first level (Breuker et al., 2001a).

2.6.5 PDS-PN

PDS-PN is a two-level search variant PDS on the first level and PNS on the second level (Winands et al., 2002). PDS-PN profits from advantages of both variants:

- PDS requires little memory
- PNS is fast

With a PDS search on the first level, the required memory to store the search tree in memory is relatively low. The second level performs PNS, which is faster than PDS, as it does not apply an iterative-deepening approach.

2.6.6 dfpn-pn

The concept of a two-level df-pn search was mentioned in van den Herik and Winands (2008) and Kishimoto et al. (2012), however was not implemented and tested in practice. Dfpn-pn is a two level search variant with a df-pn search on the first level and PNS on the second level. The df-pn algorithm is defined in a Negamax fashion (Kishimoto and Müller, 2004). Therefore, *pn* and *dpn* are changed to the following notation:

$$n.\phi = \begin{cases} \text{n.pn} & \text{if } n \text{ is an OR node} \\ \text{n.dpn} & \text{if } n \text{ is an AND node} \end{cases} \tag{2.6}$$

$$n.\delta = \begin{cases} \text{n.dpn} & \text{if } n \text{ is an OR node} \\ \text{n.pn} & \text{if } n \text{ is an AND node} \end{cases} \tag{2.7}$$

Pseudo-code of dfpn-pn can be seen in Algorithms 1 and 2. Algorithm 1 describes the general procedure of df-pn, extending it with additional functionality to carry out a second level search. The algorithm returns the value of the root. n.ϕ and n.δ are initialized with ∞ at the root and the function *MID* is called to perform multiple iterative deepening at nodes. *MID* (lines 10-33) traverses the subtree of a *node* in a depth-first manner within thresholds. If a node corresponds to a terminal position, it is evaluated and stored in the transposition

table (lines 11-14). The expansion procedure (lines 18-29) detects the best child of a node, while keeping track of the second best child. In lines 25-27, the enhancements towards two-level df-pn is made, if the child has not been evaluated before and is not in the transposition table, a PNS is performed on this child. *computeMaxNodes* is a function to limit the number of nodes of the second level search. Algorithm 2 contains utility functions that are used to determine the best child of a node, *SelectChild*, and the functions ΔMin and ΦSum. Both can be seen as equivalents of PNS, to determine the sum or minimum of *pn* and *dpn* of child nodes. *RetrieveProofAndDisproofNumbers* and *SaveProofAndDisproofNumbers* are functions that control the use of a transposition table.

Algorithm 1 Pseudo-code of the dfpn-pn algorithm

```
 1: function DFPN-PN(root)
 2:     root.φ ← ∞
 3:     root.δ ← ∞
 4:     MID(root)
 5:     if root.δ == ∞ then
 6:         return proven
 7:     else
 8:         return disproven
 9: // Perform search with thresholds
10: function MID(node)
11:     if IsTerminal(node) then
12:         Evaluate(node)
13:         // store node
14:         SafeProofAndDisproofNumbers(node,node.φ,node.δ)
15:     GenerateMoves(node)
16:     // Search within thresholds
17:     while node.φ > ΔMin(node) and node.δ > ΦSum(node) do
18:         C_best ← SelectChild(node,φ_c,δ_2)
19:         // Update thresholds
20:         C_best.φ ← node.δ + φ_c − ΦSum(node)
21:         C_best.δ ← min(N.φ, δ_2 + 1)
22:         // Extension for dfpn-pn
23:         // Check whether node has not been searched before
24:         if not RetrieveProofAndDisproofNumbers(C_best,φ,δ) then
25:             PN(C_best,computeMaxNodes())
26:             SafeProofAndDisproofNumbers(C_best,C_best.φ,C_best.δ)
27:         else
28:             MID(C_best)
29:     // Store search results
30:     node.φ ← ΔMin(node)
31:     node.δ ← ΦSum(node)
32:     SafeProofAndDisproofNumbers(node,node.φ,node.δ)
```

Algorithm 2 Utility functions of dfpn-pn algorithm

```
33: function SELECTCHILD(node,&φ_c,&δ_2)
34:     φ_c ← ∞
35:     δ_c ← ∞
36:     for for each child c of node do
37:         RETRIEVEPROOFANDDISPROOFNUMBERS(c,φ,δ)
38:         // Store smallest and second smallest δ
39:         if δ < δ_c then
40:             C_best ← c
41:             δ_2 ← δ_c
42:             φ_c ← φ
43:             δ_c ← δ
44:         else if δ < δ_2 then
45:             δ_2 ← δ
46:         if φ == ∞ then
47:             return C_best
48:     return C_best
49: // Compute smallest δ of node's children
50: function ΔMin(node)
51:     min ← ∞
52:     for for each child c of node do
53:         RETRIEVEPROOFANDDISPROOFNUMBERS(c,φ,δ)
54:         min ← min(min, δ)
55:     return min
56: // Compute sum of φ of node's children
57: function ΦSum(node)
58:     sum ← 0
59:     for for each child c of node do
60:         RETRIEVEPROOFANDDISPROOFNUMBERS(c,φ,δ)
61:         sum ← sum + φ
62:     return sum
```

2.7 Enhancements

To improve the performance of PNS and its variants, several enhancements have been proposed. Their use ranges from memory reductions, speed-ups to a guidance of the search procedure. A selection of variants is introduced in the following.

2.7.1 Update Most Proving Node

Whenever new nodes are created and initialized in the search tree, parent nodes are updated and a new is MPN selected. This procedure impacts the running

time, as the tree has to be traversed whenever nodes are being added and whenever a MPN is selected. A reduction of the traversal length reduces the running time of the search. Allis et al. (1994) proposed a method that stops the updating process of parent nodes as soon as *pn* and *dpn* remain unchanged. In order to find the next MPN, one can start from the stopping point rather than from the root, which reduces the number of nodes traversed.

2.7.2 Deleting Solved Subtrees

Nodes in a PNS search tree have three different values: they are either proven, disproven or unknown. An enhancement to reduce the memory usage of PNS is to delete the subtrees of proven and disproven nodes (Allis, 1994). The value of a proven or disproven node will not change anymore. It is therefore not required to keep the subtree in memory and it can be discarded to free memory.

2.7.3 Mobility

Rather than initializing *pn* and *dpn* with 1 for every node, a more elaborate measure can be used. One method for this is to initialize nodes with the resulting mobility (number of moves) (Allis, 1988). Using mobility for initializing *pn* and *dpn* has the effect of performing a lookahead of one move, where every child node gets initialized with *pn* and *dpn* of 1, and the sum is backpropagated to the parent. OR nodes are initialized with $pn = 1; dpn = n$, AND nodes are initialized with $pn = n; dpn = 1$, where n denotes the number of children of the node. Initializing with mobility can achieve speed-up factors of 5-6 in execution time (Winands and Uiterwijk, 2001). By using a different initialization method, one can achieve a tradeoff between speed-ups, due to fewer nodes in the search tree and computational overhead to compute the new initialization method. This enhancement can be applied to domains with non-uniform branching factor. The mobility enhancement is difficult to apply to games with uniform branching factor, as nodes on the same depth remain with equal *pn* and *dpn*, unless terminal positions are reached.

2.7.4 ϵ-trick

Iterative deepening PNS variants lead to reproductions of subtrees in the search. The ϵ-trick (Pawlewicz and Lew, 2007) can be applied to reduce the number of reproductions in PDS and df-pn in searching subtrees to a higher degree. Thereby, more time is spent on exploring a subtree rather than alternating subtrees more frequently. The alternation between subtrees is called see-saw effect. The ϵ-trick can be applied to df-pn by changing the computation of thresholds, as seen in Pawlewicz and Lew (2007), to the following:

$$pnt_c = min(pnt_n, pn_{c_2} \times (1 + \epsilon)) \tag{2.8}$$

$$dnt_c = min(dnt_n, dn_{c_2} \times (1 + \epsilon)) \tag{2.9}$$

Similarly, the ϵ-trick can be applied to PDS. Whereas initially, bounds are computed as follows:

$$pnt_c = pnt_c + 1 \tag{2.10}$$
$$dnt_c = dnt_c + 1 \tag{2.11}$$

ϵ-trick thresholds are:

$$pnt_c = pnt_c \times (1 + \epsilon) \tag{2.12}$$
$$dnt_c = dnt_c \times (1 + \epsilon) \tag{2.13}$$

ϵ is a real number greater than 0.

2.7.5 Monte-Carlo Evaluation

A combination of MCTS and PNS has been evaluated in Saito et al. (2006) for the game of Go. The PNS procedure is followed, but rather than initializing *pn* and *dpn* with unity, up to 20 Monte-Carlo evaluations have been executed to create estimates.

2.8 Graph-History Interaction Problem

The Graph-History Interaction (GHI) problem occurs if the path (history of played moves) is relevant for determining results of a game (Palay, 1985). Ignoring paths could result in an incorrect evaluation of game positions when using transposition tables. An example for this can be seen in chess, where a game results in a draw if the same position is reached for the third time. Using a transposition table, one might store a loss to the respective positions, even though the position is only a loss based on the chosen path, containing 3 repetitions. If the same position is reached from a different path, obtaining "loss" from the transposition table would assign an incorrect value. Therefore, to ensure correctness, the graph history cannot be ignored. Consider the example given in Figure 2.4. Following the path $[A, B, D, F]$, if the next chosen node is F, a repetition occurs. The game can be determined as a draw and D and F are stored in a transposition table accordingly. If F is reached via the path $[A, C, E]$, the value is unknown. By using the stored draw, from the transposition table, incorrect assumptions are being made, which outlines the GHI problem. Naive approaches to dealing with the GHI problem include not using transposition tables (Palay, 1985) or not storing or flagging information for paths with cycles (Campbell, 1985). The first approach would lead to a general loss of speed-ups that transposition tables offer, while the second approach would only result in the loss of a few entries as the GHI problem "does occur to appear infrequently" (Campbell, 1985). Kishimoto and Müller (2005) mention that Tsume-shogi programs solve the GHI problem by not storing disproofs with repetitions. Breuker et al. (2001b) propose an algorithm that flags possible draws. Nodes are stored in a base node with potentially multiple twin nodes

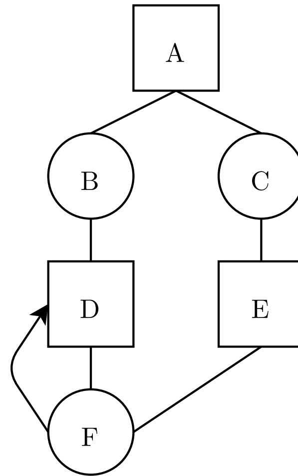

Figure 2.4: Example of the GHI problem.

of the same position reached from different paths. An implementation within df-pn ignores nodes that result in repetitions by returning to the parent and continuing the search (Nagai, 2001). The search result is ensured to not contain any repetition. Kishimoto and Müller (2005) store two table entries for *base* and *twin*, where the twin table contains the path to the node. When retrieving information from the transposition table, paths are compared and potential GHI problems detected. An experiment showed that this approach can solve 2 (out of 136) positions more than a solution ignoring the GHI problem entirely.

In the following, an approach according to Campbell (1985) is followed. Required repetitions for a draw are set to 2 for every game. An adjustment is done in regards to the flagging of transposition table entries with repetitions. At most two positions are flagged. If the node with repetition is an OR node, the previous AND node is flagged as well, otherwise only the OR node is flagged. In the example given in Figure 2.4, D and F would be flagged, to ensure that other path are not retrieving incorrect values. We can assume that this approach performs well due to the infrequency of GHI problems occurring.

2.9 Other Variants

While the aforementioned PNS variants are the most popular and well-known, other variants have been create during the course of the last years. A selection of those are outlined briefly.

2.9.1 PDS* and df-pn$^+$

PDS* is an extension to PDS, which incorporates additional information at nodes (Nagai, 1999). PDS* aims at distinguishing promising variants and search them deeper. Therefore, additional information is added to nodes:

- $cost(n, n_{child})$: effort from node n to a child node n_{child}
- $h(n)$: estimated effort to reach a solution from node n

The same extension was applied to df-pn, with the name df-pn$^+$ (Nagai and Imai, 1999).

2.9.2 Focused df-pn

Focused df-pn (FDPFN) is an extension to df-pn that focuses search effort on strong moves (Henderson, 2010). Instead of solving every sibling, including weak and strong moves, a heuristic method is applied to rank nodes and only strong moves are being considered at first. The number of children considered is bound by a fixed proportion. This reduces the branching factor of search trees.

2.9.3 Deep df-pn

Deep df-pn is a df-pn variant that aims at reducing the seesaw effect (Zhang et al., 2017). Instead of initializing *pn* and *dpn* with 1, they are being set by a function regarding their depth. Adjustments of parameters allow for changing between a broad search to a deeper search.

2.9.4 Parallel Search

The ability to solve larger problems, or to solve problems with shorter computation times can be improved by incorporating multiple processors and performing PNS in parallel. Examples for a parallel PNS algorithm are randomized parallel Proof-Number Search (RPPNS) (Saito et al., 2010) and job-level Proof-Number Search (JL-PNS) (Wu et al., 2011).

Chapter 3

Game Domains

This chapter describes board games in general, considering properties and measures of complexity. Section 3.1 gives a definition of relevant terms in game playing, putting emphasis on solvability and complexity. The following sections introduce the four games analyzed in this thesis. This explanation begins with Lines of Action (3.2), followed by Hex (3.3), Othello (3.4) and Connect6 (3.5). Additionally, an overview of the state of the art in search for each game is given.

3.1 Terms and Definitions

This section gives a general overview of characteristics of board games. Properties are analyzed and solvability and complexity are considered in detail.

3.1.1 Properties

Properties can be used to classify games in order to decide on correct methods to analyze them (Nijssen, 2013).

Number of Players. A vital property of games is the number of players. Often times, games are played by one or two players. Games with more than two players are called multi-player games.

Outcome. The outcome of games indicates how winners and losers are determined. Based on the number of players, a multitude of winners and losers can advance from a game. It can furthermore be distinguished between competitive and cooperative games. In competitive games, players try to win and therefore make their opponents lose. In cooperative games, players have the same goal and achieve mutual outcomes. Competitive two-player games are often treated as zero-sum games, where a win for one player results in a loss for the other. Not every game has to end with a winner. Those games include draws as a zero outcome, where no player won but also no player lost.

Information. If every bit of information in a game is available to every player during the entirety of the game, it is called game with perfect-information. Games with imperfect-information include aspects which are not known to all players, such as a the order of cards in shuffled deck of cards.

