TU WIEN Informatics

An AlphaZero Agent for Just 4 Fun, a Non-Deterministic Game with Imperfect Information

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Mario Gastegger
Matrikelnummer 00726289

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.Ass. Mag.rer.soc.oec. Daniel Obszelka

Wien, 20. August 2024

Mario Gastegger   Günther Raidl

Informatics

An AlphaZero Agent for Just 4 Fun, a Non-Deterministic Game with Imperfect Information

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Mario Gastegger
Registration Number 00726289

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Univ.Ass. Mag.rer.soc.oec. Daniel Obszelka

Vienna, August 20, 2024

Mario Gastegger | Günther Raidl

Erklärung zur Verfassung der Arbeit

Mario Gastegger

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. August 2024

Mario Gastegger

Danksagung

Ich möchte meinen Betreuern für ihre Geduld und Unterstützung während dieser mehrjährigen Diplomarbeit danken. Besonders Daniel Obszelka für seinen persönlichen Einsatz zur Verbesserung des Agenten und Unterstützung bei der Fehlersuche. Ich möchte auch allen Spielern auf yucata.de dafür danken, dass sie Benchmark-Daten bereitgestellt haben, indem sie gegen die implementierten Agenten gespielt haben.

Acknowledgements

I want to thank my supervisors for their patience and support throughout this years-spanning thesis. Especially Daniel Obszelka for his personal commitment to improving and fixing the agent. I want to also thank all the players on yucata.de for providing benchmark data by playing against the implemented agents.

Kurzfassung

Im Streben nach Fortschritt in der künstlichen Intelligenz und den Suchalgorithmen wird ein Agent als ein System betrachtet, das Entscheidungen trifft und Aufgaben auf der Grundlage seines Verständnisses der Situation oder des Environments ausführt. Dieses Environment ist das umgebende System oder die Welt, mit der der Agent interagiert. Nichtdeterministische Environments mit unvollständigen Informationen sind durch Zufallsereignisse und einen Zustand gekennzeichnet, der den Agenten vollständig oder teilweise verborgen bleibt.

Das AlphaZero-Framework ist bei mehreren schwierigen Spielen wie Go, shogi und Schach sehr erfolgreich. Es verfügt über einen Baumsuchalgorithmus, der von einem deep neural network geleitet wird. Das Netzwerk wurde ohne Verwendung menschlichen Expertenwissens, außer den Spielregeln, durch self-play mithilfe eines allgemeinen reinforcement learning Algorithmus trainiert.

Bis vor Kurzem gab es keine allgemeinen Frameworks für nichtdeterministische Environments mit unvollständigen Informationen. In dieser Arbeit schlagen wir eine neuartige Erweiterung von AlphaZero vor, die in solchen Umgebungen funktioniert.

Unser Algorithmus, betitelt AlphaJust4Fun, unterscheidet sich von AlphaZero, indem wir die Monte Carlo Baumsuche durch den Single-Observer Information Set MCTS Algorithmus ersetzen. Der Single-Observer Information Set MCTS Algorithmus ist nicht von vollständig bekannten Umgebungen abhängig, da die Suche auf Knoten durchgeführt wird, die die Suchstatistiken zufälliger Instanziierungen der verborgenen Teile eines bestimmten Umgebungszustands kombinieren.

Wir implementieren einen Prototyp und evaluieren ihn mit dem hybriden Brett- und Kartenspiel Just 4 Fun im Zwei-Spieler-Modus. Wir evaluieren unseren Algorithmus mit zwei verschiedenen Netzwerkarchitekturen an Testsätzen, die auf bestimmte Aspekte des Spiels abzielen, und in einem Benchmark. Als Referenzalgorithmus für den Benchmark verwenden wir einen Monte Carlo Tree Search Algorithmus, dem die sonst verborgenen Teile der Spielzustände bekannt sind, und menschlichen Spielern.

Die Ergebnisse zeigen, dass AlphaJust4Fun mit den verborgenen Informationen und dem Nichtdeterminismus in Just 4 Fun erfolgreich umgehen kann. Es übertrifft den Referenzalgorithmus und kann auch mit erfahrenen menschlichen Spielern mithalten. Unsere Experimente zeigen, dass die Kombination des DNN und der Baumsuche des

AlphaJust4Fun-Agenten besser abschneidet als jede Komponente für sich. Im Gegensatz zu neueren AlphaZero-Erweiterungen, die mehrere zusätzliche neuronale Netzwerke verwenden, erfordert AlphaJust4Fun nur einen zusätzlichen Hyperparameter.

Abstract

In the pursuit of advancements in artificial intelligence and search, an agent can be considered as a system that makes decisions and performs tasks based on its understanding of the situation, or the environment. This environment is the surrounding system or world that the agent operates in, providing the agent with information and responding to its actions. Non-deterministic environments with imperfect information are characterised by chance events and a state that is fully or partially hidden from the agents.

The AlphaZero framework has great success in several hard games like Go, shogi, and chess. It features a tree search algorithm that is guided by a deep neural network. The network was trained without using any human expert knowledge besides the rules of the games through self-play using a general reinforcement learning algorithm.

Not until very recently, there were no general frameworks for non-deterministic environments with imperfect information. In this thesis, we propose a novel extension of AlphaZero which does work in these environments.

Our algorithm is termed AlphaJust4Fun. The difference to AlphaZero is that we replace the Monte Carlo Tree Search with the Single-Observer Information Set MCTS. The Single-Observer Information Set MCTS does not depend on perfect information, as the search is performed on nodes that combine the search statistics of random instantiations of the hidden parts of a particular environment state.

We implement a prototype and evaluate it on the hybrid board and card game Just 4 Fun in its two-player setting. We evaluate our algorithm with two different neural network architectures on test sets which target certain aspects of the game and a benchmark. As a baseline for the benchmark, we use a Monte Carlo Tree Search algorithm that searches with full knowledge of the hidden parts of the game state and human players.

The results indicate that AlphaJust4Fun successfully handles hidden information and non-determinism in Just 4 Fun. It outperforms the baseline and can also compete with experienced human players. Our experiments indicate that the AlphaJust4Fun agent's combination of the DNN and the tree search performs better than each component on its own. In contrast to more recent AlphaZero-extensions that use multiple neural networks, AlphaJust4Fun requires only one additional hyperparameter.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Even though there is a myriad of real-world problems, which AI methods can be applied to, games have often served as abstractions and also benchmarks for the design of novel approaches and the improvement of existing methods. The history of AI for solving games started early. In 1948, even before computers were available, Alan Turing and David Champernowne invented a lookahead-based algorithm for chess. [49, 10, 40]

One class of algorithms that has been successfully applied in game AI is tree search. Nodes represent states of the game and the edges, the transition between parent and child state, represent the actions chosen by agents. For adversarial two-player zero-sum games (e.g. Go, chess, Connect 4, Tic-Tac-Toe, ...) the **Minimax algorithm** [42] returns an optimal strategy by searching the whole game tree. In each node, the player's reward is maximised. The performance of Minimax is highly dependent on the branching factor of a game: A high branching factor means a high number of possible actions from each state. This in turn means, in a given period of time, the search tree cannot be explored to the same depth as would be possible for games with a lower branching factor, which in general results in a lower agent strength. For simple games like Tic-Tac-Toe, searching the whole game tree is still feasible. But in more complex games like chess it is not feasible anymore. Knuth and Moore describe the **α-β procedure** [30], a method to optimise Minimax, where parts of the search tree, that would lead to worse outcomes than already explored parts, are pruned. This allowed IBM's Deep Blue, which is based on α–β search, to beat Garry Kasparov, who was the reigning world champion in chess in 1997. Deep Blue's reward function contained a lot of domain knowledge by grand masters of chess and was highly tuned to that particular game. Since the game of Go has an even higher branching factor than chess, α–β search has not been successful in achieving even amateur human-level performance in this game.

In 2007, Coulom introduced **Monte Carlo Tree Search (MCTS)** [11], which avoids the necessity for a reward function by using the outcome of random game continuations instead. As in α–β search, not all branches of the game tree are explored to the same depth. The promising ones are more likely to be explored. MCTS is also suitable for imperfect information and non-deterministic games because its random playouts can result in a better estimate of a game state's quality and a better approximation of the corresponding Minimax tree, as Yannakakis and Togelius pointed out in their book on artificial intelligence and games [53]. Using MCTS, Go computer programs began to be successful in a smaller version of Go.

Silver et al. of Google DeepMind made a big breakthrough in 2016 with their **AlphaGo** program [45] by beating the professional Go player Lee Sedol. AlphaGo uses a **deep neural network (DNN)** (supervised learning policy network) that is trained on human expert data to predict the next move. This network is then improved by using **reinforcement learning (RL)** and **self-play (SP)** (reinforcement learning policy network).
A separate network (value network) is trained to predict the game outcomes. AlphaGo then used the policy and value network in a modified version of Monte Carlo Tree Search to select the next node. In 2017, **AlphaGo Zero** was introduced, demonstrating superior performance compared to its predecessor's, achieving this even without relying on any human knowledge except for the rules of the game [46]. It replaces the two networks in AlphaGo with a single convolutional residual neural network with a policy output as well as a value output.

The field of artificial intelligence has seen rapid growth and major advancements in the past two decades. RL is a paradigm, where no extensive training dataset is necessary. The system is learning to take actions within its environment to maximise its reward based on a predefined function. RL has seen major breakthroughs by acquiring a superhuman skill level in hard games, where exact solutions are infeasible.

AlphaZero (AZ) [44] represents such a major milestone since it required no human knowledge except for the rules of a game and was still able to achieve superhuman performance on several strictly determined two-player zero-sum games with perfect information (chess, shogi, and Go), i.e. where the entire state of the game is visible to both players.

AlphaZero is a framework for building AI agents based on the following components:

A general purpose *reinforcement learning algorithm* that uses self-play, i.e. improving by competing against itself.

A deep neural network to approximate the *value function*, i.e. predict the game outcome from a given state of the game, and predict the *policy*, i.e. a probability distribution over the available actions from this state.

And a general purpose *Monte Carlo Tree Search algorithm* for generation of high-quality actions.

The previously discussed approaches address deterministic games without hidden information or randomness. Another category of games that presents a distinct challenge are stochastic games with imperfect information. In these games, the state of the game is fully or partially concealed, and events occur non-deterministically. This includes actions like shuffling and dealing cards or rolling dice. Examples of such games are Poker and Dou dizhu. Successfully navigating stochastic games with hidden information, particularly those involving multiple players, remains a challenging undertaking.

Libratus, introduced by Brown and Sandholm [6], represents an application-agnostic framework for solving imperfect information games, as it was the first to successfully defeat top human professionals in the challenging game **heads-up no-limit Texas hold'em poker (HUNL)**. HUNL is the two-player variant of no-limit Texas hold 'em poker. Much like AZ, Libratus does not rely on domain knowledge or human training data.

The framework consists of 3 modules:

> *Abstraction*: The first module creates a simpler abstraction of the underlying task and computes an initial strategy that is used in the early stages of the game.
>
> *Subgame-solving*: In the later stages of the game, strategies for specific subgames are computed.
>
> *Self-improvement*: The third module improves the initial strategy based on opponents' actions.

Pluribus [7] represents another major milestone but in the domain of multiplayer games of imperfect information. It achieved a superhuman skill level in six-player no-limit Texas hold 'em poker, employing SP and **Monte Carlo Counterfactual Regret Minimisation (MCCFR)** which turned out to be a powerful method to address uncertainty. **Counterfactual Regret Minimization (CFR)** is an iterative SP algorithm that traverses the game tree by taking actions and investigating how much better or worse it would have done, having chosen the other available actions instead, and then minimises the regret.

More recently, AlphaZero has been extended to handle tasks within dynamic environments (MuZero [43]) and stochastic environments (Stochastic MuZero [1]).

1.2 Goals of this Work

Until recently, most of the research focused on hard two-player games with perfect information. This thesis contributes to the research by addressing problems that involve multiple agents in an environment with partially hidden states and stochastic events, by employing RL and tree search. The hybrid board/card game **Just 4 Fun (J4F)** [54] serves as a benchmark problem. Just 4 Fun is a non-deterministic multiplayer game with imperfect information.

The goals of this thesis are the following:

- Since AlphaZero's design does not provide particular elements to handle non-deterministic games with imperfect information, to modify AlphaZero such that it can be applied effectively to games with those properties.
- To model Just 4 Fun's game mechanics and game state in the context of MCTS. In particular, the aim is to address the two-player version of J4F.
- To find suitable inputs for the DNN.
- To evaluate the resulting AI agents in a benchmark.
- To achieve at least human level performance. We use *Yucata.de*, which is a web-based online gaming platform, as a reference for human performance.
- To obtain more generally applicable algorithmic principles, machine learning architectures and concepts to handle the aspects of randomness and hidden information.

Just 4 Fun is played on a 6×6 board of 36 fields and with a deck of cards. Each field has a number between 1 and 36 and each number occurs exactly once. Further, there are playing cards with values between 1 and 19 and each player is in possession of 20 stones. In each turn, a player can play a subset of 1–4 cards out of their hand. The sum of the cards' values indicates the field on which the player places one of their stones. A player wins if they reach the majority (more stones than the opponents) on 4 fields in a row. Either horizontally, vertically or diagonally. If no player can get four in a row, before running out of stones, the player with the most points wins. A player gets the number of points equal to the field value, for each field they hold the majority on. If neither player has 4 in a row nor more points than any other player, the player who has the majority on the field with the highest value wins. There may be at least 2 and up to 4 players.
The basic rules are taken from https://www.yucata.de/ [55]. A picture of the tabletop edition of Just 4 Fun is depicted in Figure 1.1. A more thorough description of Just 4 Fun follows in Section 3.1.