Chance. Chance in games shows whether randomness is included. Deterministic games do not include any elements of chance, whereas non-deterministic do. An example for chance is the use of dice in a game to determine the outcome of an action.

Flow. The flow of a game determines when moves are performed by players. This can occur in a sequential manner, where a player moves individually at a specified point of the game. Another option is for players to make moves simultaneously.

Symmetry. Symmetry in games indicates whether players have the same goal and try to achieve a win under identical conditions. In symmetric games, each player follows the same goal, whereas player can pursue different goals in asymmetric games.

Game domains considered in this thesis are deterministic, symmetric two-player, zero-sum games with perfect information. Players alternate in turns to perform moves.

3.1.2 Solvability

Solvability is a game property that determines the ability of creating strategies to achieve the best game-theoretic outcome in a position (Allis, 1994). Three different levels of solving games exist:

Ultra-weakly. The game-theoretic value of the initial position under exact play is known. It is not required to know how to achieve the outcome.

Weakly. A strategy has been devised for the initial position to at least achieve the game-theoretic value. It is not required to achieve a better outcome if the chance arises.

Strongly. For every position, a strategy can be determined to achieve the game-theoretic value of the position under exact play.

The state of solving of a game can help in determining the ability to solve endgame positions or entire games.

3.1.3 Complexity

When it comes to solving games, the characteristic of complexity becomes relevant. One has to know how many options have to be considered when taking actions and how much storage could be required to store game trees in memory. It is therefore of importance to consider two types of complexity spaces.

State Space. The state space shows how many possible positions can be achieved in a game.

Search Space. The search space complexity gives an intuition on the potential size of search trees when searching a game. The shape is determined by the search depth, the number of moves until the game terminates, and the search width, the number of possible action in a given position.

Inspired by van den Herik et al. (2002), a characterisation in complexity allows for an adequate choice of search methods, as indicated in Figure 3.1:

State-space complexity		
	Category 3 if solvable, by knowledge based methods	**Category 4** unsolvable
	Category 1 solvable by any method	**Category 2** if solvable, by brute-force

Search-space complexity

Figure 3.1: Classification of state space and search space complexity. Search-space complexity is increasing from left to right, state-space complexity is increasing from bottom to top.

3.2 Lines of Action

Game Description. Lines of Action (LOA) is a two-player game played on an 8×8 board. Initially, each player starts with 12 pieces. The first player has black pieces and the second player white pieces. Black pieces are placed on the top and bottom row of the board, white pieces are placed on the leftmost and rightmost columns. Corners of the board remain empty. The starting position of LOA can be seen in Figure 3.2 (a). Both players alternate in moving one of their pieces. Pieces can move in 8 directions, horizontally, vertically or diagonally. The distance of a move is equal to the number of pieces of either color in the line of movement. Pieces are allowed to jump over pieces of the same color, but

not over pieces of a different color. If a piece is placed on top of a piece with different color, this piece is captured and removed from the board. Pieces of the same color are not allowed to be captured. Potential moves of a piece are shown in Figure 3.2 (b). If a player is not able to perform a move, he passes his turn and the other player continues. The game ends in a draw after two consecutive passes. Additionally, the game ends when one of the players connected all of his pieces to a unit, where the distance of each piece to the closest other piece is at most one, as shown in Figure 3.2 (c). Connections can be formed in any of the movement directions. A characteristic of LOA is, that both player can end the game with the same move, by creating a connected unit of pieces each. This results in a draw. Another terminal condition for LOA is a three-fold repetition if the player to move has encountered the current position for the third time, the game ends in a draw.

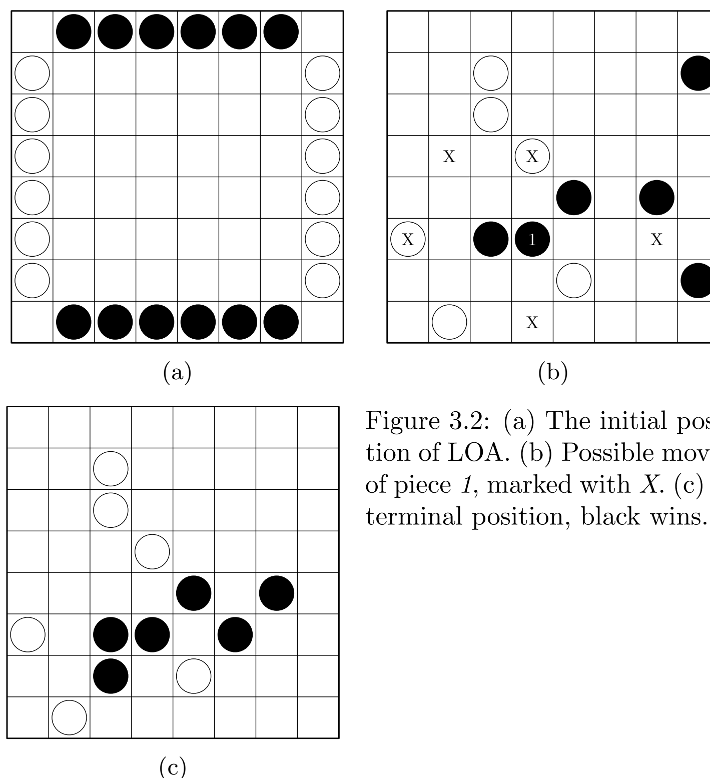

Figure 3.2: (a) The initial position of LOA. (b) Possible moves of piece *1*, marked with *X*. (c) A terminal position, black wins.

State of the Art. The state space complexity of LOA is 10^{23} positions. In general, positions have an average branching factor of 29 and the average game length is 44 moves. The game-tree complexity is estimated to be around 10^{64} (Winands, 2004). Different PNS variants have been applied to solve endgame positions in LOA, including PN^2, PN* and PDS-PN (Winands et al., 2002).

Engines for LOA have been designed that can play on an expert level (Winands et al., 2004a). The LOA engine MIA[1] uses a $\alpha\beta$ depth-first iterative-deepening search and Enhanced Realization Probability Search (Winands and Björnsson, 2007). At the time of writing, LOA is solved up to a board size of 6×6 (Winands, 2008).

3.3 Hex

Game Description. The game of Hex is a two-player game played on a rhombic board with hexagonal fields. Hex was developed by Hein (1942) and John Nash in 1948 (Nash, 1952). Hex is played on boards of various sizes, ranging from 6×6 to 19×19. Each of the players in Hex gets assigned opposing sides of the board, left-right and top-bottom. The goal of a player is to connect his sides with a consecutive line of pieces. The player with black stones tries to connect top to bottom, the player with white stones left to right. Pieces can be connected to a consecutive line if they share an edge. Players alternate in placing pieces on the board until one of them connects his sides and wins the game, this can be seen in Figure 3.3. Unlike most games, Hex is proven to have a winner, which means that there are no draws (Gale, 1979). Even though board sizes can be chosen freely, edges of the board should have the same length, otherwise there is an advantage for the player with sides of lesser length. Hex applies the pie rule, which can also be referred to as swap rule. The pie rule is applied in games where it can be shown that a first-move advantage exist. To reduce the impact of having the first move, the second player can choose the following, after the first move has been played:

1. Change places with the first player and continue the game from there
2. Play without any changes

Hex is ultra-weakly solved for any given board size.

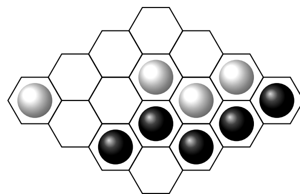

Figure 3.3: Example of a terminal position in Hex with black winning on a 4×4 board.

State of the Art. An approach to solving Hex is to solve openings of one move separately in order to solve an 8×8 board of Hex, one would have to

[1] https://dke.maastrichtuniversity.nl/m.winands/loa/

solve for 32 different openings, based on symmetry. In 2000, Hex was strongly solved on a 6×6 board (van Rijswijck, 2000) and later on an 8×8 board (Henderson et al., 2009). In 2013, all 9×9 openings have been solved with long computations (Pawlewicz and Hayward, 2013). A variant of Hex, Reverse Hex (Rex) (Gardner, 1957) has been used as an application for scalable parallel df-pn (Young and Hayward, 2016). The PNS variant FDFPN was introduced for Hex in (Henderson, 2010). It was later applied, using CNN estimations trained from games by experts (Gao et al., 2017). This suggests that learning from strong players can help for solving Hex positions. This resulted in solving all Hex openings on an 8×8 board with 46.7% fewer node expansions. Other work includes the use of df-pn for speed ups on search for boards up to a size of 9×9 (Arneson et al., 2011).

3.4 Othello

Game Description. Othello is a two-player game played on an 8×8 board. Two players alternate in placing black and white stones on the board. In the initial position, the 4 central fields are occupied with 2 pieces of each player. This is illustrated in Figure 3.4 (a). The player with black stones starts the game and places a piece on the board. Pieces are only allowed to be placed if the player would capture at least one piece of the opponent. A capture occurs if there are opponent pieces next to the placed piece and if there is a consecutive line of pieces in which lies between two pieces of the current player. This results in the captured pieces to change their color. If no such capturing move is available, the player passes his turn. Allowed moves in the starting position can be seen in Figure 3.4 (a). The game ends if the entire board is filled with pieces or both players pass consecutively. When the game is finished, the number of pieces of each color is being counted and the player with more pieces wins the game. The game results in a draw if both players have the same number of pieces. Figure 3.4 (b) illustrates a potential terminal position, where the black player wins.

State of the Art. The state space of Othello has been estimated to at most 10^{28} with a tree complexity of approximately 10^{58} (Allis, 1994). Othello has been used to test df-pn when it was first proposed by Nagai (1999). Later on, records of Othello games have been used to train df-pn+ (Nagai and Imai, 1999). The use of a mobility enhancement can be problematic in Othello, due to additional computations (Buro, 1997).

3.5 Connect6

Game Description. Connect6 is a connect-k type board game. In those, two players compete against each other in trying to connect k pieces in an orthogonal or diagonal without any empty fields in between. The first player to achieve this

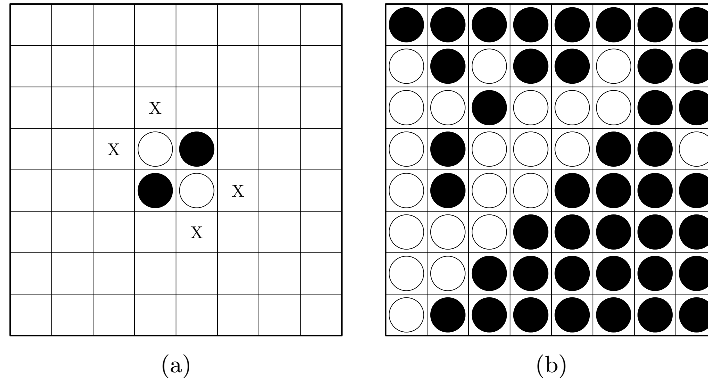

Figure 3.4: (a) The initial position of Othello, possible moves for black are marked with *X*. (b) Terminal position in Othello, black wins with 42 pieces over white with 22 pieces.

wins the game. In general, Connect6 is played on 19×19 boards, however any board size can be chosen with at least 6 rows and columns. A terminal position is illustrated in Figure 3.5 (b). In Connect6, players alternate in placing their stones on the board until the terminal condition of connecting 6 stones in a row is reached or the board is entirely filled with stones. If the board is filled with stones and neither of the two players connected 6 stones, the game ends in a draw. The game begins with the first player to place one stone on the board. In every following turn, two pieces are placed on empty fields. This strategy allows for a reduction of the first-move advantage. A possible position after the first 2 moves is shown in Figure 3.5 (a).

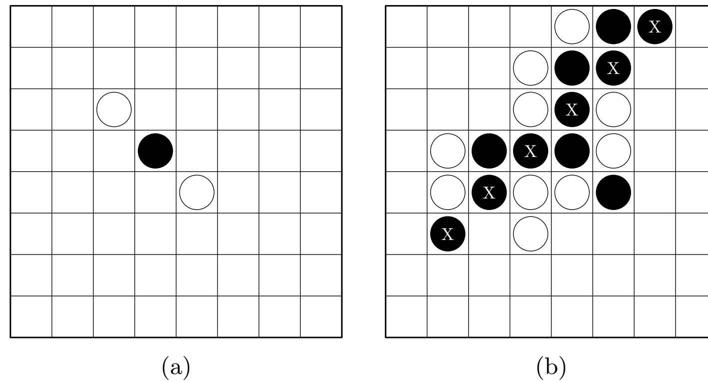

Figure 3.5: (a) Example opening of Connect6 on an 8×8 board. (b) Terminal position in Connect, black wins with 6 connected pieces, marked with *X*.

State of the Art. Connect6 is a game with a high complexity. The state space complexity is 10^{172}, almost as high as in Go (Wu et al., 2005). The search space complexity is 10^{140} (van den Herik et al., 2002), with an average game depth of 30 (Wu et al., 2005). Positions for solving Connect6 range from openings with few stones (Wu et al., 2011), openings on a 7×7 board (Gao and Xu, 2016), to endgame positions. Some approaches create their own positions to solve (Zhang et al., 2017; Xu et al., 2009), while others use publicly available Connect6 puzzles (Yen and Yang, 2010). Approaches to solve Connect6 positions include deep df-pn (Zhang et al., 2017) and job-level PNS (Wu et al., 2011). A common strategy when it comes to playing Connect6 is to apply threats (Wu and Huang, 2006; Xu et al., 2009; Yen and Yang, 2010). Furthermore, it is mentioned that initializing *pn* and *dpn* with unity is not the best choice when it comes to solving endgame positions (Xu et al., 2009).

Chapter 4

Machine Learning

This chapter outlines different concepts of machine learning. Firstly, an introduction to machine learning and different learning disciplines is given in Section 4.1. Artificial neural networks are explained in Section 4.2 and convolutional neural networks in Section 4.3. Section 4.4 concludes with an overview on the application of machine learning methods in games.

4.1 Introduction to Machine Learning

Machine learning is the study of models that learn to solve tasks without explicit commands and algorithms. Machine learning can be applied when problems are too complex to construct definite rules to solve them. Models are built using training data to make predictions on unseen data. Data in the training process can either be labeled or unlabeled. With labeled data, input data is matched with a corresponding output. Unlabeled data does not contain any output-information. Based on the type of training data, machine learning occurs in different forms:

Supervised. Labeled data exists. Given the input data, models are trained to replicate the output.

Semi-supervised. A small set of labeled data can be used in combination with a large set of unlabeled data. Labeled data is used to predict labels of unlabeled data to improve the performance of the model by utilizing more data.

Unsupervised. Only data without labels is used for training. An application for unsupervised learning is clustering. With easily available processing power, larger machine learning models became more popular, introducing deep learning as a subfield of machine learning.

4.1.1 Reinforcement Learning

Reinforcement learning is used to train an agent with desired behaviour by giving rewards (Kaelbling et al., 1996). The goal of an agent is to behave accordingly, to maximize its rewards. Problems are reformulated to an environment with observable states. States can be changed by performing actions. Rewards are associated with the combination of states and actions performed. In order to solve reinforcement learning problems, one can attempt to solve it with known actions and behavioural patterns. Another approach is to estimate the reward of unknown states and actions to solve problems. Depending on the environment, agents are either able to explore it themselves to generate training data or the training data is provided beforehand.

4.1.2 Tasks

Machine learning methods can be applied to several tasks. To solve those optimally, a proper output function has to be chosen. A distinction can be made between the following tasks:

Regression. Regression models learn a function that approximates a real valued prediction.

Classification. Given a set of classes, predictions are made on which class a data sample belongs to. Often, a Softmax (Bishop, 2006) is applied to normalize all class estimates within the range $[0, 1]$, summing to 1.

Preference. Another task is to consider preferences. In a preference ranking model, preference relations are being predicted (Fürnkranz and Hüllermeier, 2010). This invokes the prediction of more complex structures and allows for an increased flexibility.

4.2 Artificial Neural Network

Artificial neural networks (ANN) are models that estimate non-linear functions. ANNs can be compared to the human brain, as the brain consists of a multitude of connected neurons and ANNs of connected nodes. Nodes are structured in layers which are connected. An input is parsed to a layer, each node computes an output value and passes it to the next layer. This process is followed until the output layer is reached and final predictions are made. A visualization of connected layers can be seen in Figure 4.1. Every layer that is neither an input or output layer is called hidden layer. Multiple hidden layers can be used in a network. The computation in a node can be described as follows:

$$f(\sum x_i \times w_i) \tag{4.1}$$

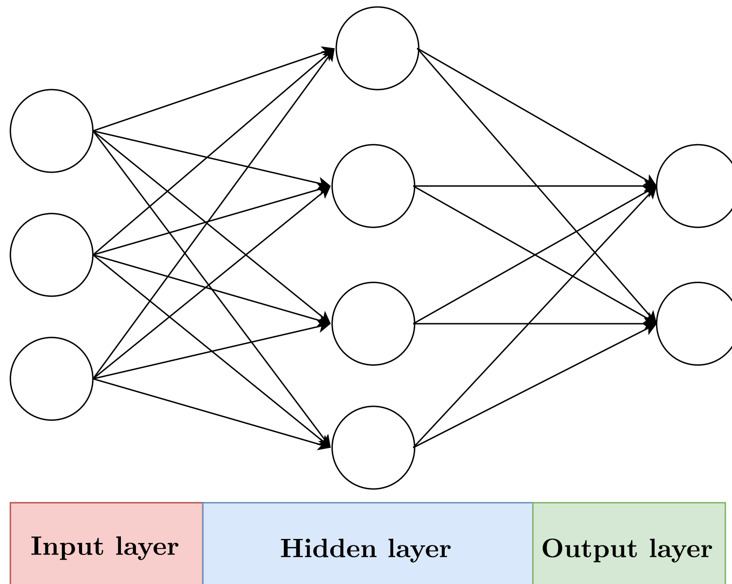

Figure 4.1: Example of an artificial neural network with 1 hidden layer.

Where f is an activation function, x is input passed on from nodes in the previous layer and w represents the weight of each input. A single valued output value is created per node. By following this procedure, any function can be approximated. The approximation is done by tuning weights of the ANN in order to reconstruct outputs given training input data.

4.3 Convolutional Neural Network

Convolutional neural networks (CNN) are a type of ANN, primarily applied when dealing with pattern recognition in image related tasks. The CNN architecture allows to efficiently deal with images, multi-dimensional input vectors, whereas ANNs would require a multitude of nodes and weights (O'Shea and Nash, 2015). To deal with and process large input data, CNNs alternate between convolutions and subsampling (LeCun and Bengio, 1998).

4.3.1 Convolution

Convolution determines the activation of neurons. Kernels are applied to local regions of the input by applying scalar multiplications. Kernels are applied in a spatial domain, shifted over the entirety of the input. Applying convolution to an input can end with a result with reduced size. Padding techniques can be applied to extend the output to the input size. An example for a single convolution is shown in Figure 4.2.

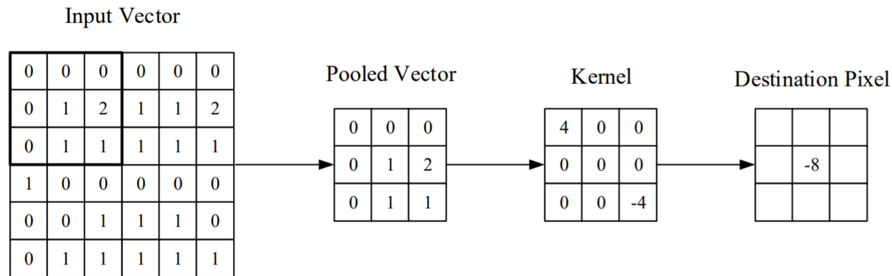

Figure 4.2: Visualization of the convolution process. A kernel is placed over the input vector and applies a weighted sum. Taken from O'Shea and Nash (2015).

4.3.2 Subsampling (Pooling)

Pooling is a subsampling technique to reduce the dimensionality of data. The most common pooling technique is max pooling (O'Shea and Nash, 2015). Given an excerpt of data points, they will be reduced to the maximum value among those. This has the effect of reducing the complexity of proceeding computations. Furthermore, small details in the data are negligible and can be removed by pooling.

4.4 Learning in Game Playing

A multitude of learning techniques have been introduced, which can be applied in various domains, including game playing. A selection of learning techniques applied to board games is given below, to enable an understanding of the opportunities learning has in board games. Reinforcement learning has been applied to multiple games such as Othello (van der Ree and Wiering, 2013), Connect4 and Tic-Tac-Toe (Imran, 2004). Reinforcement learning models can be trained from game records or by self-play. Runarsson and Lucas (2014) apply preference learning in Othello, using game records for training. A combination of reinforcement learning and neural networks can be seen in deep reinforcement learning, which has been applied to games such as Hex (Young et al., 2016). A comprehensive example for applying learning techniques in board games can be found in AlphaGo (Silver et al., 2016). Deep reinforcement learning has been performed with CNNs, using expert games for training. Additionally, training data has been created from self-play. Later on, trained networks have been used to predict values as a regression task, to guide MCTS.

Chapter 5

Proof and Disproof Number Estimation

In this chapter, three enhancements for PNS are introduced and investigated in their ability to estimate *pn* and *dpn*. Section 5.1 outlines two enhancement based on moves from expert games. Section 5.2 introduces a deep learning model to predict *pn* and *dpn* based on board positions. The objective is to create enhancements that can be used independent of game domains.

5.1 Move Frequency and Availability

The first approach for estimating *pn* and *dpn* is to analyze expert games for common moves that are played over multiple games. Assuming that a move is played in every game would lead to the conclusion that it is important for winning games and should therefore receive a lower *pn* or *dpn*. Likewise, if a move is rarely played, one can assume that it is not good leading to a higher estimate. Another interpretation is to consider how frequent moves have been played, given that they are available. The goal of the second approach is to assess less frequent moves more accurately.

5.1.1 Data

Game notations of high-level play are analyzed, this includes either expert games or games of engines. Games are processed, keeping track of each move that has been played. This results in a mapping of moves to their frequency of being played in games (Formula 5.1), or their availability (Formula 5.2). Moves are scored according to their frequency:

$$score_{frequency} = f/n \tag{5.1}$$

$$score_{availablity} = f/a \tag{5.2}$$

f denotes the frequency of a move occuring, n the total number of games and a the total number of positions in which a move is available. Scores are ceiled at 1. Each score is in a range of $[0 - 1]$, with frequent moves having a higher score. High scores should lead to lower pn and dpn estimates, as all estimates should be ≥ 1. A method to transform scores to pn and dpn is as follows:

$$estimate = 1/score \tag{5.3}$$

In order to further weight moves, a hyper-parameter k is introduced:

$$estimate = 1/score^k \tag{5.4}$$

Unseen moves are assigned with an estimate of $0.5/n$ when considering frequency and an estimate of 1 when considering availability. As positions that are supposed to be solved are rarely found within the first few moves (opening), one could skip opening moves for computing scores.

5.1.2 Implementation

This method is used to initialize pn and dpn as follows: OR nodes are initialized with $pn = 1; dpn = 1/score^k$, AND nodes are initialized to $pn = 1/score^k; dpn = 1$. Therefore, each node has to store the last executed move, which is used to determine the score. All nodes except for the root are initialized in this way.

5.2 Deep Learning

Deep learning with CNNs can be applied to board positions in games in order to make predictions. In the following, the use of CNNs to determine pn and dpn is evaluated.

5.2.1 Data

The same set of expert games from Section 5.1 is used to generate training data for a CNN. Each game is processed and the result at the end of the game is determined. This data alone allows for the prediction of the result of a board position. In order to ascertain realistic pn and dpn it is not only of interest who wins the game, but also how many moves are required to reach the end of the game. This information can be added to every board position by subtracting the current move from the total number of moves of a game. Given these estimates, the potential number of nodes required to solve a subtree can be estimated. Therefore, tuples of the following form are extracted from games:

$$(board, moves, winner)$$

Frequently, games carry characteristics of symmetry, which allows for data augmentation techniques applied to generate more data. An overview of the applied

augmentation techniques per game can be seen in Appendix C. After augmentation, the available data is described in Table 5.1.

	Games	Black wins	White wins	Draw	Unique tuples
LOA	32,672	19,508	11,720	1,444	1,392,894
Hex	56,860	44,730	12,130	0	305,458
Othello	124,219	56,233	60,599	7,387	1,209,773
Connect6	30,219	13,709	16,510	0	156,242

Table 5.1: Description of training data for deep learning after applying augmentation.

It is noted that, although draws are possible in Connect6, they do not occur in any of the recorded games. Therefore, the outcome of Hex and Connect6 games predicts either a win for black or white, whereas a win for black or white or a draw is predicted for LOA and Othello.

5.2.2 Input

Each board position is preprocessed to extract three feature layers, which are fed into the CNN. Similar to Silver et al. (2016), a one-hot encoding of layers is used. The following layers are being extracted:

1. Black: Layer with 1 for black pieces and 0 otherwise
2. White: Layer with 1 for white pieces and 0 otherwise
3. Empty: Layer with 1 for empty spaces and 0 otherwise

Preprocessing is applied to 8×8 and 19×19 boards in the same manner. This resembles a minimum of information that can be extracted from a game, which allows this approach to be applied to various different game domains. Other approaches include information about the current player to move (Silver et al., 2017; Liskowski et al., 2017; Hlynur Daví, 2017; Gao et al., 2018), which was disregarded in order to keep the network as simple as possible.

5.2.3 Core

Following model is inspired by Gao et al. (2018) used for Hex and Liskowski et al. (2017) for Othello. The input is processed by various, consecutive convolutional layers, that all apply a ReLU[1] activation function. The following four architectures are trained for games with a board size of 8×8:

[1]Rectified Linear Unit: $f(x) = max(0, x)$

NW1	128 → 128 → 128 → 128 → 128 → 128 → 128
NW2	128 → 128 → 128 → 128 → 128 → 128 → 128 → 128
NW3	64 → 64 → 128 → 128 → 256 → 256
NW4	64 → 64 → 128 → 128 → 256 → 256 → 256 → 256

Table 5.2: Description of 4 network models for 8×8 board games. Nodes of consecutive convolutional layers are given.

For a board size of 8×8, filters of the size $(3, 3)$ and a stride of 1 are used. CNNs have not been applied to Connect6 before, nonetheless CNNs have been applied to Go with 19×19 boards. A characteristic that can be seen in Barratt and Pan (2017), is the use of larger filter sizes. These four architectures for 19×19 boards are trained:

NW1	128(5) → 128(5) → 128(5) → 128(3) → 128(3) → 128(3) → 128(3)
NW2	128(7) → 128(5) → 128(5) → 128(3) → 128(3) → 128(3) → 128(3)
NW3	64(7) → 64(5) → 128(5) → 128(5) → 256(3) → 256(3) → 256(3) → 256(3)
NW4	64(5) → 64(5) → 128(5) → 128(5) → 256(3) → 256(3) → 256(3) → 256(3)

Table 5.3: Description of 4 network models for 19×19 board games. Nodes of consecutive convolutional layers are given with filter size in braces.

As different sizes of layers are used, the respective filter size for each layer is denoted in brackets. After each convolutional layer, zero-padding is applied to remain with data of the same dimensionality. The dimensionality of the input is small, which gives no reason to apply any pooling in the network (Springenberg et al., 2014). Every convolutional layer applies a RelU activation function.

5.2.4 Output

The CNN is used to generate two outputs. Firstly, the outcome of the game is predicted. This is either a binary classification, if no draws occur in the game domain, or tertiary, if draws are possible. Secondly, the number of moves required to reach a terminal position is predicted. This two-fold output carries similarities to the network in Silver et al. (2017), where outcome and a scoring of moves is predicted. Table 5.4 lists the appended layers to predict the *outcome* and the required number of *moves* until the game is finished: A complete network model is illustrated in Figure 5.1.