1.3 Structure of this Work

In the following sections, a detailed overview of the structure and organisation of this thesis is provided, offering insights into the key components and their interconnections.

Chapter 1 An introduction of the research on game-playing AI is presented. The structure is designed to facilitate a logical progression of ideas and insights gained through rigorous investigation.

Figure 1.1: Achim Raschka (https://commons.wikimedia.org/wiki/File:Just_4_Fun_01.jpg), CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0), via Wikimedia Commons

Chapter 2 This chapter begins with a thorough exploration of existing literature to establish a solid foundation for the research. It reviews key concepts and empirical studies relevant to game-playing AI and heuristic planning in non-deterministic environments with imperfect information.

Chapter 3 To illustrate the applications of the research topic, we first present a concrete constrained example problem through the game Just 4 Fun. With its non-deterministic element and imperfect information, it provides a tangible application of the theoretical concepts. The chapter continues with a detailed description of the two methods, the AlphaZero framework and Information Set Monte Carlo Tree Search, the proposed approach is based on.

Chapters 4 and 5 These chapters present AlphaJust4Fun, the approach that is proposed to tackle the problem of planning in non-deterministic environments with imperfect information in an effective and efficient way.

Chapter 6 This chapter outlines the methods used to assess the proposed algorithm's effectiveness and provides essential insights into its implementation of the algorithm described in Chapter 4. The chapter aims to offer a clear and thorough account of the experimental groundwork, including measurement techniques, to support the subsequent findings and conclusions.

Chapter 7 A dedicated chapter is allocated to present the results of the analysis and detailed exposition of key findings. It employs statistical tools and graphical representations to interpret the data and derive meaningful insights.

Chapter 8 Building upon the results, this chapter critically interprets the findings in the context of existing literature. It explores implications, limitations, and offers insights that contribute to the broader understanding of the topic.

Appendices Supplementary materials, such as algorithms, configurations, and additional analyses, are included in the appendices to maintain the flow of the main narrative.

CHAPTER 2

Related Work

In this chapter, we conduct a comprehensive review of the existing body of research, elucidating the advancements and challenges that have significantly influenced the landscape of artificial intelligence in games.

2.1 Fundamentals

This section aims to introduce methodologies, terms, and concepts that are important to the understanding of the concepts our agent is based on.

2.1.1 Terms and Game Theory

Game theory is a broad field that includes reasoning about games as well as any competitive activity in which entities contend with each other according to a set of rules. As Osborne outlines in their introduction to game theory [38], it is used in economics, politics, social sciences, biology as well as the more recent field of cryptocurrencies and crypto economics, and computer games.

Extensive Games Extensive games are a model for the description of games in which players sequentially take actions. Players can change their strategy at every decision point according to their expected reward, i.e. their strategy is not required to be predetermined. In accordance with Osborne and Rubinstein [39], Definition 2.1.1 illustrates the characteristics of extensive games with perfect information.

Definition 2.1.1 (Extensive games with perfect information)**.** An **extensive game with perfect information** has the following components:

- The set of players N.
- A set H of action sequences (finite or infinite) that satisfies the following three properties:
  - The empty sequence $\emptyset$ is a member of H.
  - If $(a^k)_{k=1,\ldots,K} \in H$ (where K may be infinite) and $L < K$ then $(a^k)_{k=1,\ldots,L} \in H$
  - If an infinite sequence $(a^k)_{k=1}^{\infty}$ satisfies $(a^k)_{k=1,\ldots,L} \in H$ for every positive integer L, then $(a^k)_{k=1}^{\infty} \in H$.

  A history $(a^k)_{k=1,\ldots,K} \in H$ is terminal if it is infinite or if there is no a^{K+1} such that $(a^k)_{k=1,\ldots,K+1} \in H$. The set of terminal histories is denoted by $Z \subset H$.
- A function P that assigns to each non-terminal history (each member of $H \setminus Z$) a member of N.
- For each player $i \in N$, a preference relation $\succeq_i$ on Z (the preference relation of player i).

Game Tree An extensive game with perfect information can be described as a tree, where the nodes represent the states s of the game and the edges represent the actions a taken by players. $h \in H$ are the sequence or history of actions $(a^k)_{k\in\mathbb{N}}$. For each leaf node s_t (i.e. a terminal state), the terminal history h_t of actions is $(a^k)_{k=1,\ldots,t}$. Z is the set of all terminal histories, and a subset of H. The relation $\succeq_p$, maps action sequences $h \in Z$ to a value, that indicates the payoff or reward $z \in \mathbb{R}$ for the player p.

Figure 2.1 shows an example of a game tree for a two-player game.

Branching factor The number of possible actions in each state of a game is called the branching factor. In terms of a game tree, this is the number of edges from each state. For many games, the branching factor is not a constant. E.g. the game of Tic-Tac-Toe has a branching factor of 9 at the first state and only 1 at the state, in which only 1 field is not yet filled. For those games, however, the average branching factor can be calculated instead.

Observability Observability refers to the observability of the game state. Games in which the state is fully known to all players at all times are called **perfect information** games. Games with hidden or partially hidden information, are called **imperfect information** games. The card game Poker is an imperfect information game as a result of the players' private cards. The board game chess is a perfect information game, as the board and all its pieces are visible to both players.

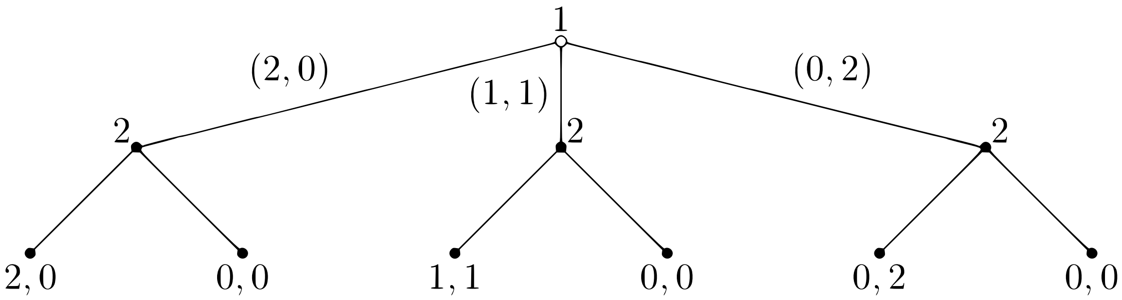


Figure 2.1: The game tree for a simple two-player game with perfect information (Figure 91.1 on page 91 in [39]). The numbers at the inner nodes of the tree represent the player and the edges their respective actions. At the leaf nodes, the game has terminated, and the numbers below the leaf nodes are the players' rewards. The first one is the reward for player 1 and the second, for player 2. The numbers in brackets next to the edges show the highest achievable reward for each player. E.g. $1, 2$ is a win for player 2 and $0, 0$ a draw.

Stochasticity Games that contain some sort of randomness, e.g. rolling of dice or shuffling cards, are called **non-deterministic**. In contrast, games without any elements of chance are called **deterministic**. Poker is an example of a non-deterministic game, and chess is one of a deterministic game.

Extensive Games with Imperfect Information Extending the definition of extensive games with perfect information, as described in Definition 2.1.1, to allow for players to have only partial information about previous events, as well as exogenous uncertainty, provides a better model for J4F. It covers the initial shuffle of the stack of cards, the unknown hand of other players and possible reshuffles in later stages of the game. The definition of extensive games, as given by Osborne and Rubinstein [39] is shown in Definition 2.1.2:

Definition 2.1.2 (Extensive games)**.** An **extensive game** with possibly imperfect information has the following components:

- A set N (the set of players).
- A set H of sequences (finite or infinite) that satisfies the following three properties:
  - The empty sequence $\emptyset$ is a member of H.
  - If $(a^k)_{k=1,\dots,K} \in H$ (where K may be infinite) and $L < K$ then $(a^k)_{k=1,\dots,L} \in H$
  - If an infinite sequence $(a^k)_{k=1}^{\infty}$ satisfies $(a^k)_{k=1,\dots,L} \in H$ for every positive integer L, then $(a^k)_{k=1}^{\infty} \in H$.

  A history $(a^k)_{k=1,\dots,K} \in H$ is terminal if it is infinite or if there is no a^{K+1} such that $(a^k)_{k=1,\dots,K+1} \in H$. The set of actions available after the non-terminal history h is denoted $A(h) = \{a\colon (h, a) \in H\}$, and the set of terminal histories is denoted Z.

- A player function P that assigns to each non-terminal history (each member of $H \setminus Z$) a member of $N \cup \{c\}$, i.e. a player or a chance event.
- A function f_c that associates with every history h for which $P(h) = c$, a probability measure $f_c(\cdot \mid h)$ on $A(h)$, where each such probability measure is independent of every other such measure.
- For each player $i \in N$, a partition $\mathcal{I}_i$ of $\{h \in H : P(h) = i\}$ with the property that $A(h) = A(h')$ whenever h and h' are in the same member I_i of $\mathcal{I}_i$. For $I_i \in \mathcal{I}_i$, we denote by $A(I_i)$ the set $A(h)$ and by $P(I_i)$ the player $P(h)$ for any $h \in I_i$. $\mathcal{I}_i$ is the information partition of player i, and a set $I_i \in \mathcal{I}_i$ is an information set of player i.
- For each player $i \in N$, a preference relation $\succeq_i$ on lotteries over Z (the preference relation of player i) that can be represented as the expected value of a payoff function defined on Z.

Extensive games with imperfect information can also be described with a game tree, as depicted in Figure 2.2.

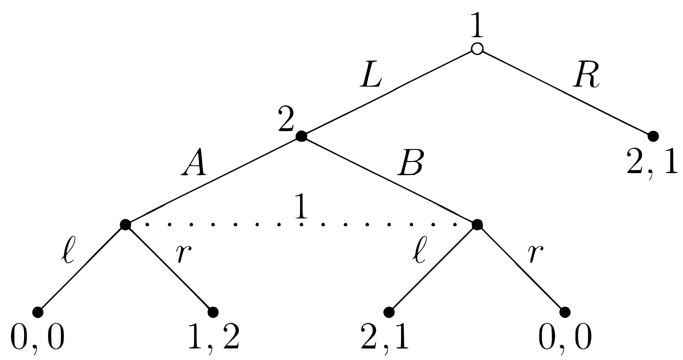


Figure 2.2: The game tree for a simple two-player game with imperfect information (slightly modified version of Figure 202.1 on page 202 in [39]). The numbers at the inner nodes of the tree represent the player and the edges their respective actions. At the leaf nodes, the game has terminated, and the numbers below the leaf nodes are the players' rewards. The first one is the reward for player 1 and the second, for player 2. E.g. $1, 2$ is a win for player 2 and $0, 0$ a draw. The information set for player 1, after the unknown decision of player 2 between A and B, is indicated by the dotted line.

Recall Recall in terms of extensive games refers to the ability of players to recall past actions. Perfect recall is not necessarily the case, but it is a reasonable assumption in the case of an AI agent. Since MCTS/ISMCTS is the policymaker, which keeps track of the cards that have been already played or even only seen, the agent proposed in this work does have perfect recall.

Strategies A strategy for player i is denoted by σ_i, and the set of all strategies is denoted by Σ_i. σ_i is a function, that assigns a probability distribution over $A(I_i)$ for each $I_i \in \mathcal{I}_i$. A **strategy profile** contains a strategy for each player: $\sigma = \{\sigma_i \mid i \in N\}$. σ_{-i} are the strategies $\sigma \setminus \{\sigma_i\}$. The probability of a history h occurring, when players act according to σ, is denoted by $\pi^\sigma(h)$. [32]

Equilibria The Nash Equilibrium is a strategy profile, for which all players play their optimal strategy. That is, no player is better off deviating from its particular strategy, which is also denoted to as the best-response strategy, given that the other players play according to the strategy profile.

Regret Informally, the regret for a player, is the missed expected reward, when deviating from their Nash Equilibrium strategy. Let $u_i(h)$ be the expected reward for a player i, for a history h, and similarly, let $u_i(\sigma)$ the expected reward for Player i under the strategy profile σ and $u_i(\sigma_i^*, \sigma_{-i})$ the expected reward for player i, when following some strategy σ_i^*, while the other players follow the strategies σ_{-i}. Zinkevich et al. [56] propose the following formulation for the average overall regret for a player i. For a repeated game, player i's average overall regret at the time T is $R_i^T = \frac{1}{T} \cdot \max_{\sigma_i^* \in \Sigma_i} \Sigma_{t=1}^T (u_i(\sigma_i^*, \sigma_{-i}^t) - u_i(\sigma^t))$. Where $\sigma_i^t \in \sigma^t$ the strategy of a player i, and σ_i^* the Nash Equilibrium strategy in round t. An iterative, regret minimising algorithm, according to this formulation, approximates a Nash Equilibrium when t goes towards infinity.

$\mathbf{m}, \mathbf{n}, \mathbf{k}$-Games J4F is a multiplayer (2 up to 4 players) m, n, k-game with playing cards. m, n, k-games are played on a grid of m columns and n rows. The goal is to connect k game pieces horizontally, vertically or diagonally. Well-known examples of m, n, k-games are Tic-Tac-Toe, where $m = 3$, $n = 3$ and $k = 3$, and Connect Four, where $m = 7$, $n = 6$ and $k = 4$. In the case of J4F, $m = 6$, $n = 6$ and $k = 4$.