5.2.5 Training

The network is trained with a batch size of 256 over 20 epochs. Stochastic gradient descent is applied for optimization, with a learning rate of 0.1. Both

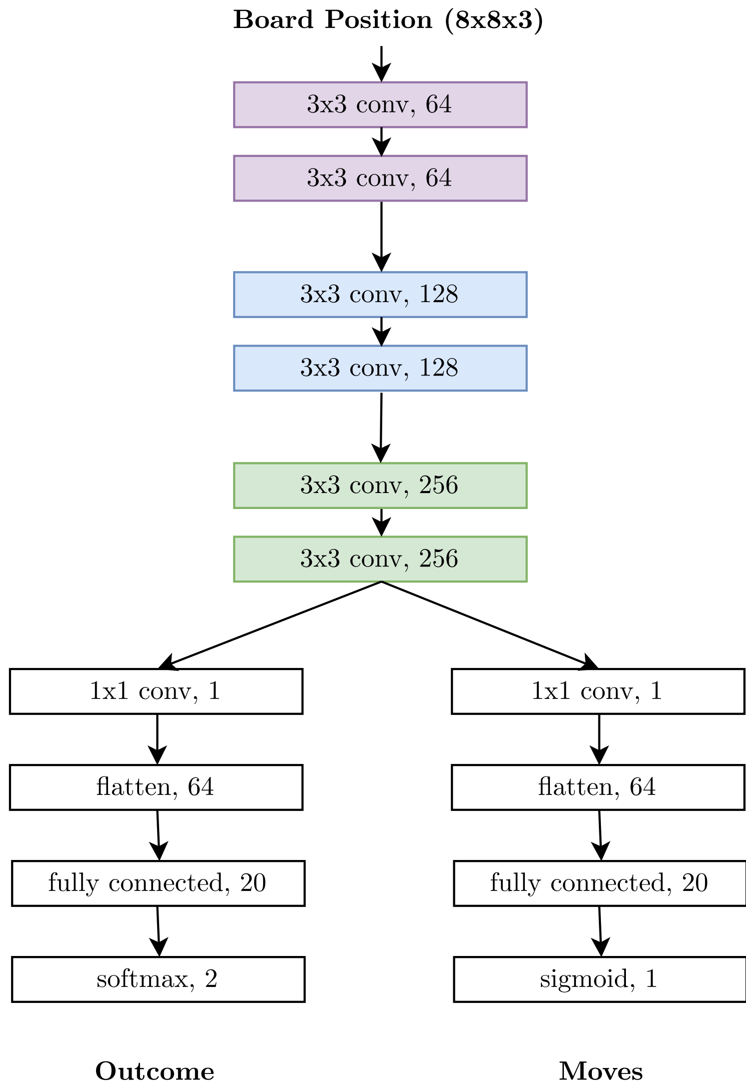

Figure 5.1: Example of a network model for 8×8 board games. Game has two results (win and loss).

Outcome	Moves
Convolution (1,1)	Convolution (1,1)
Flatten	Flatten
Dense (20 Nodes, RelU)	Dense (20 Nodes, RelU)
Softmax (2 or 3 Nodes)	Dense (1 Node, Sigmoid)

Table 5.4: Descriptions of Layer that predict Outcome and Number of Moves until a terminal position is reached.

outputs, *outcome* and *distance*, are weighted with 50% to determine the loss of an epoch. Mean squared error is applied to determine the *distance* loss; categorical cross-entropy is applied to determine the *outcome* loss. 90% of the data is used for training and 10% for testing. Models are trained on a single NVIDIA Tesla P100 SXM2 with 32 Gb of memory.

Chapter 6

Experiments

In this chapter, several experiments are conducted. At first, the experimental setup is explained in Section 6.1, followed by an outline of the data used in Section 6.2. This includes a selection of expert games and positions for each of the four game domains. Following, a multitude of experiments are conducted. Firstly, the base performance of PNS variants is evaluated (Section 6.3) followed by an evaluation of the move frequency and availability enhancements (Section 6.4). The training process of deep learning models is described in Section 6.5, where the best performing model is selected and applied to positions (Section 6.6). An application of enhancements on LOA with a smaller board size, of 5×5 is performed in Section 6.7. Lastly, Section 6.8 discusses results.

6.1 Experimental Setup

Experiments are conducted on the games LOA, Hex, Othello and Connect6. The following PNS variants are applied to each game:

- PNS
- PNS + mobility
- PN*
- PDS
- df-pn
- PN^2
- PDS-PN
- dfpn-pn

Hereby, PNS with mobility enhancement is only applied to LOA due to the branching factor of Othello, Hex and Connect6. For PNS, solved subtrees are deleted. Both PDS and df-pn are applied with ϵ-trick. Corresponding to Pawlewicz and Lew (2007), $\epsilon = 1/16$ in PDS and $\epsilon = 1/4$ in df-pn. Transposition tables are used with a Two-Big replacement scheme; hashes are created with Zobrist hashing (Zobrist, 1970). In two-level searches, the size of the second level will be set to $min(total_{memory} - size_{first\text{-}level}, size_{first\text{-}level} + 1)$. This ensures that a minimum of two nodes are explored at every second level evaluation. Therefore, in every case, the second level root is extended with children. Finally, memory constraints are set to keep a maximum of 1 million nodes in memory and a maximum of 500 million node evaluations are allowed to be performed.

6.2 Data

In the following the gathering of data is outlined for each game domain. This includes end game positions as well as notations of played games. The game notations are later used to extract information about the frequency of played moves and to train deep learning models.

6.2.1 Lines of Action

Positions. A set of 488 positions was gathered from Mark Winands' homepage[1]. This set includes tactical endgame positions that have been used to test endgame solvers.

Games. A set of 8,168 LOA games is used. Games were created with two engines playing against each other. Both engines apply MCTS (Winands and Björnsson, 2010).

6.2.2 Hex

Positions. Solving algorithms for Hex are applied to the starting position. On an empty board, every first move is being analyzed. When choosing a board size, two factors have to be considered. Firstly, the board has to be large enough to be challenging to solve. Secondly, the board has to be small enough to be solvable within reasonable time and memory constraints. For this purpose, a board size of 8×8 is chosen. Previous work solved some openings, using df-pn, with less than 50,000 nodes in memory (Gao et al., 2017). 32 out of the 64 openings are analyzed. In the case that openings are too complex too solved within constraints of the experiments, a set of 20 endgame positions has been created. These positions are verified to be solvable by at least one of the PNS variants. The endgame positions are described in Appendix B.

[1] https://dke.maastrichtuniversity.nl/m.winands/loa/

Games. A set of games, used in Gao et al. (2017) for an 8×8 Hex board, is used. Games have been created with an engine that has played against itself. This set includes a total of $50,795$ games.

6.2.3 Othello

Positions. A set of 20 Othello puzzles made by Marc Tastet, former Othello world champion in 1992, can be found online.[2] For every puzzle, the best result is known, however the player to move is not winning in all of the positions. For experiments, it is not intended to find the best score, but only the winning side of a position.

Games. Games have been obtained from the French Othello Association.[3] This includes a total of $124,219$ game records, played from 1977 to 2018. Game records can be retrieved in the *wthor* format; a Python reader is provided on Github.[4]

6.2.4 Connect6

Positions. The Taiwanese Connect6 association[5] provides a collection of puzzles online. Yen and Yang (2010b) covered 30 problems from the year 2008. The set of 30 problems is used for experiments. It is noted that a total of 201 puzzles are available by the time of writing.

Games. Up to this point, no set of Connect6 games was publicly available, however well performing engines can be found. In order to create games in self-play, the engine *Cloudict*[6], first place in the 16th Computer Olympiad 2011, is used. *Cloudict* uses an alpha-beta search with further enhancements. All enhancements are being used and a search depth of 5 is applied. As alpha-beta search is deterministic, every played game would look identical. To prevent this, a random factor is added to the evaluation of positions. Every evaluation will be multiplied with a random factor in the range $[0.5, 2]$. In total, $30,219$ games have been created in this fashion.

[2]http://www.radagast.se/othello/ffotest.html
[3]http://www.ffothello.org/
[4]https://github.com/maxhort/WTB_reader
[5]http://www.connect6.org/
[6]https://github.com/lang010/cloudict

6.3 Base Performance

Lines of Action. In Table 6.1, it can be seen that timeouts occur for PDS, df-pn and dfpn-pn. Timeouts happen if the execution time for solving a position exceeds a day. This leads to the position to be considered as not solved. In addition to the number of positions solved by each variant, two sets of comparisons are conducted. Firstly, one-level searches are compared on 81 positions, which variants solve in common. Among those, PNS with mobility enhancements is able to solve the positions the fastest, while df-pn requires the smallest number of evaluations. Two-level search variants are compared on 394, which each variant is able to solve. Among two-level variants, PN^2 requires the fewest number of nodes, followed by dfpn-pn. Despite a fewer number of nodes evaluated, dfpn-pn requires the shortest time to solve all positions. PDS-PN is performing worst among the three variants. PDS is significantly outperforming two-level variants, reducing the number of evaluated nodes by a factor greater than 40 and execution time by a factor of approximately 4.

			81 positions		394 positions	
Variant	Solved	Timeouts	Nodes	Time in s	Nodes	Time in s
PNS	161	0	3,247,204	1,259	-	-
PNS + mobility	366	0	349,940	432	-	-
PN*	33	0	-	-	-	-
PDS	415	4	349,193	1,215	16,624,894	59,090
df-pn	100	332	309,696	740	-	-
PN^2	459	0	-	-	714,908,878	217,019
PDS-PN	459	0	-	-	1,091,230,642	230,847
dfpn-pn	437	30	-	-	829,789,778	186,230

Table 6.1: Comparison of nodes evaluated and execution time for PNS variants on LOA positions.

A closer investigation on how variational and spread the number of nodes required to solve positions by the variants PDS, PN^2, PDS-PN and dfpn-pn, is done on the same set of 394 positions. Table 6.2 indicates that there is a correlation between the total number of nodes required to solve all positions and the standard deviation. As with nodes required, PDS has the lowest standard deviation followed by PN^2, dfpn-pn and PDS-PN. Based on this, one can assume that PDS-PN requires more nodes to solve the same set of positions, because it is more variational in its performance.

Hex. At first, PNS variants have been applied on 32 openings for Hex on 8×8 boards. However, no variant was able to sole any of the openings under the given constraints. Therefore, 20 self-created positions are evaluated.

Variant	Nodes	Time in s	Standard Deviation
PDS	16,624,894	59,090	65,193
PN^2	714,908,878	217,019	6,469,113
PDS-PN	1,091,230,642	230,847	8,318,599
dfpn-pn	829,789,778	186,230	6,978,976

Table 6.2: Comparison of standard deviation for a selection of PNS variants.

		10 positions		14 positions	
Variant	Solved	Nodes	Time in s	Nodes	Time in s
PNS	10	722,373	267	-	-
PN*	10	291,352	409	-	-
PDS	19	21,192	50	174,067	783
df-pn	17	23,813	57	341,059	1,474
PN^2	20	-	-	30,105,022	9,130
PDS-PN	18	-	-	42,329,978	8,224
dfpn-pn	16	-	-	43,118,777	7,660

Table 6.3: Comparison of nodes evaluated and execution time for PNS variants on Hex positions.

Table 6.3 shows a comparison of the performance of PNS variants on 20 positions. PN^2 is the only variant to solve all positions. One-level searches solve 10 positions in common. Among those variants, PNS and df-pn outperform PNS and PN* by a large margin, while they only slightly deviate in terms of nodes evaluated and time required. For two-level searches, PN^2 requires approximately 25% less nodes than PDS-PN and dfpn-pn. However, the fastest two-level variant is dfpn-pn. As can be seen in the comparison of two-level variants, one-level variants solve the same set of positions quicker.

Othello. As can be seen in Table 6.4, only 5 positions from 20 can be solved at all by two-level variants and only 2 positions by PNS and PN*. A comparison of PNS variants in Table 6.4 shows that dfpn-pn is outperforming other two-level search variants. PNS performs slightly better than PN*, however the problems that are being solved are too small, to draw any further conclusions.

Connect6. None of the 30 Connect6 problems could be solved with the given constraints. A further investigation of PNS variants on Connect6 is omitted.

		2 positions		5 positions	
Variant	Solved	Nodes	Time in s	Nodes	Time in s
PNS	2	394	0.148781	-	-
PN*	2	181	1.368938	-	-
PDS	0	-	-	-	-
df-pn	0	-	-	-	-
PN^2	5	-	-	47,552,042	17,363
PDS-PN	5	-	-	43,529,984	19,984
dfpn-pn	5	-	-	35,120,747	11,760

Table 6.4: Comparison of nodes evaluated and execution time for PNS variants on Othello positions.

6.4 Move Frequency and Availability

In the following, the move frequency and availability enhancement proposed in Section 5.1 is applied to LOA, Hex and Othello. For this purpose, PNS is used as a base algorithm and enhanced to initialize *pn* and *dpn*,

Lines of Action. In Table 6.5, it can be seen that both the frequency and availability enhancements achieve substantial reductions for nodes evaluated and execution time. The biggest improvement is achieved by the availability enhancement with $k = 1$ and skipping the first 10 moves. This reduces the number of nodes evaluated and execution time by almost 90%. In general, the availability enhancement performs better than the frequency enhancement. Skipping the first 10 moves leads to small improvements.

		51 positions		313 positions	
Variant	Solved	Nodes	Time in s	Nodes	Time in s
PNS	161	2,037,305	746	-	-
PNS + mobility	366	311,292	346	8,003,300	9,693
PNS + frequency (k=1)	329	406,717	163	11,900,519	5,223
PNS + frequency (k=2)	132	1,346,887	640	-	-
PNS + frequency10 (k=1)	321	422,280	162	10,566,240	3,906
PNS + frequency10 (k=2)	88	1,796,465	715	-	-
PNS + availability (k=1)	358	266,206	109	8,638,133	3,178
PNS + availability (k=2)	149	1,751,443	669	-	-
PNS + availability10 (k=1)	357	263,229	98	8,074,799	3,165
PNS + availability10 (k=2)	199	1,046,780	428	-	-

Table 6.5: Comparison of nodes evaluated and execution time for move frequency and availability enhancements on LOA positions.

Variant	Nodes	Time in s	Standard Deviation
PNS + mobility	8,003,300	9,693	38,806
PNS + frequency (k=1)	11,900,519	5,223	40,641
PNS + frequency10 (k=1)	10,566,240	3,906	36,791
PNS + availability (k=1)	8,638,133	3,178	37,085
PNS + availability (k=10)	8,074,799	3,165	35,471

Table 6.6: Comparison of standard deviation for PNS with enhancements.

The best performing variants from Table 6.5, that have been compared on 313 positions, are analyzed for their standard deviation. Table 6.6 shows the standard deviations of the 5 variants. Unlike the previous investigation on standard deviations in Table 6.2, the order of magnitude of standard deviations is unlike the order of magnitude of nodes required. The highest standard deviation is achieved by PNS with frequency enhancement followed by PNS with mobility enhancement. Based on nodes required alone, PNS with mobility enhancement is performing the best. Even though PNS with mobility enhancement performs with the least number of nodes required, it is impacted by a high variance. This most likely shows that there is a subset of positions which PNS with mobility enhancement can solve easily, while other variants cannot. The lowest standard deviation is achieved by PNS with availability enhancement, skipping the first 10 moves. Skipping opening moves may indicate a reduction in standard deviation, as it is the case for frequency and availability enhancement.