For some of these games, it can be shown that the first player will always win with a perfect play. For others, there is no winning strategy for the first player, they are called a draw. The $3, 3, 3$-game Tic-Tac-Toe is an example of a game-theoretic draw-game.
A $6, 6, 4$-game is a game-theoretic win for the first player [50].

In J4F, however, the state is only partially observable. The board, the already played cards and the player's own cards are known, and the shuffled stack of cards and the opponents' cards are unknown. Thus, there are non-deterministic restrictions on where players are allowed to place their pieces on the board. The fact that players can put multiple pieces on a single field and can both have pieces on the same field, distincts J4F further from the game-theoretic properties of a $6, 6, 4$-game. When played with 3 or 4 players, the game is of incomplete information, since the opponents' policy is not known throughout the whole game. Players might form coalitions for a limited period during a game, or their strategy can change from reaching 4 in a row to a win by points.

Multi-Armed Bandit Problem The multi-armed bandit problem describes the scenario of k one-armed bandits, as it might be found in casinos. When played, each machine yields a reward according to its own distribution. An individual usually wants to maximise its overall reward by only playing those machines which yield the highest expected reward. Definition 2.1.3 formulates the K-armed Bandit Problem as defined by Auer et al. [2].

Definition 2.1.3 (K-armed Bandit Problem)**.** A **K-armed bandit problem** is defined by random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, where each i is the index of a gambling machine. Successive plays of the machine i yield rewards $X_{i,1}, X_{i,2}, ...$ which are independently and identically distributed (according to an unknown law) with unknown expectations μ_i. Independence also holds for rewards across machines; i.e., $X_{i,s}$ and $X_{j,t}$ are independent (and usually not distributed identically) for each $1 \leq i < j \leq K$ and each $s, t \geq 1$.

A human might play every machine several times to get an intuition of the reward distributions, and gradually give preference to the more promising machines. On the one hand, one wants to play every machine to reinforce the intuition about each machine. However, to keep the losses as small as possible or the reward as high as possible, one has to only play the single machine, that yields the highest reward. Since it is not possible to do both, the player at least wants to minimise the regret of not having always played the best machine.

Exactly this dilemma of exploration versus exploitation, in the face of reward maximisation, or conversely regret minimisation, was addressed by Auer et al. [2]. They show that there exists a simple policy **UCB1**, which is described in Definition 2.1.4, for selecting machines over n plays with logarithmic regret.

Definition 2.1.4 (UCB1)**.** Play each machine once. Then continue playing the machine i, that maximises $\overline{x}_i + \sqrt{2} \cdot \sqrt{\frac{\ln n}{n_i}}$, where $\overline{x}_i$ is the current average reward of the machine i, n_i the number of times the machine i has been played and n the overall number of plays.

2.1.2 Monte-Carlo Tree Search

For the literature review on Monte Carlo Tree Search, the survey by Browne et al. [9], which covers variants and extensions up to the year 2012, was an excellent starting point. The survey of Świechowski et al. [48] covers more recent extensions of MCTS up until 2021.

Monte Carlo Tree Search as presented by Coulom [11], combines Monte-Carlo evaluation and tree search.

The general Monte Carlo Tree Search is an iterative algorithm. In each **iteration**, the algorithm performs 4 steps: 1. Selection, 2. Expansion, 3. Simulation and 4. Backpropagation. Figure 2.3 shows a visualisation of the incremental extension of the tree [9].

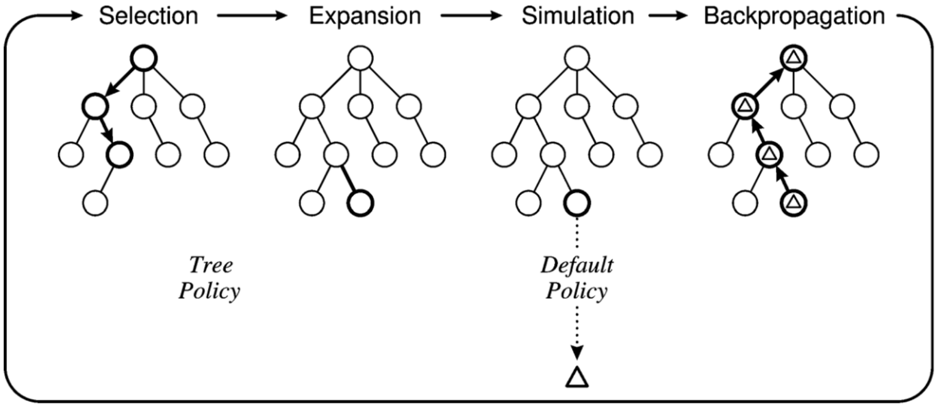


Figure 2.3: Steps of an MCTS algorithm (Fig. 2 on page 6 in [9]).

In the context of extensive games, the game tree is searched. In the selection step, the game tree is traversed, according to some **tree policy**, from the root until a leaf node is encountered. In the expansion step, the tree is expanded by creating a new node as a child of this leaf node. In the simulation step, the game that is represented by the new node is played, selecting random actions (**default policy**) for all players until termination. In the backpropagation step, the resulting value of the simulated game is propagated back up through the tree.

Kocsis and Szepesvári [31] consider the decisions or, selection policy, in a game tree as multi-armed bandit problems. In their algorithm **UCB1 applied to trees (UCT)**, they apply the UCB1 policy to a rollout-based tree search, similar in principle to the one Coulom proposed.

In UCT, each node represents a state s of the game and each edge represents an action a from s. An action's estimated reward Q is the cumulated, discounted reward of the paths originating from s. In the context of the K-armed bandit problem, the actions, or edges, correspond to the machines' arms. The algorithm keeps track of the number of times $N_{s,d}(t)$, s has been encountered at depth d during search until time t. In the selection step, when traversing the tree, the actions that maximise $Q_t(s,a,d) + c_{N_{s,d}(t),N_{s,a,d}(t)}$, are chosen. With $Q_t(s,a,d)$ being the estimated value, $N_{s,d}(t)$ the number of times s has been encountered at depth d until time t and $N_{s,a,d}(t)$ the number of times action a has been chosen from s at depth d until time t. Kocsis and Szepesvári show that the bias term $c_{N_{s,d}(t),N_{s,a,d}(t)} = 2C_p \cdot \sqrt{\frac{\ln N_{s,d}(t)}{N_{s,a,d}(t)}}$ (with an appropriate constant C_p), still accounts for the changing probabilities of the tree policy (caused by the changing reward $Q_t(s,a,d)$), to achieve a logarithmic regret [31].

The worst-case expected regret upper-bound is in $O(\sqrt{K \cdot n \cdot \ln n})$, as shown by [41]. It depends, besides the number of trials n, on the number of arms K or, in the context of game trees, the number of actions. This especially impacts the performance on games with a high branching factor.

Rosin removes the dependence on K with their algorithm **Predictor + UCB (PUCB)** [41]. They introduce a predictor that recommends actions at the root node, before any trials, and improves as more contextual information is added with every trial.

They assume that the reward distributions for the actions from a specific game state is constant over multiple trials. This assumption holds for deterministic games like Go, but not necessarily for non-deterministic ones. It allows for a better upper-bound of $O(\frac{1}{M_*}\sqrt{n \cdot \ln n})$, where M_* is the weight on the optimal arm and M_i (with $i \in [1, K]$ and $\sum_i M_i = 1$) a vector, that assigns a weight to each arm (action). The weights M_i are proportional to the prior probability of the arm i being the optimal arm. As M_* depends on the predictor (Rosin proposes an offline-training approach), the algorithm's performance does now depend on the predictor's quality instead of the number of actions. This makes it possible to handle games with high branching factor, such as Go.

Information Set Monte Carlo Tree Search (ISMCTS), proposed by Cowling et al. [12], is an extension of MCTS that addresses games with hidden information and uncertainty. In ISMCTS, the nodes of the game tree represent, besides the public information, also all possible permutations of the hidden information. The algorithm is described in greater detail in Section 3.3.

2.1.3 Counter-Factual Regret Minimization

In 2007, Zinkevich et al. [56] proposed the concept of counterfactual regret. Counterfactual regret splits the overall regret into per-information set terms, which can be minimised independently. They subsequently show that minimising those terms minimises the overall regret and thus allows to find an approximate Nash Equilibrium. In the domain of poker, while still dealing with a simpler abstraction of the game, their **Counterfactual Regret Minimization (CFR)** algorithm was able to surpass the previous state of the art.

Lanctot et al. [32] extended CFR by Monte Carlo sampling. They show that their domain-agnostic self-play algorithm, **Monte Carlo Counterfactual Regret Minimisation (MCCFR)**, significantly improves overall convergence speed towards a Nash Equilibrium over vanilla CFR.

In 2015, Bowling et al. [5], in continuation of their previous work on CFR, presented **CFR$^+$** which includes several optimisations over previous CFR algorithms. With CFR$^+$, they were the first to successfully handle the full version of heads-up limit Texas hold'em poker (HULHE)[1]. With HULHE, a two-player variant of poker, they solved a challenging real-world game with imperfect information.

[1] They *essentially weakly solved* HULHE, i.e. were able to provide a $\epsilon-$Nash Equilibrium with reasonable small ϵ.

2.1.4 Neural Networks and Learning

Many algorithms [46, 44, 43, 1, 7, 8] that display state-of-the-art performance are utilising reinforcement learning (RL) and self-play (SP).

Self-play is a popular concept in AI and learning, as outlined by Plaat in a dedicated chapter on this topic in his book *Learning to Play: Reinforcement Learning and Games* [40]. One type of SP occurs in planning. MCTS- and CFR-based algorithms, when planning ahead (look-ahead), i.e. traversing the game tree, do explore possible histories. In this case, the AI agent not only performs its own actions, but also acts as the opponent. I.e. in the case of Monte Carlo Tree Search, the tree-policy for action selection. Another way, in which SP is used, is to generate training samples for learning a model. In AlphaZero, the agent plays games against a separate copy of itself. The resulting game traces are used as training samples for improving the DNN.

Reinforcement learning can be divided into model-based and model-free RL. Most state-of-the-art algorithms [44, 43, 1, 7, 8] are hybrids in this regard. Model-free RL uses direct feedback from the environment for value and policy generation. Model-based RL on the other hand uses an intermediate model for planning and determination of value and policy. The model can be a learned model, domain knowledge or rules of the environment.

E.g. in AZ, the learned model is a deep neural network, that serves as a function approximator for value and policy. It is trained end-to-end, using the raw state of the game as input. The output is the estimated outcome of, and the estimated policy from, a game in a given state. The parameters of the network are updated using the stochastic gradient descent method. The network's architecture is based on **residual network (ResNet)** [22]. Besides the other well-known elements of neural networks, i.e. fully connected feed-forward networks, activation functions and batch normalisation, it consists of convolutional layers and skip connections. Convolutions are particularly successful in image classification. Deep convolutional networks can learn basic visual features as well as higher level abstractions. Skip connections, are feeding the input of one component of a neural network, e.g. one or more layers, to the output, i.e. the input of the next component. This reduces the effective depth of the network, which in turn helps to prevent accuracy degradation and vanishing gradients.

All the above concepts are explained in more detail in Plaat's textbook [40] on reinforcement learning.

2.1.5 Performance and Benchmarking

Performance evaluation is necessary to determine the strength of a player. The usually used skill rating system, for humans as well as AI agents, for two player zero-sum perfect information games like chess and Go, is **Elo** [14]. The fact that J4F is a multiplayer game with imperfect information requires a different, more suitable skill rating system. **TrueSkill** [23] is a Bayesian skill rating system that supports draws and an arbitrary number of players with changing skill level. It is well proven and deployed on a large scale in online computer gaming platforms such as Microsoft's Xbox 360 Live [23]. TrueSkill is also necessary for comparison against human J4F players, as it is the system used on https://www.yucata.de/. For experiment evaluation with two players, to determine, e.g. the convergence of different agent configurations or the overall performance, a benchmark based on the win-rate is a sufficient metric.

As the Nash Equilibrium strategy is not known, we will evaluate player strength, similar to other works on MCTS [35, 13], in a benchmark against the following baseline opponents. A cheating MCTS agent, i.e. an agent that performs Monte Carlo playouts with full knowledge about the hidden parts of the game state. And the random agent that chooses among its available actions uniformly at random.

2.2 State of The Art

In this section, we introduce the current state of research.

2.2.1 Towards AlphaZero

Silver et al.'s **AlphaGo** program [45] uses one DNN for policy estimation, i.e. to predict the next move. This has been trained in a supervised fashion, using data from human experts. This network was then improved by RL through SP. For value estimation, a separate network was trained to predict the game's outcome from a given state. AlphaGo then used the policy and value network, in an MCTS algorithm, to select the next node.

In **AlphaGo Zero** [46], the search algorithm was simplified in terms of the used node- and edge-information, and it uses a single DNN for both, value and policy estimation, without any Monte Carlo rollouts. The convolutional residual neural network is now only trained from self-play reinforcement learning, without any human expert knowledge. Only the pieces were used as input features. The tree search implemented the game's rules without any heuristics.

Shortly after, in 2018, a generalised version of the algorithm was published by Silver et al. under the name of **AlphaZero** [44]. AlphaZero was another major breakthrough since it, similar to AlphaGo Zero, didn't require any domain knowledge but the rules, and was able to play chess, shogi and Go at a superhuman skill level. AlphaZero is one of the main foundations of this work. It is explained in great detail in Chapter 3.