Hex. Table 6.7 shows a comparison of the frequency and availability enhancement on Hex. It can be seen that skipping the first 10 moves leads to slight improvements over considering all moves. The best performance is achieved by the frequency enhancement with $k = 2$ and skipping the first 10 moves. Due to the reason that every move in Hex is available, as long it has not been played already, the availability enhancements transforms to an evaluation of how early moves are being played.

Othello. Table 6.8 shows that no improvements are achieved by the availability enhancement. The frequency enhancement was not applied, as it is common for Othello, that every move is played once in a game, therefore not leading to a deviation in scores for moves.

		9 positions	
Variant	Solved	Nodes	Time in s
PNS	10	693,949	258
PNS + frequency (k=1)	10	323,390	129
PNS + frequency (k=2)	10	162,608	77
PNS + frequency10 (k=1)	10	208,206	78
PNS + frequency10 (k=2)	10	92,597	35
PNS + availability (k=1)	13	156,392	54
PNS + availability (k=2)	5	-	-
PNS + availability10 (k=1)	11	143,657	56
PNS + availability10 (k=2)	5	-	-

Table 6.7: Comparison of nodes evaluated and execution time for move frequency and availability enhancements on Hex positions.

		2 positions	
Variant	Solved	Nodes	Time
PNS	2	394	1
PNS + availability (k=1)	2	10,220	9
PNS + availability (k=2)	2	150,704	93
PNS + availability10 (k=1)	2	47,210	37
PNS + availability10 (k=2)	2	173,764	106

Table 6.8: Comparison of nodes evaluated and execution time for move frequency and availability enhancements on Othello positions.

6.5 Deep Learning - Training

In order to assess the performance of training networks on different games, Figure 6.1 compares the training loss of the four network architectures, proposed in Section 5.2, on each game. The performance of each network is evaluated on the test set. The best performing network for LOA, Hex, Othello and Connect6 are $(NW2, NW1, NW1, NW4)$ respectively. A detailed visualization of the loss of both outputs on these networks is given in Appendix D. The training loss for LOA, Hex and Othello remains fairly similar among all network architectures, while they differ substantially for Connect6. It can be seen that networks with a filter size of $(7, 7)$ (NW2 and NW3) are outperformed by networks with a smaller filter size of $(5, 5)$. Furthermore, the overall loss of Hex and Connect6, where no draw is possible, have a lower overall loss than Othello and LOA. A reason for this can be seen in the fact that an output has to be predicted among three options (win,loss,draw) rather than two. This is supported by Appendix D, which shows that networks are able to predict the number of moves, until a terminal position is reached, very accurately, while the outcome is prone to more inaccuracies.

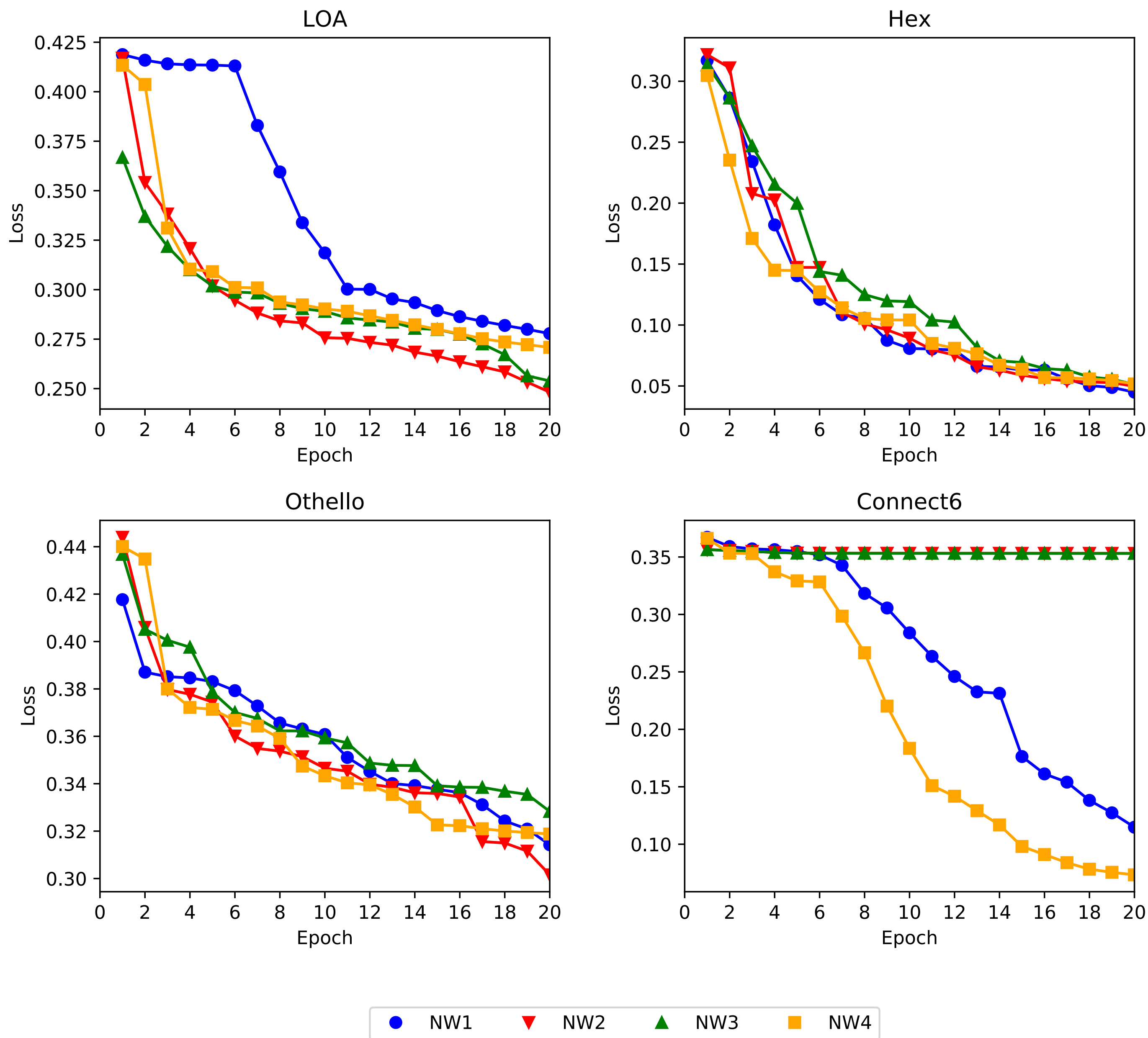

Figure 6.1: Training loss of four different network architectures on each game.

6.6 Deep Learning - Enhancement Performance

The best performing network found in Section 6.5 is used to initialize *pn* and *dpn* on LOA, Hex and Othello positions. Therefore, the prediction output of *moves* and *outcome* is used to compute an estimate for *pn* and *dpn*. As for the mobility enhancement, the *dpn* is set to this estimate at OR nodes, and the *pn* is set to this estimate at AND nodes. Formula 6.1 is used to calculate estimates, where a, b, c are hyper-parameters that can be chosen.

$$estimate = a \times moves^{b} \times (1/winner^{c}) \tag{6.1}$$

The notion of *winner* is gathered from the *outcome* prediction. Assuming that the first player is winning a given position, *winner* is equal to the prediction that the outcome of the first player to win occurs.

Lines of Action. Table 6.9 shows that the deep learning enhancement can be used to reduce the number of nodes evaluated, in contrast to plain PNS. However, due to the amount of computations required to make a prediction, the computation time increases at minimum by a factor of 40. The best performance is achieved with the setting: $a = 2; b = 1; c = 1$. This reduces the number of nodes evaluated by more than 80% in comparison to PNS and almost by 50% in comparison to all other configurations of the deep learning enhancement. As the deep learning enhancement reduces the number of nodes required, it is compared to the frequency and availability enhancement in an upcoming section.

					114 positions	
Variant	a	b	c	Solved	Nodes	Time in s
PNS	-	-	-	161	6,310,336	2,355
PNS + Deep Learning	1	0	1	165	5,828,078	265,118
PNS + Deep Learning	1	1	0	327	1,537,682	67,409
PNS + Deep Learning	1	1	0.5	322	1,499,188	90,443
PNS + Deep Learning	1	1	1	307	1,486,971	65,458
PNS + Deep Learning	1	1	2	297	1,455,198	71,229
PNS + Deep Learning	1	2	2	240	4,502,965	253,919
PNS + Deep Learning	2	1	1	343	890,299	45,926

Table 6.9: Comparison of nodes evaluated and execution time in seconds for deep learning enhancements on LOA positions.

Hex. Similarly to LOA, all configurations of the deep learning enhancement achieve reductions in the number of nodes evaluated over plain PNS (Table 6.10). The largest reduction is achieved with a setting of $a = 2; b = 1; c = 1$, reducing by more than 50%. As with LOA, the computation time of the deep

learning enhancement is severely higher than without. A comparison to the frequency and availability enhancement is performed in an upcoming section.

					8 positions	
Variant	a	b	c	Solved	Nodes	Time in s
PNS	-	-	-	10	604,877	227
PNS + Deep Learning	1	0	1	10	386,306	13,749
PNS + Deep Learning	1	1	0	9	439,625	19,267
PNS + Deep Learning	1	1	0.5	9	453,902	16,124
PNS + Deep Learning	1	1	1	9	477,126	16,804
PNS + Deep Learning	1	1	2	10	474,692	17,021
PNS + Deep Learning	1	2	2	9	552,179	19,837
PNS + Deep Learning	2	1	1	9	241,723	8,701

Table 6.10: Comparison of nodes evaluated and execution time in seconds for deep learning enhancements on Hex positions.

Othello. As for the availability enhancement, no improvements have been achieved with the deep learning enhancement in Othello, as can be seen in Table 6.11.

					1 Problem	
Variant	a	b	c	Solved	Nodes	Time in s
PNS	-	-	-	2	251	0.1
PNS + Deep Learning	1	0	1	0	-	-
PNS + Deep Learning	1	1	0	0	-	-
PNS + Deep Learning	1	1	0.5	0	-	-
PNS + Deep Learning	1	1	1	0	-	-
PNS + Deep Learning	1	1	2	0	-	-
PNS + Deep Learning	1	2	2	0	-	-
PNS + Deep Learning	2	1	1	1	12,319	1,192

Table 6.11: Comparison of nodes evaluated and execution time in seconds for deep learning enhancements on Othello positions.

6.7 Application to 5×5 Lines of Action

Enhancements such as initializing *pn* and *dpn* with mobility are easily transferable to different games and same games with different board sizes. The reason for this is that the notion of mobility is adapted by the game. The question remains whether the move frequency/availability and deep learning enhancements are transferable. This assumption is tested on LOA with a 5×5 board, and the use of enhancements trained on 8×8 games is evaluated.

LOA games on 8×8 boards are used to extract moves that could have been played on a 5×5 board. Therefore, the board is divided into four areas of the size 5×5 (see Figure 6.2). Every move that is played on the 8×8 board, where the starting and end field of the moving piece remain in the same area, is translated to a move on a 5×5 board. Experiments with PN^2 to solve 5×5 LOA, show that the mobility enhancement solves with the fewest nodes evaluated. The frequency enhancement achieves reductions of nodes evaluated of up to 50%, while it remains slower than the mobility enhancement. The availability enhancement achieves even further reduction, up to around 75% in contrast to plain PN^2. This does not outperform the mobility enhancement in terms of evaluated nodes, but reduces the computation time by 40%. Additionally, a few experiments have been conducted with the deep learning enhancement, however none could achieve mentionable improvements over PN^2. While the frequency and availability enhancement achieved improvements on 5×5 LOA, deep learning enhancement could not achieve significant improvement over a plain PN^2 search. A reason for this can be seen in the board positions, as it is unlikely to achieve a position, such as the starting position in 5×5 LOA at any given time in 8×8 LOA. This results in a lack of training data for this particular case, and predictions are made with a lack of knowledge. The winning move found for 5×5 LOA is $B5 - D2$.

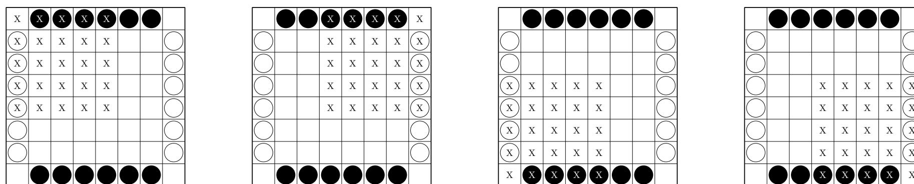

Figure 6.2: Division of an 8×8 LOA board into four 5×5 boards.

Variant	Nodes	Time in s
PN^2	7,823,893	225.08
PN^2 + mobility	1,331,810	106.63
PN^2 + frequency (k=0.5)	5,197,143	163.69
PN^2 + frequency (k=1)	3,789,996	113.59
PN^2 + frequency (k=2)	5,188,762	180.25
PN^2 + availability (k=0.5)	2,010,575	62.97
PN^2 + availability (k=1)	2,893,053	88.39
PN^2 + availability (k=2)	7,235,441	257.05

Table 6.12: Comparison of PN^2 to solve LOA on a 5×5 board.

6.8 Results and Discussion

6.8.1 Lines of Action and Hex

The set LOA and Hex positions have proven to be suitable to use the frequency/availability as well as deep learning enhancement. Therefore, a further comparison of all enhancement among each other is conducted.

Lines of Action. Table 6.13 shows that the deep learning enhancement is not able to reduce the number of nodes evaluated of the availability enhancement and does so for the frequency enhancement. Overall, the availability enhancement is solving 301 commonly solved positions the fastest, while requiring almost as little nodes as the mobility enhancement.

		152 positions		301 positions	
Variant	Solved	Nodes	Time in s	Nodes	Time in s
PNS	161	9,166,271	3,414	-	-
PNS + Mobility	366	1,225,184	1,417	7,554,739	9,163
PNS + Frequency10 (k=1)	326	2,087,844	776	9,967,591	3,675
PNS + Availability10 (k=1)	357	1,289,007	509	7,585,166	2,962
PNS + Deep Learning (a=2,b=1,c=1)	343	1,649,471	82,161	9,871,856	487,400

Table 6.13: Comparison of nodes evaluated and execution time for search enhancements on LOA positions.

Hex. The deep learning enhancement in Hex is not able to outperform the frequency and availability enhancement in any of the measures. The number of nodes evaluated is larger by a factor of $2-3$, while the computation time to solve the same set of 9 problems is almost 300 times as large.