2.2.2 CFR based Algorithms for multiplayer games with imperfect information

Since Just 4 Fun is a multiplayer game with up to 4 players, even though not directly used in this work, I want to briefly mention the development in the realm of imperfect information games with more than two players.

In 2017, Moravčík et al. introduced **DeepStack** [37] which marks a major advancement in mastering imperfect information games. DeepStack uses CFR and a guided tree search for policy generation. Two feed-forward networks with several hidden layers are trained using RL and SP to approximate the state's value and to assist the search. With this approach, DeepStack was able to achieve human expert level performance in heads-up no-limit Texas hold'em poker (HUNL), a two player game with imperfect information.

One year later, in 2018, Brown and Sandholm published **Libratus** [6], which was able to beat even the top professionals in HUNL. Libratus is building a *blueprint* for the overall strategy. This blueprint provides a detailed strategy for the early phases of the game. Then, taking the blueprint strategy into account, it computes more fine-grained strategies for subgames in real-time. A third module extends the blueprint strategy. As in DeepStack, the actual game tree is searched using a variant of CFR.

Multiplayer games with imperfect information are much harder to solve. Brown and Sandholm presented **Pluribus** [7] in 2019, which was a major milestone in AI for multiplayer games with imperfect information. Pluribus was able to achieve superhuman performance in six-player no-limit Texas hold'em poker. It uses an abstraction of the game and self-play to pre-compute a strategy for the overall game, and then improves on it during and based on play against an opponent. Pluribus also uses a variant of CFR for searching the actual game tree.

2.3 Beyond AlphaZero

In 2020, Brown et al. published the framework named **Recursive Belief-based Learning (ReBeL)** [8], which combines deep RL and CFR to solve perfect information, as well as imperfect information games. They describe a method to transform any imperfect information game into a perfect information game. The private knowledge about the player's cards is replaced by a public belief state that is known by all players. CFR is used for search in subgames, and the value network[2] is trained by SP. Their system was able to achieve superhuman performance in HUNL. Applied to perfect information games, ReBeL reduces to an algorithm similar to AZ.

Also in 2020, Schrittwieser et al. published a further generalisation of AZ, **MuZero (MZ)** [43], which did not even require a perfect simulator, i.e. the game's rules or the dynamics of a real-world system, and can handle environments with a continuous action space. In addition to the policy and value, the network also predicts an immediate reward to improve planning (model-based RL). For every state in the course of a game, an internal state representation is predicted from the actual state, using a learned function (representation function). From that internal state representation, a MCTS is performed, where follow-up states and their rewards are predicted by a separate learned functions (dynamics function). The tree search is guided by another learned function (policy and value prediction function). The actual next action is sampled from the policy generated

[2] Optionally also a policy network can be trained to bootstrap CFR

by that tree search. The training data that is used to train MuZero is generated from the trajectories (of fictitious states, actions, and rewards), generated by the MCTS. The parameter updates are based on the difference between the fictitious trajectories and trajectories of actual game states, actions, and rewards. MuZero was, in addition to matching AlphaZero's performance in chess, shogi and Go, able to play Atari games (in which traditionally model-free approaches were used) with state-of-the-art performance.

Even more complex and challenging than Atari games is Blizzard Entertainment's real-time strategy game StarCraft II. It resides in the science fiction genre and features three different races. Successful play requires micromanagement units and economy management in a real-time multi-agent environment. It has a vast search space with imperfect information and a combinatorial action space. **AlphaStar** is another implementation of the AlphaZero framework that, for the first time, was able to beat top human professionals [51].

Antonoglou et al. extended MuZero even further. In 2022, they introduce **Stochastic MuZero (SMZ)** in [1]. Previous versions of the framework were planning on, and learning a deterministic model of the environment. To account for uncertainty regarding chance events and partial knowledge of the system, their algorithm features chance nodes within the MCTS. These chance nodes represent states (*afterstates*), which represent the environment after an action has been performed, but before the environment's stochastic responses to that action. With those afterstates, two further learned functions are introduced. One to predict the environment's response to actions (afterstate dynamics function) and a second one to predict the value of afterstates and a distribution over possible chance outcomes (afterstate prediction function). The training of those functions works similar to the training of MZ, with the addition of afterstates within the compared trajectories. Besides matching the state-of-the-art performance in benchmark games like Go, Stochastic MuZero also achieves state-of-the-art performance in stochastic games like 2048 and backgammon.

AlphaZero's combination of RL and planning has also been successful aside from games. **AlphaTensor** is based on AlphaZero and designed to find efficient matrix multiplication algorithms [15]. In contrast to AZ, the problem of finding those algorithms is modelled as a single-player game in AlphaTensor. The neural network is transformer-based. It was able to find an algorithm that improves upon a reference algorithm that has stood for 50 years.

AlphaDev [36] is an algorithm, based on MuZero, to find fast assembly code sorting algorithms. As with AlphaTensor, the problem is formulated as a single-player game and the network is also transformer-based. It was able to find algorithms that outperformed human benchmarks and, as such, was integrated into the LLVM standard C++ sort library.

2.4 Differentiation to ReBeL, MuZero and Stochastic MuZero

While ReBeL, MuZero and especially Stochastic MuZero, are dealing with similar issues as the agent proposed in this thesis is, they follow different approaches.

The differentiating element in ReBeL is the search algorithm. AlphaJust4Fun is using a SO-ISMCTS, while ReBeL's search algorithm is based on CFR. MuZero and Stochastic MuZero use additional neural networks, over the single one used in AlphaZero, to handle imperfect information and non-determinism. These four additional networks, in the case of Stochastic MuZero, mean additional parameterisation and also training effort. This makes them computationally more expensive and potentially more difficult to configure. AlphaJust4Fun's SO-ISMCTS uses, with the number of determinizations per turn, only a single additional hyperparameter over AlphaZero.

CHAPTER 3

Just 4 Fun, AlphaZero and Information Set Monte Carlo Tree Search

The agent that is proposed in this work is termed **AlphaJust4Fun (AZJ4F)**. The non-deterministic multiplayer game Just 4 Fun is used as a benchmark problem to evaluate the performance of AZJ4F. AZJ4F is a combination of the AlphaZero framework and Information Set Monte Carlo Tree Search. This chapter contains the detailed descriptions of the components, AZJ4F is based on. It starts off with the rules of Just 4 Fun, followed by the AlphaZero framework and Information Set Monte Carlo Tree Search.

3.1 Just 4 Fun

Just 4 Fun [26] is a non-deterministic multiplayer board game with playing cards. It was invented by the German game designer *Jürgen P. Grunau* in 2005 [21]. For the basic rules, we used the ones described on https://www.yucata.de/ [54, 55]. Besides the original field-value distribution, which is shown in Figure 3.1, Yucata also implements an ordered distribution (shown in Yucata.de & Just 4 Fun, Figure A.1) and a random distribution. Another source for the game's rules is https://www.spielregeln-spielanleitungen.de, they also provide a picture of the original rule book, published by KOSMOS[1].

The game is played on a 6×6 **board** of 36 numbered **fields** and a numbered **deck** of 55 **cards**. Each field has a number between 1 and 36 and each number occurs only once. The deck contains 4 copies of each card numbered 1 through 12 inclusive, and for cards numbered 13 up to and including 19, there is only 1 copy of each in the deck. Players initially have 20 **stones**. A player can play between 1 and 4 cards (regular action; see

[1]KOSMOS Verlag. https://www.kosmos.de/de

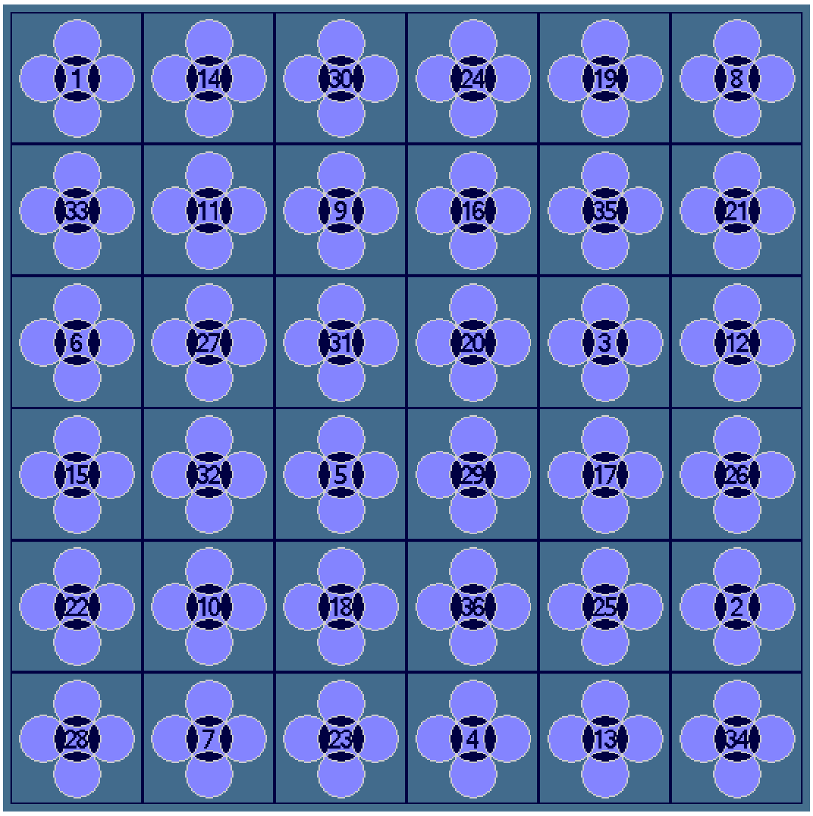


Figure 3.1: Just 4 Fun board with original field value distribution [54]. The value of the fields is indicated by the number in the centre. The players' stones are placed on the light-blue circles above, left of, right of and below the number.

also Figure 3.2) out of their hand and put a stone on the field with the number equal to the sum of the played cards' numbers. A player may not place a stone on a field on which any other player has a **majority** of 2 stones relative to that player (see also Figure 3.4). If there does not exist any card combination indicating a valid field to place a stone on, all cards in the player's hand are replaced with cards from the **stack of cards** (**redraw action**). Cards that have been played are put onto the pile of **used cards**.

At the end of each player's turn, if their hand is depleted, cards are drawn from the stack of cards and added to their hand until it reaches the hand size of 4. This process ensures that all players begin their subsequent turns with a full hand of cards.

If the stack of cards is depleted during the hand refilling process, the pile of used cards is shuffled, and used to construct a new stack of cards from which subsequent draws will occur. This prevents any scenario where a player cannot fully replenish their hand.

A player wins if they can reach a majority in the number stones on 4 fields, aligned horizontally, vertically, or diagonally (Win Condition 1, see Figure 3.3). If no player can fulfil a winning **pattern** before all players run out of stones, the player with the **most points** wins. A player gets the number of points equal to the sum of field values for each field they hold the majority on (Win Condition 2). If neither player has 4 in a line nor more points than any other player, the player who has the majority on the field with the **highest value** wins (Win Condition 3). In the very unlikely case that neither player fulfils the above win conditions (i.e., played the same fields), the game is interpreted as a **draw**. In the case that a player runs out of stones, the other players play on until all stones have been placed. There may be at least 2, up to 4 players. In this thesis, we will only address the game with 2 players.

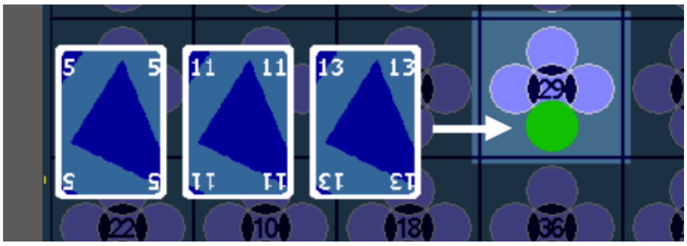


Figure 3.2: Action (with cards): Green is allowed to put their stone on field 29 because the sum of the played cards (5, 11 and 13) equals 29.

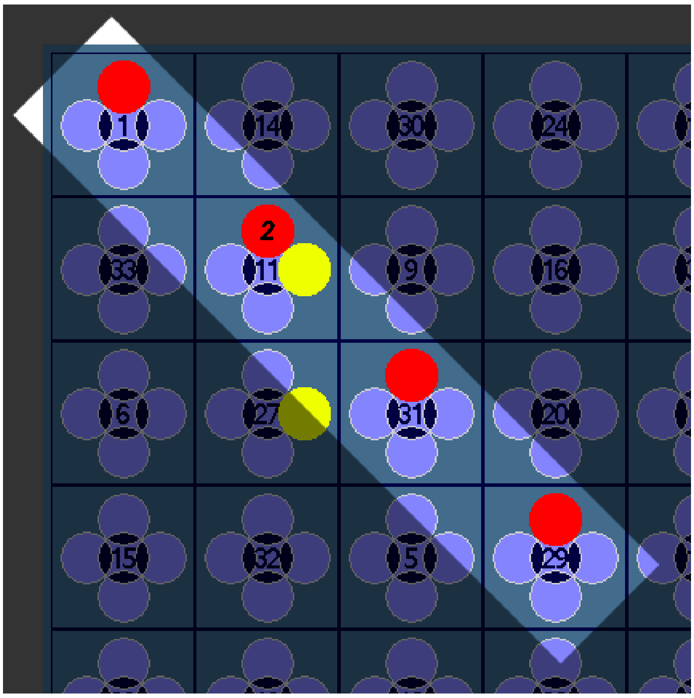


Figure 3.3: Win by pattern (4 in a diagonal, majority): The red player has the majority of stones on every field on the diagonal from field 1 towards field 29. I.e. they have at least one more stone than every other player on those fields. In particular, on 11 they have two stones, whereas yellow has only one stone.

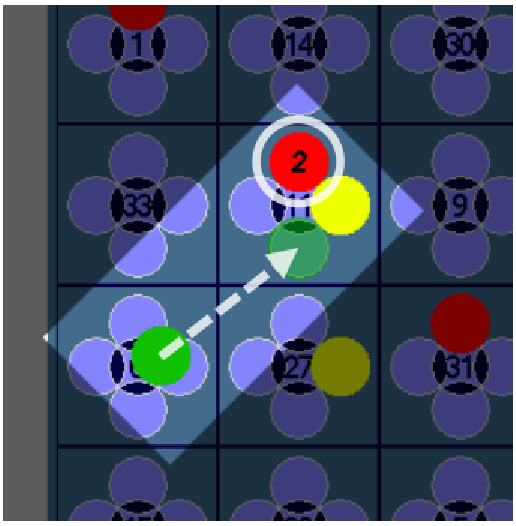


Figure 3.4: An invalid action: Player red has two stones on field 11 and yellow has one. The green player is not allowed, even if they are in possession of the necessary cards, to put one of their stones on this field (the invalid action is indicated by the white, dashed arrow) because the red player has a majority of two stones over green. Player yellow is allowed to put further stones on this field because they have only one stone less than red.

3.1.1 Empirical Analysis of Just 4 Fun

By repeatedly sampling 4 cards from the (full) deck and then counting all the reachable field values from those hands, we calculate an approximation of the probability distribution of reaching fields with a random hand. Figure 3.5 shows the field reachability on the board as heat-map and Figure 3.7 the probability ordered by field value as a barplot. The different probabilities for reaching fields, entails also that some patterns are easier to achieve than others. Figure 3.6 shows the patterns which are easiest to achieve.

We divide the game result into 4 categories:

Win by pattern Either of the players won by constructing a pattern.

Win by points None of the players were able to construct a pattern, but either player had more points than the other players and won the game that way.

Win by max field None of the players were able to construct a pattern and all players even had the same number of points. However, one player held the majority on a field with a higher value than all the fields of the other players.

End in draw The extremely rare case in which all players put all their stones on the same fields and thus did not hold a majority on any field.

1 26.73%	**14** 46.92%	**30** 19.87%	**24** 33.26%	**19** 48.39%	**8** 45.06%
33 13.53%	**11** 54.83%	**9** 48.45%	**16** 47.75%	**35** 10.55%	**21** 40.54%
6 39.24%	**27** 26.70%	**31** 17.63%	**20** 41.95%	**3** 31.64%	**12** 57.85%
15 47.74%	**32** 14.95%	**5** 36.90%	**29** 22.35%	**17** 48.33%	**26** 28.54%
22 38.20%	**10** 51.27%	**18** 48.09%	**36** 9.02%	**25** 30.62%	**2** 28.68%
28 24.39%	**7** 42.50%	**23** 36.18%	**4** 33.80%	**13** 46.71%	**34** 11.81%

Figure 3.5: Empirical field (field-value is in bold) reachability (probability in percent below the field-value) calculated by sampling 10^6 times 4 cards from the (full) deck at random, drawn on the game board as heat-map. As an example, the field with value 1 can be reached with a probability of 26.73% with a random hand.

By simulating two-player games, in which both players select their actions randomly, we found that even then, 45.32% of the games (45,317 out of 100,000) terminated with a *win by pattern* (54,247 or 54.25% with *win by points*, 436 or 0.44% with *win by max field*, and none of the games ended in a *draw*).

As a conclusion, with the set of valid actions constrained by the cards in hand, even though with an opponent, who is not particularly inclined to prevent patterns, it is still reasonably easy to construct a pattern of four in a line in a two-player game.

In a similar experiment over 50,000 games, which consisted of 1,745,860 turns, we found the arithmetic mean of the number of playable fields per turn to be 11.603, with a standard deviation 2.210, and a median of 12.

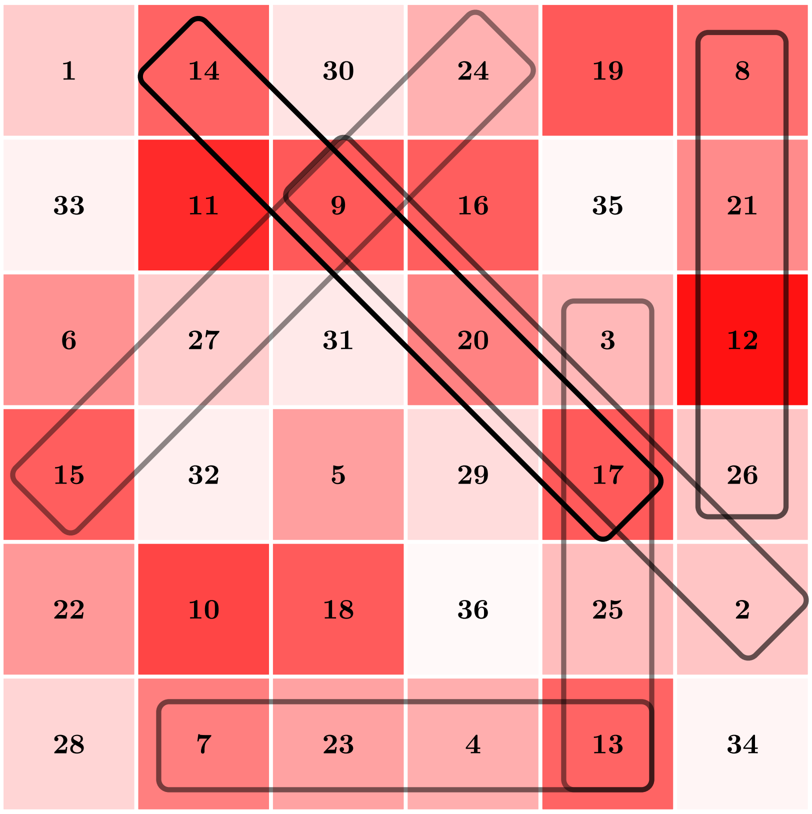


Figure 3.6: Approximately easiest to achieve patterns w.r.t. field reachability. The opacity of the lines reflects the difference in likelihood, relative to the most likely pattern, i.e. 14-9-20-17.

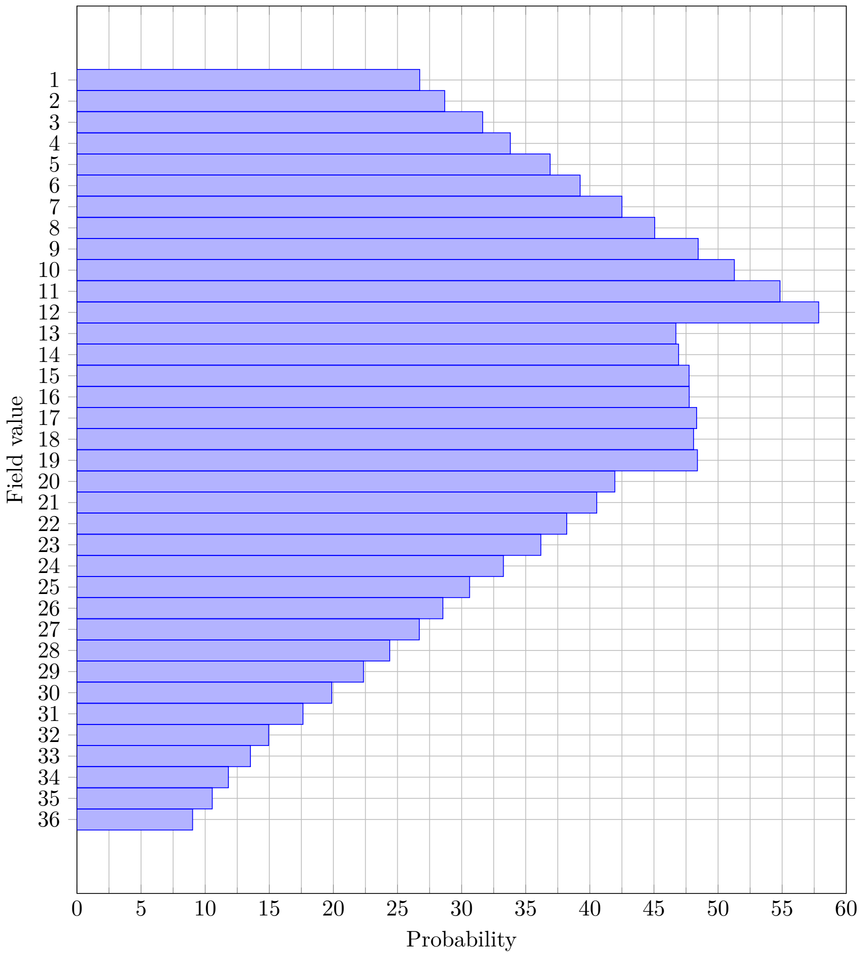


Figure 3.7: Approximate field reachability, calculated by sampling 10^6 times 4 cards from the deck at random, visualised in a barplot. The fields between 1 and 12 are increasingly easier to reach, the fields between 13 and 19 are on a plateau and fairly easy to reach, and the fields between 20 and 36 are getting increasingly harder to reach.

3.2 AlphaZero

The key innovation in AlphaGo Zero [46] was the absence of human domain knowledge involved in training. The algorithm used in AlphaGo [45] was improved through the simplification of the tree search, and the combination of the two neural networks into a single one. AlphaZero's innovation was the high performance on various challenging tasks, like the benchmark problems chess and Go, while using the same parameters and network architecture.

AlphaZero is composed of three main components that complement each other as follows: The general purpose reinforcement learning algorithm is improving the DNN by performing numerous self-play games per training iteration.[2] In each iteration, training samples are generated from the collected game traces and added to a cyclic memory buffer. Batches of samples are then drawn uniformly at random from this buffer and used in a stochastic gradient descent optimiser to update the network's parameters. The updated network is then used in subsequent iterations. During SP and evaluation, a modified version of the PUCB MCTS algorithm [41] is being used. In each game, the MCTS algorithm performs lookahead by searching and expanding the game tree. In classical MCTS, in the simulation step, the value of new leaf nodes is determined by the result of a playout, i.e. continuing the game using random actions for both players until termination. In contrast, AZ is not performing any playouts, but is using the neural network instead to estimate the expected reward and the best-response policy, i.e. a probability distribution over the available actions.

Monte Carlo Tree Search AlphaZero's MCTS maintains a search tree with nodes s (e.g. in Go, the stones on the board) and arcs (s, a) for each action a (e.g. in Go, placing a stone on a certain position on the board). Each edge (s, a) stores the following information:

- The prior probability $P(s, a)$, which is provided by the DNN.
- The visit count $N(s, a)$, i.e. the number of times a has been performed from s during MCTS-iterations.
- The total action value $W(s, a)$, i.e. cumulated action value.
- The action value $Q(s, a)$, i.e. the mean action value $Q(s, a) = \frac{W(s,a)}{N(s,a)}$.

AlphaZero's tree search iteratively extends the game tree on each turn. In each MCTS-iteration, the tree is traversed starting by the root node s_0. At each node s_t, with $t < L$, the actions $a_t \in A(s_t)$ (among the available actions from s_t) are selected, which maximise an upper confidence bound $Q(s, a) + U(s, a)$. Equation (3.1) shows the selection policy.

$$a_t = \arg\max_a (Q(s_t, a) + U(s_t, a)) \tag{3.1}$$

[2]Silver et al. generated a total of 10^6 self-play games for chess.

The term $U(s,a)$, shown in Equation (3.2), penalises arcs (s,a) that have often been selected in previous MCTS-iterations ($\frac{\sqrt{N(s)}}{1+N(s,a)}$).

$$U(s,a) = C(s) \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)} \tag{3.2}$$

The exploration rate, denoted by $C(s)$, increases slowly with each iteration of the MCTS-iteration algorithm that passes the node s. This increase follows the formula $C(s) = c_{\text{init}} + \log \frac{1+N(s)+c_{\text{base}}}{c_{\text{base}}}$, where c_{init} and c_{base} are constants[3], and $N(s)$ represents the number of visits of node s.

Once a preliminary leaf node s_{L} (i.e. a node that is not yet present in the search tree) is reached, the tree is extended. The new node's edges $(s_{\text{L}},a) \mid \forall a \in A(s_{\text{L}})$ are initialised with $N(s_{\text{L}},a)=0$, $W(s_{\text{L}},a)=0$, $Q(s_{\text{L}},a)=0$ and $P(s_{\text{L}},a)=p_a$. The prior probability $P(s,a)$ is initialised with the policy estimation $\mathbf{p}$ of the neural network f_θ, with $(\mathbf{p},v)=f_\theta(s_{\text{L}})$ and p_a being the policy component for the respective edges (s_{L},a).