		9 positions	
Variant	Solved	Nodes	Time in s
PNS	10	693,949	258
PNS + Frequency10 (k=2)	10	92,597	35
PNS + Availability10 (k=1)	10	143,657	56
PNS + Deep Learning (a=2,b=1,c=1)	9	258,096	9,319

Table 6.14: Comparison of nodes evaluated and execution for search enhancements on Hex positions.

Conclusion. Both LOA and Hex positions show that the frequency and availability enhancement can be applied to achieve reductions in both nodes evaluated and time required. While the deep learning enhancement achieves reductions in nodes evaluated, it does not perform as well as the other two enhancements and requires more time to solve the same number of positions.

Overall, three well performing enhancements, applicable to multiple games have been proposed. The actual performance is highly impacted by design choices. It is therefore difficult to draw a conclusion on which enhancement reduces the number of evaluated nodes the most, however if the goal is to reduce computation time, frequency and availability enhancements are preferred over a deep learning enhancement.

6.8.2 Othello and Connect6

In contrast to LOA and Hex, Othello and Connect6 positions show to be more difficult to apply PNS to. Ideas for reasons behind this are given in the following.

Othello. Even though PNS variants are able to solve 5 out of 20 Othello positions, no improvements are achieved by any of the enhancements. The amount of experiments does not allow for explicit reasons to be drawn for this, nonetheless one can assume that the rules of Othello impact the performance of PNS. Firstly, games are played until the board is filled and the winner is decided thereafter. This characteristic prevents PNS from focusing on narrow search paths. Every path in Othello has to be searched until the board is filled, furthermore Othello has a branching factor that is not related to any notion of winning or losing. Whereas a small branching factor in LOA indicates fewer options to move, which remains until the end of the game, a small branching factor does not give an indication of following branching factors along the same path.

Connect6. As illustrated in Section 3.1, the ability to solve game positions depends on the state space and search space complexity of the game. With the high complexity of Connect6 (Section 3.5), PNS variants are not able to solve any of the given Connect6 positions. This could only be circumvented

by adapting the PNS algorithm, in particular the generation of children nodes. Without changes there are up to $380 \times 379 = 144020$ potential moves in a given position. Creating all children nodes quickly leads to an exceedance of experiment constraints.

6.8.3 Deep Learning

Deep learning has shown to be advantageous for LOA and Hex, when it comes to reducing the number of nodes evaluated. The base for this performance originates from the training process. Networks are successfully trained on boards with size 8×8 and 19×19. The prediction of remaining moves until a terminal position is reached has shown to be more accurately to predict than the outcome. Over all games, the number of moves is predicted with an error of approximately one move, at the end of the training process.

The outcome of the games is predicted with high accuracy ($80\% - 90\%$) in Hex and Connect6. Both of these games carry similarities when it comes to winning the game; a consecutive line of pieces has to be connected, which seems to be predictable by the networks. The sequential placing of pieces on the board shows which player is about to make the next move, which is an important aspect of predicting the winner of a game. The same can be concluded from an Othello position, as one piece is placed every move. Based on the design choice to keep the network as simple as possible and not include the player who has to move, it cannot be concluded with certainty which player is about to make a move in a LOA position. This can be one reason why the prediction of the outcome is less accurate on LOA. The prediction of the outcome of Othello games shows to be less accurate as well. This could be owing to the nature of Othello, as the winner is decided after the board is entirely filled, leading to long-term dependencies.

A deficit of using the deep learning enhancement is an increased execution time to solve positions. An increase by a factor of 20 in LOA and 40 in Hex can be witnessed. Two reasons can be given for this. Firstly, making predictions with a neural network is more computationally complex than initializing pn and dpn with 1. Secondly, additional computational overhead is required to process information of the current position such that it can be used in the neural network. Using a more efficient programming language than Python for implementing experiments could reduce the overhead of preprocessing board information.

Chapter 7

Conclusion

This chapter concludes the thesis. The three research questions as well as the problem statement established in Chapter 1 are discussed by reviewing results obtained in previous chapters. Concluding, proposals for future research are given.

7.1 Research Questions

In the following, the three research questions proposed in Section 1.4 are addressed.

1. *How do the different PNS variants perform?*

Experiments show that regular PNS requires the most nodes evaluated for solving positions, followed by PN*. Nonetheless PNS is has a faster execution time than PN*. PDS and df-pn are the best performing one-level searches. In LOA, df-pn performs slightly better than PDS; in Hex, PDS performs slightly better than df-pn. Kishimoto et al. (2012) claimed that dfpn-pn "probably replace PDS-PN in domains where PDS-PN is preferable to df-pn and PN^2". This shows, that dfpn-pn can be used to achieve improvements over other PNS variants. Experiments show, that dfpn-pn reduces the number of nodes searched in LOA. dfpn-pn has the fastest running-time in LOA, Hex and Othello among all two-level variants. This shows, that with dfpn-pn, an improvement in PNS has been achieved in regards to the running time. An improvement in regards to the number of nodes evaluated can be seen in LOA.

2. *How can proof and disproof number of leaf nodes be initialized to improve the search procedure, independent of game domain?*

Expert games are used as a base to a scoring method to initializing *pn* and *dpn*. The first approach is to analyze move performed in each game. This can be used to either count the frequency of how often moves are played in regards to the number of games, or the frequency of moves played in regards to them being

available in a position. Both of these enhancements achieve reductions in the number of nodes searched in PNS for the games LOA and Hex. The move availability enhancement outperform the mobility enhancement in LOA, in terms of execution time by a factor of 3, while evaluating a comparable number of nodes. It furthermore allows for a speedup for solving Hex problems, where a mobility enhancement is not applicable. Interestingly, the proposed enhancement do not only lead to improvements in their specific game domain, but also for LOA on a smaller board. Both, frequency and availability enhancement can logically be applied to all of the four games, except for the frequency enhancement for Othello. This is show that a combination of both is broadly applicable, in contrast to the mobility enhancement which could be applied to LOA.

3. *Can deep learning be used to improve the search procedure by initializing proof and disproof number of leaf nodes?*

Experiments show that deep learning methods reduce the number of nodes evaluated in LOA and Hex. This is done by processing board positions and estimating the winner of the current position and the number of moves required to reach the terminal position. These two factors are used to calculate *pn* and *dpn* predictions. The deep learning enhancement has achieved improvements by reducing the number of nodes evaluated in LOA and Hex. In LOA, the number of nodes evaluated is reduced by 80% and in Hex by 50%. While the number of nodes evaluated decreases, the execution time increases by a factor of 20 in LOA and 40 in Hex. This shows that that deep learning can be used to improve the search by reducing the number of nodes evaluated with a trade-off of increasing the execution time.

7.2 Problem Statement

The problem statement of this thesis was defined in Section 1.4 as follows:

How to improve Proof-Number search for two-player board games?

An improvement of PNS has been investigated with two different approaches, by search variants and search enhancements. The newly proposed search variant two-level df-pn has shown to reduce the processing time among two-level search variants over the games LOA, Hex and Othello. The frequency and availability enhancement can easily be applied to various game domains to achieve improvement, as shown for LOA and Hex. Both enhancements achieve reductions in evaluated nodes and execution time. A deep learning enhancement is interesting reducing the number of evaluated nodes. However, it is inferior to frequency and availability enhancements, based on higher execution times at the current state.

7.3 Future Research

While improvements have been achieved in PNS, there are research opportu-

nities for to investigate proposed techniques in more depth or investigate new ideas.

PNS variants, in particular two-level variants, carry variables that can be chosen, which impact the search procedure. This includes the size of the second-level search as well as the choice of ϵ. Further experiments could be conducted to adjust those variables. For experiments in this thesis, their value has been fixed. Accordingly, formulas have been proposed, in Section 5, to initialize *pn* and *dpn*. An in-depth analysis of potential formulas could lead to a further improvement.

Rather than creating improvements for various games concurrently, research could focus on individual games. On one hand, this can lead to specific neural network models that are fine tuned for particular games. Those would not only consider three input layers, which is a minimum of information gathered from a board position, but can include domain specific knowledge such as the current player to move. One the other hand, formulas to apply an initialization of *pn* and *dpn* with move frequency/availability can be chosen more precisely.

Different games are also of interest, such as tsume-shogi Seo et al. (2001). In tsume-shogi, different piece types occur. It would be interesting to evaluate the performance of deep learning models on games with different piece types.

An open field of research in PNS is the use of three-level search. Three-level search for games has been used in Allis (1988), however not with PNS. Allis used a knowledge-based search on the first level, CNS on the second level and depth first search on the third level to assess Connect4 positions. The created codebase of this thesis, which is available on Github[1] allows for a simple concatenation of layers and therefore the use of three-level search without the need of further development.

[1]https://github.com/maxhort/ProofNumber-Search

Bibliography

Allis, L. V. (1988). A Knowledge-based Approach of Connect-Four - The Game is Solved: White Wins. Master's thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands.

Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Maastricht University, Maastricht, The Netherlands.

Allis, L. V., van der Meulen, M., and van den Herik, H. J. (1994). Proof-number search. *Artificial Intelligence*, 66(1):91 – 124.

Arneson, B., Hayward, R. B., and Henderson, P. (2011). Solving Hex: Beyond Humans. In van den Herik, H. J., Iida, H., and Plaat, A., editors, *Computers and Games*, pages 1–10, Berlin, Heidelberg. Springer Berlin Heidelberg.

Barratt, J. and Pan, C. (2017). Playing Go without Game Tree Search Using Convolutional Neural Networks. `http://cs231n.stanford.edu/reports/2017/pdfs/603.pdf` (Last checked on June 02, 2019).

Berkey, D. D. (1988). *Calculus*. Saunders College Publishing, New York NY, USA.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Bouton, C. L. (1901). Nim, A Game with a Complete Mathematical Theory. *Annals of Mathematics*, 3(1/4):35–39.

Bowden, B. V., editor (1953). *Faster Than Thought: A Symposium on Digital Computing Machines*. Pitman Publishing, Inc., Marshfield, MA, USA.

Breuker, D. M., Uiterwijk, J. W. H. M., and van den Herik, H. J. (1996). Replacement Schemes and Two-Level Tables. *ICCA Journal*, 19(3):175–180.

Breuker, D. M., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2001a). The PN^2-search algorithm. In van den Herik, H. J. and Monien, B., editors, *Advances in Computer Games*, volume 9, pages 115–132.

Breuker, D. M., van den Herik, H. J., Uiterwijk, J. W. H. M., and Allis, L. V. (2001b). A solution to the GHI problem for best-first search. *Theoretical Computer Science*, 252(1):121 – 149.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Buro, M. (1997). An Evaluation Function for Othello Based on Statistics. *Technical Report 31, NEC Research Institute.*

Campbell, M. (1985). The Graph-history Interaction: On Ignoring Position History. In *Proceedings of the 1985 ACM Annual Conference on The Range of Computing : Mid-80's Perspective: Mid-80's Perspective*, ACM '85, pages 278–280, New York, NY, USA. ACM.

Campbell, M., Hoane, A. J., and Hsu, F.-h. (2002). Deep Blue. *Artificial Intelligence*, 134(1):57 – 83.

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, pages 72–83, Berlin, Heidelberg. Springer-Verlag.

David, E., Netanyahu, N. S., and Wolf, L. (2017). DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess. *CoRR*, abs/1711.09667.

Frey, P. W. (1983). *Chess Skill in Man and Machine*. Springer-Verlag, Berlin, Heidelberg, 2nd edition.

Fürnkranz, J. and Hüllermeier, E. (2010). *Preference Learning*. Springer-Verlag, Berlin, Heidelberg, 1st edition.

Gale, D. (1979). The Game of Hex and the Brouwer Fixed-Point Theorem. *The American Mathematical Monthly*, 86(10):818–827.

Gao, C., Hayward, R., and Müller, M. (2018). Move Prediction Using Deep Convolutional Neural Networks in Hex. *IEEE Transactions on Games*, 10(4):336–343.

Gao, C., Müller, M., and Hayward, R. (2017). Focused Depth-first Proof Number Search using Convolutional Neural Networks for the Game of Hex. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 3668–3674.

Gao, Q. and Xu, X. (2016). A Solving Strategy of Connect6 Based on K-in-a-row types. In *2016 Chinese Control and Decision Conference (CCDC)*, pages 5041–5045.

Gardner, M. (1957). Mathematical games: Concerning the game of Hex, which may be played on the tiles of the bathroom floor. *Scientific American*, 197(1):145–150.

Greenblatt, R. D., Eastlake, III, D. E., and Crocker, S. D. (1967). The Greenblatt Chess Program. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 801–810, New York, NY, USA. ACM.

Hein, P. (1942). Vil de laere Polygon? (in Danish). Article in Politiken newspaper, 26 December.

Henderson, P. (2010). *Playing and Solving the Game of Hex.* PhD thesis, University of Alberta, Alberta, Canada.

Henderson, P., Arneson, B., and Hayward, R. B. (2009). Solving 8×8 Hex. In *Twenty-First International Joint Conference on Artificial Intelligence*, pages 505–510.

Hlynur Daví, H. (2017). Predicting expert moves in the game of Othello using fully convolutional neural networks. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden.

Imran, G. (2004). Reinforcement Learning in Board Games. Technical report, University of Bristol, United Kingdom.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement Learning: A Survey. *CoRR*, cs.AI/9605103.

Kishimoto, A. and Müller, M. (2004). *DF-PN in Go: An Application to the One-Eye Problem*, pages 125–141. Springer US, Boston, MA.

Kishimoto, A. and Müller, M. (2005). A solution to the GHI problem for depth-first proof-number search. *Information Sciences*, 175(4):296 − 314. Heuristic Search and Computer Game Playing IV.

Kishimoto, A., Winands, M. H. M., Müller, M., and Saito, J.-T. (2012). Game-Tree Search Using Proof Numbers: The First Twenty Years. *ICGA journal*, 35:131–156.

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

LeCun, Y. and Bengio, Y. (1998). The Handbook of Brain Theory and Neural Networks. chapter Convolutional Networks for Images, Speech, and Time Series, pages 255–258. MIT Press, Cambridge, MA, USA.

Liskowski, P., Jaskowski, W., and Krawiec, K. (2017). Learning to Play Othello with Deep Neural Networks. *CoRR*, abs/1711.06583.

McAllester, D. A. (1988). Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, 35(3):287 – 310.

Nagai, A. (1998). A new AND/OR Tree Search Algorithm Using Proof Number and Disproof Number. In *Complex Games Lab Workshop*, pages 40–45, Tsukuba. ETL.