In a backpropagation step for all $t \leq \text{L}$, the visit counts and action values are updated as described in Equation (3.3), with v being the value, i.e. the outcome of the game from s_t, estimated by the neural network $(\mathbf{p},v)=f_\theta(s_t)$.

$$\begin{aligned} N(s_t,a_t) &= N(s_t,a_t)+1 \\ W(s_t,a_t) &= W(s_t,a_t)+v \\ Q(s_t,a_t) &= \frac{W(s_t,a_t)}{N(s_t,a_t)} \end{aligned} \tag{3.3}$$

The number of MCTS-iterations can be controlled by either a time limit or by a constant. Algorithm 3.1 contains the pseudocode for AZ's MCTS.

After all MCTS-iterations have been performed, i.e. the search has completed, the visit-based probabilities $\pi(s_0)$, over the actions available in s_0, are returned. The probability for playing action a_i ins_0, with $a_i \in A(s_0)$, is shown in Equation (3.4). The temperature τ controls the exploitation.

$$\pi(a_i|s_0) = \frac{\sqrt[\tau]{N(s_0,a_i)}}{\sum_{a_j \in A(s_0)} \sqrt[\tau]{N(s_0,a_j)}} \tag{3.4}$$

The temperature in AZ is controlled by the number of turns t into the game. During self-play, the first 30 turns are performed with $\tau(t)=1 \mid_{1 \leq t \leq 30}$. This results in actions being selected proportionally to their visit count $N(\cdot,\cdot)$, and thus more exploration. For following turns $t>30$, τ is set to an infinitesimal value, which results in more exploitation (heavy bias towards most frequently visited actions). Another way, in which exploration is achieved, is the addition of Dirichlet noise to the prior probabilities at s_0. I.e. setting $P(s_0,a)=(1-\epsilon)\cdot p_a + \epsilon \cdot \eta_a$, with $\eta \sim Dir(\alpha)$. The parameter α was chosen as the inverse of the approximate number of legal moves in a typical position.[4]

[3]Silver et al. use c_{base}=19,652 and $c_{\text{init}}=1.25$.

[4]Silver et al. use $\alpha=0.3$ for chess.

Algorithm 3.1: AlphaZero's MCTS algorithm

Input: s_0 - the a root node of a game subtree, composed of nodes s_w and arcs (s_x, a_y), where some player z is about to play
n_{iter} - the number of iteration

for *n_{iter} iterations* **do**
    Start from the root node, assigning $s_i = s_0$
    repeat `// Selection`
        Descend the subtree by selecting arcs (s_i, a_j) with $a_j \in A(s_i)$, that maximise the upper confidence bound $Q(s_i, a_j) + U(s_i, a_j)$, and updating s_i to the selected child node $s_i = s_j$ `// see Eq. (3.1)`
    until *an arc (s_i, a_j), that leads to a node which is not in the game tree yet, is reached* ***or until*** *arc (s_i, a_j) leads to a terminal node s_j*
    if *arc (s_i, a_j) has no child node* **then** `// Expansion`
        Add a child node s_j to s_i's arc (s_i, a_j)
    end
    if *s_j is a terminal node* **then** `// Simulation`
        Initialise s_j value using the terminal reward
    else
        Initialise s_j using the value estimation of the DNN
        Initialise s_j's arcs (s_j, a_k) with $a_k \in A(s_j)$ using the policy estimation of the DNN
    end
    for *each arc (s_i, a_j) visited during this iteration* **do** `// Backpropagation`
        Update (s_i, a_j)'s visit count $N(s_i, a_j)$ and total action value $W(s_i, a_j)$
    end
end

This ensures that despite a high degree of exploitation introduced by small τ, all action may be explored, but the search may still overrule bad actions. [44, 46]

Algorithm 3.2 describes AZ's decision-making in each turn.

Algorithm 3.2: AlphaZero's decision-making algorithm

Input: g - a game in state r
t - the current turn since the start of g
τ - the temperature-function
dn - the dirichlet-noise-function

1 Look-up node s_{r} representing state r
2 Apply exploration noise $\mathrm{dn}(s_{\mathrm{r}}, a_i)$ to the prior probabilities $P(s_{\mathrm{r}}, a_i)$ of s_{r}'s arcs with $a_i \in A(s_{\mathrm{r}})$
3 Perform Algorithm 3.1 with root node s_{r}
4 Get the policy $\mathbf{P}_{\mathrm{r}} = \pi(s_{\mathrm{r}}, \mathrm{T})$ with temperature $\mathrm{T} = \tau(\mathrm{t})$ applied
5 **return** $\mathbf{P}_{\mathrm{r}}$

Learning AlphaZero's neural network parameter updates and its self-plays are running independently and in parallel. Algorithm 3.3 describes the process of generating training data during self-play.

The neural network's parameters θ are initialised with random values before RL starts, and the memory buffer is initially empty. The network parameters are continually updated in each training steps u[5], using a batch of samples b_u (drawn uniformly at random)[6] from the current memory buffer[7].

Every 1,000th training step, a checkpoint c is reached, and the most recent network f_{θ_c} is saved and from then on used within self-play. Every game that is being generated during SP using the MCTS algorithm with the neural network f_{θ_c} is continually pushed to the memory buffer as a game-trace. A game-trace is the game's state- and action-history s_t and π_t respectively, and the outcome of the game z for all $t \in [0, \mathrm{T}]$, i.e. from the initial state s_0 to the terminal state s_{T}.

In every training step u, the current network's parameters θ_{u-1} are being updated to minimise the value prediction error and maximise the policy similarity over a batch b_u of triples (s, π, z) using stochastic gradient descent (with momentum[8] m and the learning-rate lr) on the loss function l. The learning-rate was decreased three times during training from 0.2 down to 0.0002. Equation (3.5) shows the loss function l, which

[5]Silver et al. use 700,000 training steps (0< u ≤700,000).
[6]Silver et al. use a batch size of $|b_u|$=4,096.
[7]Silver et al. use a memory buffer size of 10^6.
[8]Silver et al. use the momentum $m = 0.9$ for chess.

Algorithm 3.3: AlphaZero's self-play algorithm

```
Input: n_sp - the number of self-play games
Initialise player p_c with the current neural network checkpoint f_θc
for n_sp iterations do
    Initialise player p_u with the most recent neural network f_θu
    Start a new game g between p_u and p_c
    repeat
        Perform search on g using the current player according to Algorithm 3.1
        Obtain the policy P for the current player according to Algorithm 3.2
        Record the state of g and the policy P
        Sample an action a according to the probabilities P
        Apply action a to game g
        Record the current reward on g // 0 if g is in a non-terminal
            state, otherwise +1, 0 (draw), or −1
    until g has terminated
    Generate training samples from the recorded game data
    Update the memory buffer with the most recent samples
end
```

is the sum of the mean-squared-error of the value v and the entropy loss of the policy $\mathbf{p}$, plus the L_2 loss of the weights θ, multiplied with a constant weight decay[9] wd with wd << 1, where $(\mathbf{p}, v) = f_\theta(s)$.

$$l(\theta, b) = \sum_{(s,\pi,z)\in b} [(z - v)^2 + \boldsymbol{\pi}^T \cdot \log \mathbf{p}] + \mathrm{wd} \cdot \parallel \theta \parallel_2^2 \tag{3.5}$$

Algorithm 3.4 describes the learning process for the DNN on the memory buffer, populated by SP using Algorithm 3.3.

[9]Silver et al. use a weight decay factor of $\mathrm{wd} = 10^{-4}$.

Algorithm 3.4: AlphaZero's learning algorithm

Input: n_{learn} - the overall number of training steps
d_{c} - the number of network update steps, after which the neural network, used in SP, is replaced

Initialise the neural network f's weights θ_c randomly and set $\theta_u = \theta_c$
for n_{learn} *iterations* **do**
    Sample a batch b_u from the memory buffer uniformly at random
    Evaluate b_u using f_{θ_u}
    Calculate the loss on the data from b_u and f_{θ_u}'s estimation
    Perform stochastic gradient descent to optimise θ_u according to the loss
    if *every* d_c*-th update step* **then**
        Set $\theta_c = \theta_u$
    end
end

Neural network The DNN is a convolutional residual neural network based on the residual network architecture. It consists of a **trunk**, which receives the board state as input, and two heads. One head is generating the value output v (**value head**) and the other one is generating the policy output vector $\mathbf{p}$ (**policy head**). The high-level network architecture is depicted in Figure 3.8.

The trunk is composed of one convolutional block CB_t followed by a series of 19 residual blocks RB_i. The initial convolutional block CB_t is depicted in Figure 3.9. It consists of a convolutional layer with a 3×3-kernel, a stride of 1 and 256 filters, followed by batch normalisation and a rectifier nonlinearity [46, 47].

Each residual block RB_i contains a convolutional layer (3×3-kernel, stride 1, 256 filters) and a batch normalisation, followed by a rectifier nonlinearity, followed by another convolutional layer (3×3-kernel, stride 1, 256 filters) and batch normalisation, followed by an addition operation, that adds the input of the residual block, and finally a last rectifier nonlinearity [46, 47]. Figure 3.10 shows on residual block.

The output of the trunk is the input for both, the value head and the policy head. The value head consists of a convolutional block CB_v which is depicted in Figure 3.11a, made up of a convolutional layer (1×1-kernel and 1 filter), followed by batch normalisation and a rectifier nonlinearity. The convolution with a 1×1-kernel reduces the dimensionality of feature planes [34]. CB_v is followed by a fully connected linear layer to a hidden layer with size 256, a rectifier nonlinearity, another fully connected linear layer to a scalar (F_v) and a final tanh-nonlinearity that outputs the value v (see Figure 3.8) [46, 47].

The policy head consists of a convolutional block CB_p which is depicted in Figure 3.11b, made up of a convolutional layer (1×1-kernel and 2 filters), followed by batch normalisation

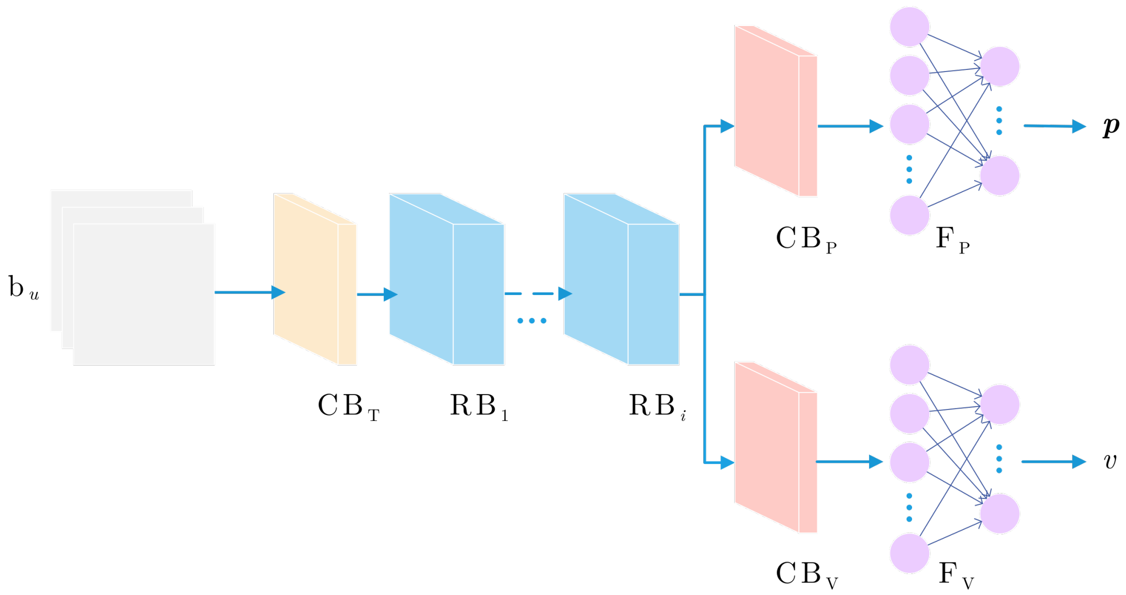


Figure 3.8: The architecture of AZ's DNN. Input is the batch b_u on the left. The initial convolutional block CB_t is followed by a series of residual blocks RB_1 to RB_{19}. Afterwards, the data is duplicated and put into two heads for value and policy. Both of them with convolutions (CB_v and CB_p) at the start and followed up with a fully connected network (F_v and F_p). The value head output is generated by applying a tanh-function and the policy head output is generated by F_p.

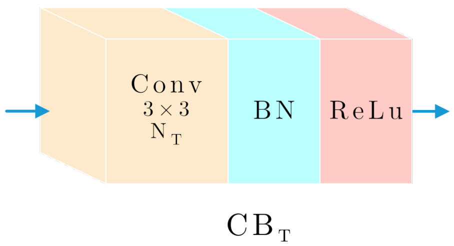


Figure 3.9: The initial convolutional block of the trunk part of the network consists of a convolutional layer with 256 filters (N_T), batch normalisation and a rectifier nonlinearity.

and a rectifier nonlinearity [46, 47]. CB_p is followed by a fully connected linear layer F_p, that outputs the policy vector **p** (see Figure 3.8) [46, 47]. The value head for chess and shogi is different, i.e. instead of F_p, it contains another a convolutional layer. Silver et al. [47] however, mention that the reason was only shorter training times and not agent performance.

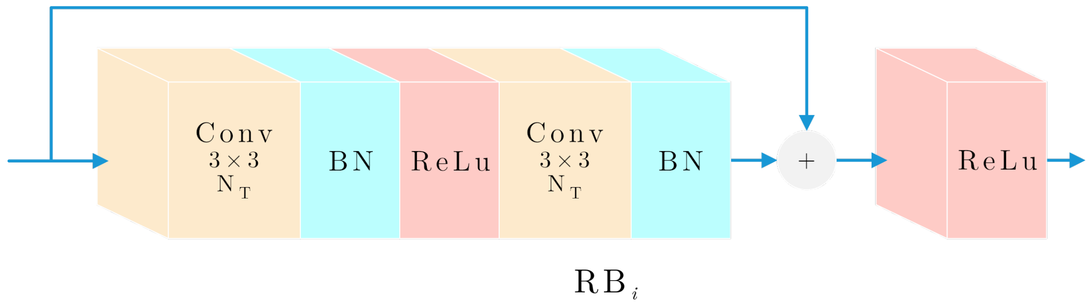


Figure 3.10: Each residual block in the trunk part of the network consists of two convolutional layers with 256 filters (N_T), batch normalisation and rectifier nonlinearities. The input additively combined with the output of the convolutional layers, followed by a rectifier nonlinearity.

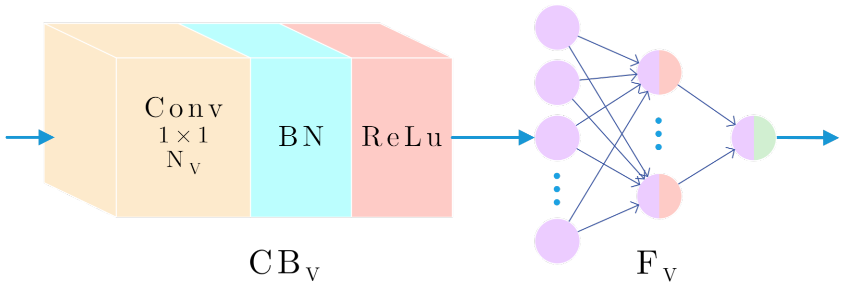
Neural Network Inputs and Outputs for Go The network input, which is based on the state of the game, is a tensor of 17 binary feature planes of the board size (19×19). The first 16 planes contain the positions of the players' stones for the past 8 turns, starting with the most recent turn. The planes contain the value 1 if the corresponding position on the board has a stone of the one player, and the value 0 if the other player or no player has a stone on the position. A subsequent plane then contains the stone positions from the perspective of the other player. The player positional planes are interleaved for the past 8 turns. From the beginning of the game until 8 turns into the game, the planes are all 0. The final plane indicates the player, which is about to play, all 1 for the player in black and all 0 for the player in white.

The value output $v \in [-1, +1]$ is a prediction of how likely it is to win ($+1$) or lose (-1) from the current state of the game. The policy output $\mathbf{p}$ in Go is the logit probabilities for all positions on the board and the pass action $|\mathbf{p}| = 19 \times 19 + 1 = 362$.

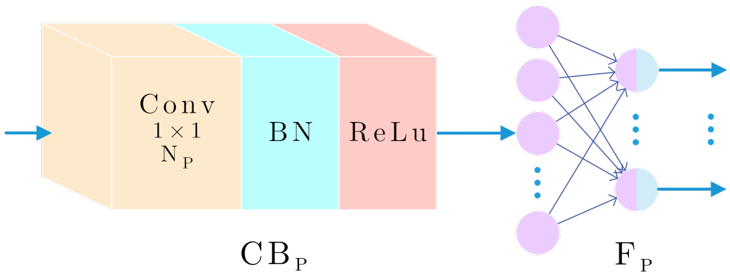
Regular Play and Competition During training, AZ is generating games much faster than during competition/evaluation, i.e. taking much less time for MCTS-iterations. The number of iterations performed by MCTS during training was set to only 800 [44]. During competition, the number of iterations is usually determined by a time control, e.g. a limited total *thinking time*. AlphaZero used $\frac{1}{20}$ of the remaining total thinking time for each turn [44].

Summary of AlphaZero's Hyper-Parameters

> **Dirichlet Noise** The two parameters ϵ and α are controlling the noise that is applied to the prior policy, when selecting the next action to play. Silver et al. chose the value of ϵ inverse proportion to the approximate number of legal moves in a typical position (0.3 for chess).



(a)



(b)

Figure 3.11: The head parts of the network consist of a convolution with a kernel size of 1×1, batch normalisation and rectifier nonlinearities. The value head (a), applies one filter ($\mathrm{N_V}$) and the policy head (b) two ($\mathrm{N_P}$).

Exploration Rate The parameter $C(s)$ in the tree search controls the exploration/exploitation when selecting actions during MCTS-iterations.

Temperature schedule The schedule $\tau(t)$ that depends on the turn.

MCTS-iterations The number of iterations or "thinking time" per turn and the reward discount factor γ.

Network and Learning The number of training steps (weight updates), the number of self-play games, the checkpoint interval (agent-network update during SP), weight-update batch size, memory size (or memory size schedule), batch normalisation momentum, the gradient descent momentum, the learning-rate schedule, the weight decay and the sample weight policy ω.

3.3 ISMCTS

In a scenario with imperfect information, classical tree search cannot be used out-of-the-box. A simple but still successful solution is determinization [19, 4]. Determinization uses the same algorithm as for perfect information games on determined instances of a game with imperfect information. That is, making an assumption about the hidden information and using the resulting perfect information instance instead.

Frank and Basin [16] identify two main issues of determinization:

> **Strategy fusion** An agent erroneously assumes, it can decide on the right strategy in different states within an information set. This leads to incorrect decisions, as the states within an information set cannot be distinguished from each other.
>
> **Non-locality** During search, only a particular subtree is used to evaluate the payoff of a strategy. This subtree, however, might not be relevant, as the subtree from the actual state is in a different part of the game tree and has a different set of payoffs.

Long et al. [35] identify the following properties of a game tree, which can be used to determine the effectiveness of determinization or conversely the impact of *strategy fusion* and *non-locality*:

> **Leaf Correlation** The probability of sibling terminal nodes having the same payoff. A low leaf correlation makes it difficult for a player to affect their payoff.
>
> **Bias** The probability of a game, favouring one player over the other. With very high bias, the search space is expected to have large, homogeneous sections [35].
>
> **Disambiguation Factor** The rate, w.r.t the depth of the game tree, at which the number of nodes in a player's information set decreases. The disambiguation factor might be high in games, where hidden information is gradually revealed with each turn.

All of these will be present in J4F to a certain degree. *Leaf correlation* is likely to be present in longer games that end in points, with one player having many more points than the other player. In games in which one player manages to start a pattern early, might also have many correlated leafs. As we will see later in the results chapter, since the starting player appears to have an advantage, *bias* is present to some degree. The *disambiguation factor* is expected to be non-negligible as well, since the played cards in each turn narrow down the cards, the opponent can potentially still have or get.

To address the issue of *strategy fusion* and *non-locality*, Cowling et al. [12] proposed a new family of algorithms, **Information Set Monte Carlo Tree Search (ISMCTS)**. ISMCTS is an online search algorithm for games with imperfect information, partially observable moves and chance events. The algorithm is based on MCTS and UCT. It builds a game tree based on information sets instead of actual states. With information sets being the decision nodes, the *multi-armed bandit problem* does not fit well anymore. A better fit is the *subset-armed bandit problem* [12]. In the *subset-armed bandit problem*, only a subset of machines is available in each trial. To account for the different availability of arms, Cowling et al. use a modified version of UCT (Definition 3.3.1). They modify the UCB1 by replacing the overall number of trials n in Definition 2.1.4, i.e., the visit count of a parent node, with the number of trials in which machine i was available, i.e. action i was legal.

Definition 3.3.1 (UCB1-based bandit algorithm proposed by Cowling et al. [12])**.** Play the machine i, that maximises $\overline{x}_i + c \cdot \sqrt{\frac{\ln a_i}{n_i}}$, where $\overline{x}_i$ is the current average reward of the machine i, n_i the number of times the machine i has been played and a_i the number of times, machine i was available.

This modification addresses cases where rare actions would otherwise be over-explored. Rare in this context refers to actions that are only available in few states of an information set. That is, when n_i is still small after numerous trials n, the UCB1 gets disproportionately large. One specific ISMCTS algorithm is the **Single-Observer Information Set MCTS (SO-ISMCTS)**, described in Algorithm 3.5. Over several MCTS-iterations n, SO-ISMCTS samples a concrete state, i.e., a determinization, from an information set. During selection, expansion and simulation, the selected determinization determines the available actions.

Note that Cowling et al. also proposed other algorithms in [12], which address certain issues that may arise from properties of the game at hand. These issues, however, appear less prominent when SO-ISMCTS is used in SP, noise and temperature are applied to the policy, and with a DNN as a reward estimation.

Cowling et al. compare the performance of several agents in different games. A *Cheating UCT* (i.e. search in full knowledge about the hidden state), a *Determinized UCT* (using several trees built from individual determinizations), *Single-Observer Information Set MCTS* and two further ISMCTS variants.

Algorithm 3.5: SO-ISMCTS, as proposed by Cowling et al. [12]

Create the root node v_0 of the search tree, corresponding to the root information set $I_{i,0}$, composed of nodes v_w and arcs (v_x, a_y), where some player z is about to play
for *n iterations* **do**
  Choose a determinization d from $I_{i,0}$ at random
  repeat `// Selection`
    Descend the tree following arcs according to d and the modified UCB1
  until *A node v is reached, that leads to an information set $I_{i,v}$ which is not in the tree yet* ***or until*** *v is a terminal node*
  if *v is non-terminal* **then** `// Expansion`
    Choose at random an action from node v according to d
    Add a child node c that is corresponding to the information set $I_{i,c}$ reached using the action and set it as the new current node v
  end
  `// Simulation`
  Run a playout from v to the end of the game using determinization d
  `// Backpropagation`
  for *each node u visited during this iteration* **do**
    Update u's visit count and total simulation reward
    for *each sibling w of u, that was available for selection when u was selected, including itself* **do**
      Update w's availability count
    end
  end
end
return *an action a from the root node v_0 such that the number of visits to the corresponding child node is maximal*

CHAPTER 4

AlphaJust4Fun

This chapter describes the **AlphaJust4Fun (AZJ4F)** algorithm. AZJ4F takes the AlphaZero framework and replaces its tree search with Information Set Monte Carlo Tree Search.

The proposed agent addresses the hidden information (i.e. hidden cards on the stack and in the opponent's hand) and randomness (i.e. recreating the stack of cards by shuffling the pile of used cards) in J4F by changing the planning algorithm of AlphaZero.

More specifically, AZJ4F uses the Single-Observer Information Set MCTS [12] that is described in Section 3.3.

Stochastic aspects of Just 4 Fun The randomness that is introduced when the stack is empty, and the pile of used cards is shuffled and used as the new stack, is simplified as another piece of hidden information. It is in a sense considered as (a possibly endless) order of hidden cards, with some patterns of periodicity, as the reshuffling is treated as part of a players' turn, not as a dedicated chance node in the information set tree, and thus handled by the information set nodes (in combination with the determinization).

At the beginning of a game, the player's knowledge about the hidden cards is as a probability distribution over the unknown cards. With each turn, cards are put onto the pile of used cards, which makes the remaining hidden cards more predictable. Once the stack of cards is empty, the probability distribution is reset to one that is closer to the initial one.

These patterns of increasing and decreasing certainty, or increasing and decreasing size of information sets, are handled by the information set tree search and the learned, discounted terminal rewards.

Hidden information in Just 4 Fun We distinguish the following sets of information among the game-state:

Player-cards-state Contains the player's own cards which are only visible to the player they belong to.

Public cards-state Contains the pile of used cards, visible to all players.

Board-state Contains the number of stones of each player on each field.

Full cards-state Contains the player's own cards, the pile of used cards, the (hidden) stack of cards and the (hidden) opponent's cards.

Full game-state Contains the board-state and the full cards-state.

Information set key-state The part of the full game-state that is common to all possible full game-states within an Information Set.

The set of hidden information not only includes the opponent's cards, but also the (inexhaustible) stack of cards. In practical scenarios, when both players consistently play their entire hand and the game reaches its maximum duration, the stack will be shuffled twice at maximum. The set of known information, in a game in state s, where Player i is about to play, is $\mathrm{K}_s^i = \mathrm{BS}_s \cup \mathrm{UC}_s \cup \mathrm{PC}_s^i$, where BS_s is the board-state, UC_s is the set of already played cards and PC_s^i player i's own hand. The set of hidden information is $\mathrm{H}_s^i = \mathrm{DC} \setminus (\mathrm{UC}_s \cup \mathrm{PC}_s^i)$, where DC is the deck of cards. H_s^i can be decomposed into $\mathrm{H}_s^i = \mathrm{SC}_s \cup \mathrm{PC}_s^o$, with two unknown components SC_s and PC_s^o, the current stack of cards, and the opponent's o hand respectively.

Information Set There are two factors that influence the size of an information set. First, the nature of the deck of cards, i.e. having some cards occur four times, while others only once. The redundancy is introduced by the fact, that the order of cards in the opponent's hand is not relevant.

The second, greater influence, is due to the increasing set of revealed cards during play. With each action, a set of one to four cards is added to a set UC_s, which in turn increases the set K_s^i. As a result, the size of H_s^i continually decreases until it is empty. When that happens, the stack of cards is recreated by shuffling the pile of used cards to form a new stack. The follow-up stack is then composed of $\mathrm{DC} \setminus (\mathrm{PC}_s^o \cup \mathrm{PC}_s^i)$.