Nagai, A. (1999). A New Depth-First-Search Algorithm for and/or Trees. Master's thesis, University of Tokyo, Tokyo, Japan.

Nagai, A. (2001). *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Graduate School of the University of Tokyo, Tokyo, Japan.

Nagai, A. and Imai, H. (1999). Application of dfpn+ to Othello Endgames. In *Game Programming Workshop in Japan 99*, pages 16–23, Hakone, Japan.

Nash, J. F. (1952). Some Games and Machines for Playing them. Technical Report D-1164, Rand. Corp.

Nijssen, J. A. M. (2013). *Monte-Carlo Tree Search for Multi-Player Games*. PhD thesis, Maastricht University, Maastricht, The Netherlands.

O'Shea, K. and Nash, R. (2015). An Introduction to Convolutional Neural Networks. *ArXiv e-prints*. `https://arxiv.org/pdf/1511.08458.pdf` (Last checked on June 02, 2019).

Palay, A. J. (1985). *Searching with Probabilities*. Pitman Publishing, Inc., Marshfield, MA, USA.

Pawlewicz, J. and Hayward, R. B. (2013). Scalable parallel DFPN search. In *International Conference on Computers and Games*, pages 138–150. Springer.

Pawlewicz, J. and Lew, Ł. (2007). Improving Depth-First PN-Search: $1+\epsilon$ Trick. In van den Herik, H. J., Ciancarini, P., and Donkers, H. H. L. M. J., editors, *Computers and Games*, pages 160–171, Berlin, Heidelberg. Springer Berlin Heidelberg.

Runarsson, T. P. and Lucas, S. M. (2014). Preference Learning for Move Prediction and Evaluation Function Approximation in Othello. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):300–313.

Saito, J.-T., Chaslot, G., Uiterwijk, J. W. H. M., and Herik, H. J. (2006). Monte-Carlo Proof-Number Search for Computer Go. In *International Conference on Computers and Games*, pages 50–61. Springer.

Saito, J.-T., Winands, M. H. M., and van den Herik, H. J. (2010). Randomized Parallel Proof-number Search. In *Proceedings of the 12th International Conference on Advances in Computer Games*, ACG'09, pages 75–87, Berlin, Heidelberg. Springer-Verlag.

Sakuta, M. and Iida, H. (2001). The Performance of PN*, PDS, and PN Search on 6×6 Othello and Tsume-Shogi. In *Advances in Computer Games*, volume 9, pages 203–222. Maastricht University, The Netherlands.

Schadd, M. P. D. (2011). *Selective Search in Games of Different Complexity*. PhD thesis, Maastricht University, Maastricht, The Netherlands.

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212.

Schaeffer, J., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers Is Solved. *Science*, 317:1518–1522.

Seo, M., Iida, H., and Uiterwijk, J. W. H. M. (2001). The PN*-search algorithm: Application to tsume-shogi. *Artificial Intelligence*, 129:253–277.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of Go without Human Knowledge. *Nature*, 550:354–359.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. (2014). Striving for Simplicity: The All Convolutional Net. *CoRR*, abs/1412.6806.

van den Herik, H. J., Uiterwijk, J. W. H. M., and van Rijswijck, J. (2002). Games solved: Now and in the future. *Artificial Intelligence*, 134:277–311.

van den Herik, H. J. and Winands, M. H. M. (2008). Proof-Number Search and Its Variants. In Tizhoosh, H. R. and Ventresca, M., editors, *Oppositional Concepts in Computational Intelligence*, pages 91–118. Springer Berlin Heidelberg, Berlin, Heidelberg.

van der Ree, M. and Wiering, M. (2013). Reinforcement Learning in the Game of Othello: Learning Against a Fixed Opponent and Learning from Self-Play. In *2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL)*, pages 108–115. IEEE.

van Rijswijck, J. (2000). Are Bees Better than Fruitflies? In Hamilton, H. J., editor, *Advances in Artificial Intelligence*, pages 13–25, Berlin, Heidelberg. Springer Berlin Heidelberg.

von Neumann, J., Morgenstern, O., and Rubinstein, A. (1944). *Theory of Games and Economic Behavior*. Princeton University Press.

Winands, M. H. M. (2004). *Informed Search in Complex Games*. PhD thesis, Maastricht University, Maastricht, The Netherlands.

Winands, M. H. M. (2008). 6×6 LOA is solved. *ICGA Journal*, 31(4):234–238.

Winands, M. H. M. and Björnsson, Y. (2007). Enhanced Realization Probability Search. In *Information Sciences 2007*, pages 643–649, Singapore. World Scientific Publishing Company.

Winands, M. H. M. and Björnsson, Y. (2010). Evaluation Function Based Monte-Carlo LOA. In van den Herik, H. J. and Spronck, P., editors, *Advances in Computer Games*, Lecture Notes in Computer Science, pages 33–44. Springer.

Winands, M. H. M., Herik, H. J., and Uiterwijk, J. W. H. M. (2004a). An Evaluation Function for Lines of Action. In *Advances in Computer Games*, pages 249–260. Springer.

Winands, M. H. M. and Uiterwijk, J. W. H. M. (2001). PN, PN^2 and PN* in Lines of Action. In *The CMG Sixth Computer Olympiad Computer-Games Workshop Proceedings. Technical Reports in Computer Science CS*, pages 01–04.

Winands, M. H. M., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2002). Pds-pn: A new proof-number search algorithm. In *International Conference on Computers and Games*, pages 61–74. Springer.

Winands, M. H. M., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2004b). An effective two-level proof-number search algorithm. *Theoretical Computer Science*, 313(3):511–525.

Wu, I., Huang, D.-Y., Chang, H.-C., et al. (2005). Connect6. *ICGA Journal*, 28(4):235–242.

Wu, I.-C. and Huang, D.-Y. (2006). A New Family of k-in-a-Row Games. In van den Herik, H. J., Hsu, S.-C., Hsu, T.-s., and Donkers, H. H. L. M. J., editors, *Advances in Computer Games*, pages 180–194, Berlin, Heidelberg. Springer Berlin Heidelberg.

Wu, I.-C., Lin, H.-H., Lin, P.-H., Sun, D.-J., Chan, Y.-C., and Chen, B.-T. (2011). Job-level Proof-number Search for Connect6. In *Proceedings of the 7th International Conference on Computers and Games*, CG'10, pages 11–22, Berlin, Heidelberg. Springer-Verlag.

Xu, C.-m., Ma, Z., Tao, J.-j., and Xu, X.-h. (2009). Enhancements of Proof Number Search in Connect6. In *2009 Chinese Control and Decision Conference*, pages 4525–4529. IEEE.

Yen, S. and Yang, J. (2010a). Searching for Stage Proof Number in Connect6. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pages 413–420.

Yen, S.-J. and Yang, J.-K. (2010b). New Simulation Strategy of MCTS for Connect6. In *The 15th Game Programming Workshop*, pages 90–93.

Young, K. and Hayward, R. B. (2016). A Reverse Hex Solver. In *International Conference on Computers and Games*, pages 137–148. Springer.

Young, K., Vasan, G., and Hayward, R. (2016). NeuroHex: A Deep Q-learning Hex Agent. In *Computer Games*, pages 3–18. Springer.

Zhang, S., Iida, H., and van den Herik, H. J. (2017). Deep df-pn and Its Efficient Implementations. In Winands, M. H. M., van den Herik, H. J., and Kosters, W. A., editors, *Advances in Computer Games*, pages 73–89, Cham. Springer International Publishing.

Zobrist, A. (1970). A New Hashing Method with Application for Game Playing. Computer Science Department, The University of Wisconsin, Madison, WI, USA. Technical Report 88.

Appendices

Appendix A

PNS-Pseudocode

Algorithm 3 Proof-Number Search with backpropagation enhancement - Part 1

1: **function** PNS(root)
2:     EVALUATE(root)
3:     SETPROOFANDDISPROOFNUMBERS(root)
4:     $currentNode \leftarrow root$
5:     **while** $root.pn \neq 0$ and $root.dpn \neq 0$ **do**
6:         $mostProving \leftarrow$SELECTMOSTPROVINGNODE(root)
7:         EXPANDNODE(mostProving)
8:         $currentNode \leftarrow$ UPDATEANCESTORS(mostProving,root)
9: // Calculate proof and disproof numbers
10: **function** SETPROOFANDDISPROOFNUMBERS(node)
11:     **if** $node.expanded$ **then** // internal node
12:         **if** $node.type == AND$ **then**
13:             $node.pn \leftarrow \sum_{\forall c \in node.children} c.pn$
14:             $node.dpn \leftarrow \min_{\forall c \in node.children} c.dpn$
15:         **else** // OR Node
16:             $node.pn \leftarrow \min_{\forall c \in node.children} c.pn$
17:             $node.dpn \leftarrow \sum_{\forall c \in node.children} c.dpn$
18:     **else** // OR Node
19:         **if** $node.value == TRUE$ **then**
20:             $node.pn \leftarrow 0$
21:             $node.dpn \leftarrow \infty$
22:         **else if** $node.value == FALSE$ **then**
23:             $node.pn \leftarrow \infty$
24:             $node.dpn \leftarrow 0$
25:         **else** // value UNKNOWN
26:             $node.pn \leftarrow 0$
27:             $node.dpn \leftarrow 0$

Algorithm 4 Proof-Number Search with backpropagation enhancement - Part 2

```
28: function SelectMostProvingNode(node)
29:     while node.expanded do
30:         if node.type == AND then
31:             node ← arg min_{∀c∈node.children} c.dpn
32:         else // OR Node
33:             node ← arg min_{∀c∈node.children} c.pn
34:     return node
35: // Expand Node
36: function ExpandNode(node)
37:     GenerateChildren(node)
38:     for for each child c of node do
39:         Evaluate(c)
40:         SetProofAndDisproofNumbers(c)
41:     node.expanded ← true
42: // Update Ancestors
43: function UpdateAncestors(node,root)
44:     do
45:         oldPN ← node.pn
46:         oldDPN ← node.dpn
47:         SetProofAndDisproofNumbers(node)
48:         if oldPN == node.pn and oldDPN == node.dpn then
49:             return node
50:         if node == root then
51:             return node
52:         node ← node.parent
53:     while true
```

Appendix B

Endgame Positions Hex

Piece positions are given for each endgame position.

Position 1:

$$black = a1, f2, g2, e3, e4, f4, g4, d5, d6, f6, c7, f7, g7$$
$$white = f1, h1, d2, b3, f3, g3, e5, e6, g6, d7, b8, c8, e8$$

Position 2:

$$black = h1, c2, h2, b3, f3, h3, b4, d4, f4, b5, e5, f6, g6, a7, e7$$
$$white = d1, g1, b2, e2, g2, c3, g3, c4, g4, c5, d5, g5, d6, f7, g7$$

Position 3:

$$black = a1, c2, d2, f2, b3, e3, g3, a4, c4, d4, a5, d5, e5, f5, a6, c6, a7, c7, d7, a8$$
$$white = c1, d1, e1, b2, e2, c3, d3, b4, e4, f4, g4, b5, g5, b6, d6, e6, b7, b8, c8, d8$$

Position 4:

$$black = b2, e2, g2, b3, f3, c4, d4, d5, e5, a6, c6, a7, d7, a8$$
$$white = d2, f2, c3, d3, e3, b4, f4, c5, b6, d6, f6, b7, c7, b8$$

Position 5:

$$black = c2, a4, a5, b5, b6, c6, b7, b8$$
$$white = e1, b3, c3, b4, c5, h7, a8, c8$$

Position 6:

$$black = a2, g2, b3, b4, c4, d4, e4, f4, f5, d6, e6, c7$$
$$white = a1, c1, d2, c3, e3, f3, g3, c5, d5, e5, d8, g8$$

Position 7:

$$black = c2, f2, d3, f3, c4, b5, d5, c6, c7, d7, e7, a8, e8$$
$$white = d2, e2, c3, e3, b4, d4, e4, d6, g6, b7, b8, c8, d8$$

Position 8:

$$black = f1, e2, b3, e3, c4, e4, d5, f5, c6, e7$$
$$white = d1, e1, g1, b2, d3, d6, e6, c7, b8, g8$$

Position 9:

$$black = d2, b3, b4, b5, d5, b6, c6, c7, e7$$
$$white = b2, e2, c3, d3, h3, c5, e6, b7, c8$$

Position 10:

$$black = d1, b4, c4, g4, d5, e5, g5, a6, c6, a7, f7, a8$$
$$white = e2, c3, f3, f4, f5, b6, d6, e6, b7, c7, b8, h8$$

Position 11:

$$black = h1, a2, c2, d2, e2, f2, a3, f3, g3, a4, c4, d4, e4, g4, a5, b5, f5, g5, a6, c6, g6, b7$$
$$white = c1, d1, e1, f1, g1, b2, g2, h2, b3, d3, e3, b4, f4, c5, d5, e5, b6, e6, f6, a7, f7, f8$$

Position 12:

$$black = c2, a3, c3, a4, c4, d4, e4, a5, d5, a6, a7, b7, d7, b8$$
$$white = b2, d2, g2, b3, d3, e3, b4, b5, c5, e5, c6, d6, c7, a8$$

Position 13:

$$black = b2, e2, g2, h2, d3, f3, h3, a4, c4, e4, f4, h4, c5, d5, h5, a6, e6, g6, a7, e7, f7, h7, a8, c8$$
$$white = g1, h1, f2, a3, c3, e3, g3, d4, g4, a5, b5, e5, f5, g5, b6, d6, f6, h6, b7, d7, b8, d8, e8, f8$$

Position 14:

$$black = c2, c3, f3, g3, c4, e4, b5, b6, b7$$
$$white = g1, e2, e3, h3, b4, c5, c6, d7, a8$$

Position 15:

$$black = b2, g2, b3, d3, e3, g3, a4, e4, g4, a5, e5, g5, a6, e6, a7, b7, e8$$
$$white = d1, c2, e2, f2, c3, f3, b4, f4, b5, d5, f5, b6, f6, d7, f7, a8, h8$$

Position 16:

$$black = e1, b2, e2, f2, g2, h2, h3, a4, c4, f4, h4, c5, f5, h5, b6, d6, f6, h6, b7, d7, e7, f7, g7, h7$$
$$white = f1, g1, h1, d2, b3, d3, f3, g3, e4, g4, b5, d5, e5, g5, c6, e6, g6, c7, c8, d8, e8, f8, g8$$

Position 17:

$$black = f1, e2, f2, f3, f4, g4, f5, e6, f7$$
$$white = e1, g1, e3, g3, e5, h5, c7, e7, d8$$

Position 18:

$$black = a1, g1, b2, c2, d2, e2, g2, a3, c3, e3, h3, a4, c4, f4, h4, a5, c5, h5, a6, d6, a7, b7, c7, d7, e8$$
$$white = b1, c1, d1, e1, f1, h1, a2, f2, b3, d3, f3, b4, d4, b5, d5, e5, g5, b6, c6, e6, e7, a8, b8, c8$$

Position 19:

$$black = h1, h2, b3, d3, e3, h3, c4, h4, b5, d5, e5, f5, g5, b6, d6, b7, f7$$
$$white = g1, d2, e2, g2, c3, g3, b4, d4, e4, g4, c5, c6, e6, g6, c7, d8, e8$$

Position 20:

$$black = f1, e2, e3, f3, g3, g4, b5, d5, e5, f5, a6, c6, a7, e7, a8$$
$$white = e1, f2, g2, d3, c4, e4, f4, c5, b6, d6, e6, b7, d7, b8, g8$$

Appendix C

Data Augmentation

C.1 LOA

LOA has three mirror axes that are used for augmenting data. Therefore, a single position is transformed to four positions, with the initial position and three mirrored positions. Given the position of a piece by (r, c) where r indicates the row and c the column of the piece, the following mirror axes exist:

- horizontal: $(9 - r, c)$
- vertical: $(r, 9 - c)$
- diagonal: $(9 - r, 9 - c)$

Coordinates are given in the range of $(1, 1) - (8, 8)$.

C.2 Hex

Hex has one mirror axis. Given an 8×8 board the following mirror procedure is applied to coordinates (r, c):

$$(r, c) -> (9 - r, 9 - c)$$

Without augmentation, there are 44,730 games where black wins and 6,065 where white wins. In order to reduce the imbalance in results, only games where white wins are mirrored. This results in 12,130 games with white winning after augmentation.

C.3 Othello and Connect6

No data augmentation techniques have been applied to Othello and Connect6. The amount of available data is sufficient and does not require augmentation.

Appendix D

Deep Learning: Output Visualization

D.1 LOA

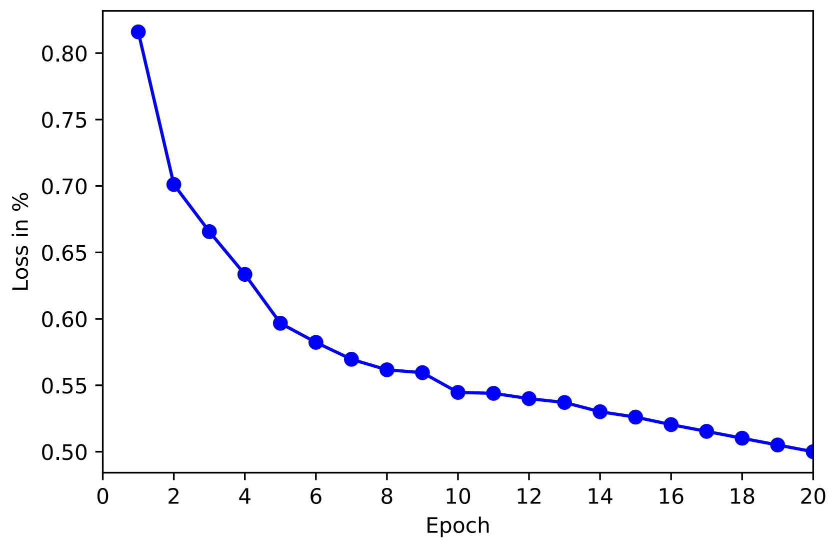

Figure D.1: Outcome Loss in % for NW2 on LOA.

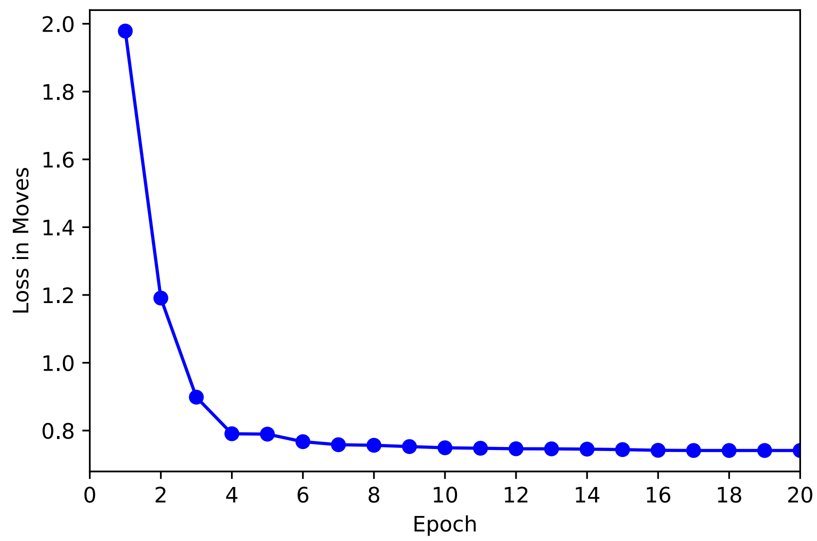

Figure D.2: Distance Loss in moves for NW2 on LOA.

D.2 Hex

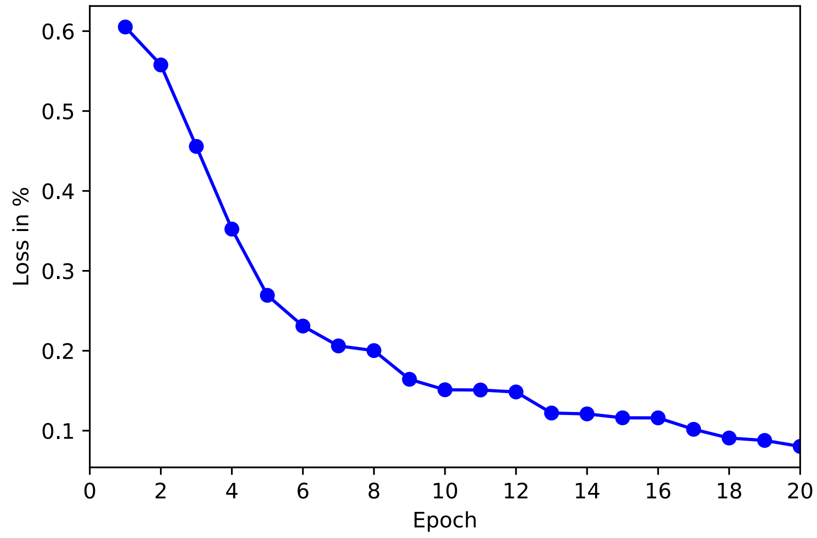

Figure D.3: Outcome Loss in % for NW1 on Hex.

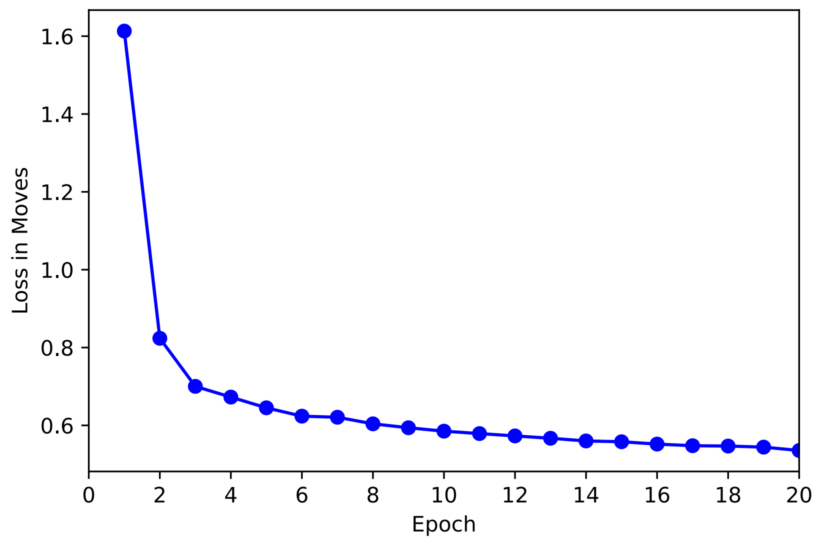

Figure D.4: Distance Loss in moves for NW1 on Hex.

D.3 Othello

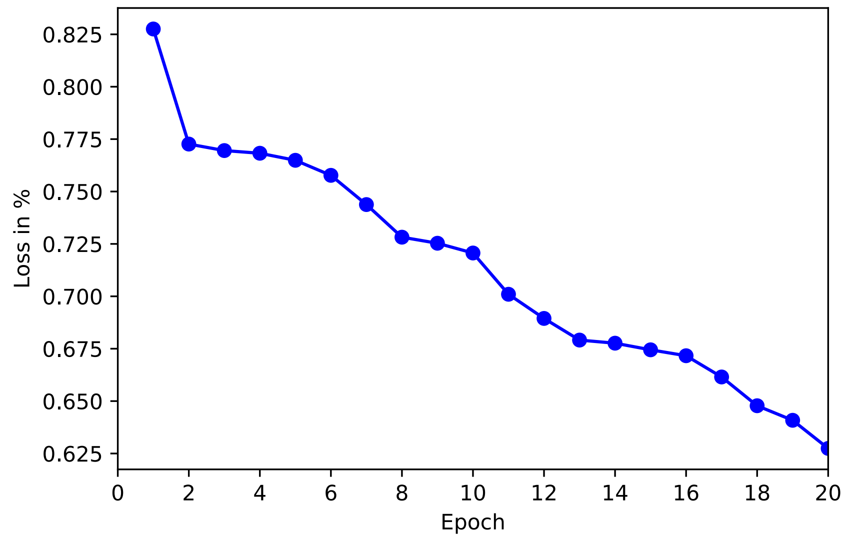

Figure D.5: Outcome Loss in % for NW1 on Othello.

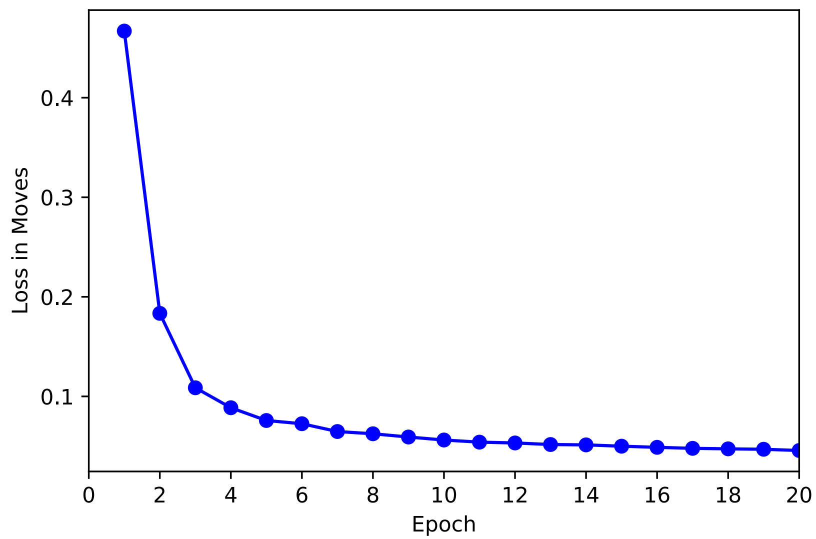

Figure D.6: Distance Loss in moves for NW1 on Othello.

D.4 Connect6

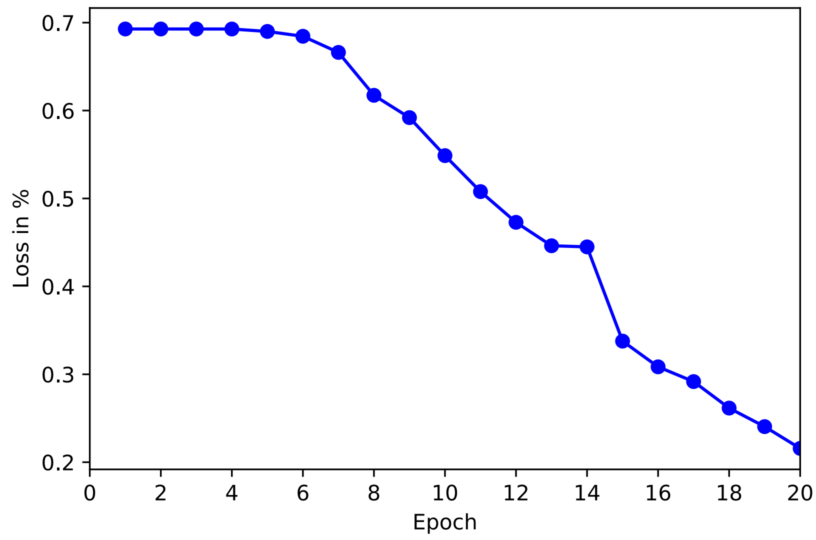

Figure D.7: Outcome Loss in % for NW4 on Connect6.

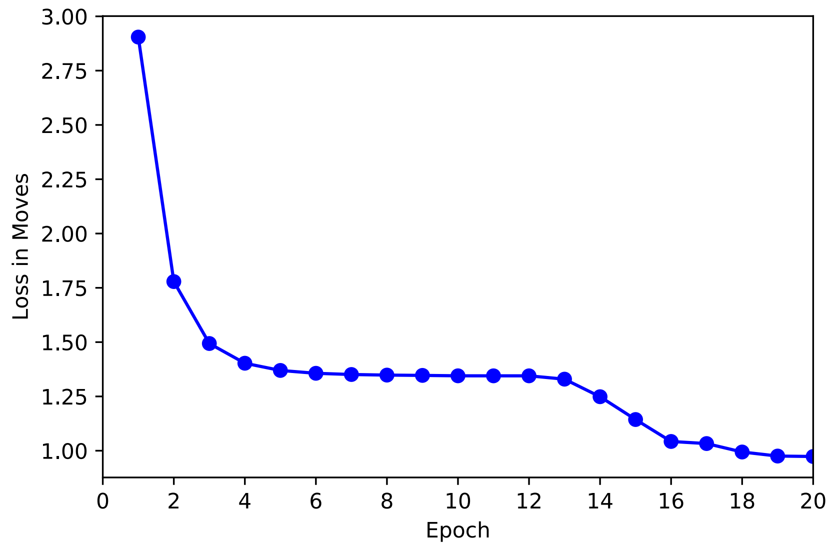

Figure D.8: Distance Loss in moves for NW4 on Connect6.