The information set for a game in state s, where Player i is about to play, is denoted by I_s^i and shown in Equation (4.1), where $D(\mathrm{H}_s^i)$ is the set of all permutations of H_s^i.

$$\mathrm{I}_s^i = \mathrm{BS}_s \times \left(\bigcup_{d_j \in D(\mathrm{H}_s^i)} \mathrm{K}_s^i \cup d_j \right) \tag{4.1}$$

As Cowling et al. noted, it might not be optimal to use a different determinization for every MCTS-iteration [12]. They investigate the effect of the balance between the number of MCTS-iterations and the number of determinizations in two different games. For one of the games[1], they report that this trade-off has little effect on the agent strength, given both, the number of MCTS-iterations and the number of determinizations, are sufficiently large. For the other game[2], agent strength decreases with an increasing number of determinizations. They conclude that the effect depends on the characteristics of the game. When long-term planning is required, a smaller number of determinizations (for a given number of MCTS-iterations) is beneficial, as it increases the average search depth.

The determinizations d_j of a state s, where player i is about to play, are defined by Equation (4.2), where sb is the MCTS-iteration-budget and nd the chosen number of determinizations.

$$d_j \in D(\mathrm{H}_s^i) \mid_{j \leq \lfloor \frac{\mathrm{sb}}{\mathrm{nd}} \rceil} \tag{4.2}$$

Monte Carlo Tree Search The tree search is performed by iteratively extending the tree of information sets, using a modified version of AlphaZero's tree search policy (see Equation (3.1)), on sampled determinizations. The selection policy uses, similar to SO-ISMCTS, the actions' availability count $N_a(s,a)$ instead of the node's visit count $N(s,a)$ at a node s with actions a. $N_a(s,a)$ is the number of times in which s has been visited and a has been available from s, i.e. $a \in A^d(s)$. As a result, rarely available actions are not disproportionally often explored during search. The selection policy is the same as in AZ (Equation (3.1)), but uses the upper confidence bound shown in Equation (4.3).

$$U(s,a) = C(s) \cdot P(s,a) \cdot \frac{\sqrt{N_a(s,a)}}{1 + N(s,a)} \tag{4.3}$$

In a backwards pass, for all the taken arcs a_t from the visited nodes s_t, also the availability counts, i.e. in addition to the arc-data maintained by AZ that are described in Equation (3.3), under a determinization d, are updated:

$$N_a(s_t, a_t) = N_a(s_t, a_t) + 1, \forall a_t \in A^d(s_t)$$

The total action value is updated using the discounted value of the node s_{next}, which a_t lead to, either the terminal reward $r^{s_{\text{next}}}$ or the value estimated by the DNN f_θ:

$$W(s_t, a_t) = W(s_t, a_t) + \gamma \cdot \begin{cases} r^{s_{\text{next}}}, & \text{if } s_{\text{next}} \text{ is terminal} \\ v^{s_{\text{next}}}, & \text{if } s_{\text{next}} \text{ is non-terminal, with } v^{s_{\text{next}}}, \boldsymbol{p}^{s_{\text{next}}} = f_\theta(s_{\text{next}}) \end{cases}$$

Algorithm 4.1 contains the pseudocode for AlphaJust4Fun's MCTS algorithm on a high level, the detailed algorithm can be found in Appendix B (Algorithm B.1).

[1] Dou dizhu

[2] Lord of the Rings: The Confrontation

Algorithm 4.1: AlphaJust4Fun's MCTS algorithm

Input: s_0 - the a root node of a game subtree, composed of nodes s_w and arcs (s_x, a_y), where some player z is about to play and which corresponds to the root information set I_0^i of that subtree
n_{iter} - the number of MCTS-iterations
n_{det} - the number of determinizations
γ - the value discount factor

for *n_{iter} iterations* **do**
    if *first iteration **OR** every $\frac{n_{iter}}{n_{det}}$-th iteration* **then**
        Select determinization d_j from $D(\mathrm{H}_0^i)$ at random
    end
    Start from the root node by assigning $s_k = s_0$
    repeat `// Selection`
        Descend the subtree by selecting arcs (s_k, a_l) with $a_l \in A^{d_j}(s_k)$, that are available from s_k under determinization d_j, and maximise the upper confidence bound $Q(s_k, a_l) + U(s_k, a_l)$ `// see Eq. (3.1) with` $U(s, a)$ `from Eq. (4.3)`
    until *an arcs (s_k, a_l) is reached that leads to a node (corresponding to an information set), which is not in the tree yet **or until** arc (s_k, a_l) leads to a terminal node*
    if *arc (s_k, a_l) leads to a node, which is not in the tree yet* **then** `// Expansion`
        Add a child node s_l to s_k's arc (s_k, a_l), that is corresponding to the information set $\mathrm{I}_{s_l}^i$
    end
    if *s_l is a terminal node under d_j* **then** `// Simulation`
        Initialise s_l value using the terminal reward
    else
        Initialise s_l using the value estimation of the DNN
        Initialise s_l's arcs (s_l, a_m) with $a_m \in A^{d_j}(s_l)$ using the policy estimation of the DNN
    end
    for *each arc (s_k, a_l) visited during this iteration* **do** `// Backpropagation`
        Update (s_k, a_l)'s visit count $N(s_k, a_l)$ and total action value $W(s_k, a_l)$ with the discounted value of the node, (s_k, a_l) lead to.
        for *each sibling (s_k, a_m) with $(s_k, a_m) \in A^{d_j}(s_k)$, that was available for selection when (s_k, a_l) was selected, including itself* **do**
            Update (s_k, a_m)'s availability count $N_{\mathrm{a}}(s_k, a_m)$
        end
    end
end

AZJ4F's decision-making during play is similar to the one of AlphaZero and shown in Algorithm 4.2.

Algorithm 4.2: AlphaJust4Fun's decision-making algorithm

Input: g - a game in state r
t - the current turn since the start of g
τ - the temperature-function
dn - the dirichlet-noise-function

1. Look-up node s_r (representing the information set $\mathrm{I}^i_{s_\mathrm{r}}$ for the AZJ4F player i) using the information set key-state (e.g. $\mathrm{K}^i_{s_\mathrm{r}}$)
2. Apply exploration noise $\mathrm{dn}(s_\mathrm{r}, a_j)$ to the prior probabilities $P(s_\mathrm{r}, a_j)$ of s_r's arcs with $a_j \in A(s_\mathrm{r})$
3. Perform Algorithm 4.1 with root node s_r
4. Get the policy $\mathbf{P}_\mathrm{r} = \pi(s_\mathrm{r}, \mathrm{T})$ with temperature $\mathrm{T} = \tau(\mathrm{t})$ applied
5. **return** $\mathbf{P}_\mathrm{r}$

Learning Training samples are generated during self-play. If SP is performed using a perfect simulator, i.e. in full knowledge of hidden information, then AZ's tree search is used as-is. If performed on an imperfect simulator, i.e. only using the known information, then AZJ4F's tree search is used. However, using a perfect simulator may lead to strategy fusion (see Section 3.3) having a bigger impact. In situations where the decision on the next action in a perfect information scenario is obvious, it might lead to the network learning an overconfident policy w.r.t. the cards the agent will get next. With vast amounts of training data, this might be circumvented, but it might introduce effects of overfitting for the value head.

Algorithm 4.3 describes the process of generating training data during self-play.

The learning process for the DNN on the memory buffer, populated by SP using Algorithm 4.3, works mostly similar to the one in AZ and is described in Algorithm 4.4. The difference is, that the discounted outcome z' is used:

$$\begin{aligned} z'_\mathrm{T} &= z_\mathrm{T} \\ z'_{t-1} &= \gamma \cdot z'_t \text{ for all } t \in [0, \mathrm{T}] \end{aligned}$$

Another difference is that the reward and the policy of samples in the replay buffer, that represent the same game state, are averaged using the arithmetic mean. All samples are then weighted with some factor w_i, according to the sample-weight policy $\omega(n_i)$, e.g. $w_i = \omega(n_i) = \log_2(n_i) + 1$, with n_i being the number of duplicates of a sample i. Very common samples, e.g. from the beginning of the games, are thus drawn with a probability inverse proportional to their number of duplicates.

Algorithm 4.3: AlphaJust4Fun's self-play algorithm

```
Input: n_sp - the number of self-play games
Initialise player p_c with the current neural network checkpoint f_θc
for n_sp iterations do
    Initialise player p_u with the most recent neural network f_θu
    Start a new game g between p_u and p_c
    repeat
        Perform search on g using the current player according to Algorithm 4.1
         (and Alg. B.1)
        Obtain the policy P for the current player according to Algorithm 4.2
        Record the state of g and the policy P
        Sample an action a according to the probabilities P
        Apply action a to game g
        Record the current reward on g // 0 if g is in a non-terminal
            state, otherwise +1, 0 (draw), or −1
    until g has terminated
    Generate training samples from the recorded game data
    Update the memory buffer with the most recent samples
end
```

AlphaJust4Fun uses the same loss function as AZ (see Equation (3.5)), but the network's parameters are updated using the **Adam** algorithm [28].

Neural Network The DNN's input is based on the known information of an information set I_s^i from the perspective of a player i in a state s. The output is, similar to AlphaZero, the value and policy estimation. In Chapter 5, we present several candidate network architectures.

Neural Network Inputs and Outputs for Just 4 Fun When comparing Just 4 Fun to Go, chess and shogi, part of J4F's game-state is similar in type. The board-state, which is known to all players, is mostly similar to Go. The first difference is the possibility of having multiple stones per field. The second one is the limited symmetry, introduced by the different field values. That is, the board state is only symmetric regarding win-patterns. This makes it suitable to be used in an architectural setting similar to AZ, i.e. with convolutional residual blocks, and with similar modelling of network inputs. The part of the state consisting of cards is different in kind and may require a different architectural setting and modelling of inputs. The modelling of network inputs and features is described in Chapter 5, along with the investigated network architectures.

Algorithm 4.4: AlphaJust4Fun's learning algorithm

Input: n_{learn} - the overall number of training steps
d_{c} - the number of network updated steps, after which the neural network used in SP is replaced

Initialise the neural network f's weights θ_c randomly
Set $\theta_u = \theta_c$
for n_{learn} *iterations* **do**
  Sample a new batch b_u from the memory buffer uniformly at random
  Evaluate b_u using f_{θ_u}
  Calculate the loss on the data from b_u and f_{θ_u}'s estimation
  Perform gradient descent to optimise θ_u according to the loss
  if *every* d_c*-th update step* **then**
    Set $\theta_c = \theta_u$
  end
end

The value output is similar to AZ with $v \in [-1, +1]$, $+1$ for *win*, -1 for *loss* and 0 for *draw*. The policy output $\mathbf{p}$ can be modelled similar to AZ in Go, as the logit probabilities for all positions on the board. This requires heuristics to select which cards to play specifically. Further possibilities to model the policy output are, e.g. over all possible card combinations or having separate outputs for board positions and cards. The modelling of network outputs is also described along with the investigated network architectures in Chapter 5.

Regular Play and Competition During self-play, AZJ4F might use a perfect or an imperfect simulator for the tree search. However, during regular play or competition (i.e. in an imperfect simulator), the number of MCTS-iterations and determinizations has a significant impact on agent performance and generally needs to be higher than during training.

Summary of AlphaJust4Fun's Hyper-Parameters

MCTS-iterations In addition to the ones of AlphaZero (see Section 3.2, Paragraph 10), AZJ4F adds one further MCTS related hyperparameter, the number of determinizations. It specifies the number of random determinizations to be used throughout MCTS-iterations or "thinking time" in each turn.

CHAPTER 5

Network Architectures and Feature Engineering

This chapter describes two network architectures we investigated, the input features, and initialisation of kernels.

The **FieldNet (FNet)** architecture in Section 5.1 is closely related to AlphaZero's network architecture, with a single trunk and two heads. **CardFieldNet (CFNet)**, which is presented in Section 5.2, has two separate trunks for the board-based information and the information based on cards, and two heads.

A cards-based action space in Just 4 Fun is the set of all combinations of cards (with the order being irrelevant), restricted by firstly the cards in the deck (varying number of duplicates per card value), secondly the field values (i.e. the sum of cards being between 1 and 36) and thirdly the action-size (i.e. 1, 2, 3 or 4 cards). The card action space can be denoted by the set $\mathbf{A}$ in Equations 5.1 where n_h is the size of the players' hand, $\mathbf{FV}$ the set of all field values, $\mathbf{DC}$ the multiset of all cards in the deck and $\mathcal{P}_h$ the powerset of cardinality h.

$$
\begin{aligned}
n_\mathrm{h} &= 4 \\
\mathbf{FV} &= \{v \mid 1 \leq \ v \ \leq \ 36\} \\
\mathbf{DC} &= \{1^4, 2^4, 3^4, 4^4, 5^4, 6^4, 7^4, 8^4, 9^4, 10^4, 11^4, 12^4, 13^1, 14^1, 15^1, 16^1, 17^1, 18^1, 19^1\} \\
\mathbf{A} &= \{ps_i \mid ps_i \in \mathcal{P}_h(\mathbf{DC}), h \in [1, n_\mathrm{h}], \textstyle\sum(ps_i) \leq max(\mathbf{FV})\}
\end{aligned}
\tag{5.1}
$$

We determined the total number of unique and legal actions that can be formed from the deck of cards by constructing, filtering and counting the powersets of cardinality 1 to 4 and found it to be 3,923.

Building the logit probabilities for such a high number of actions might lead to numeric issues. Therefore, we focus on architectures that output a board-based policy (Section 5.1 and Section 5.2), i.e. a policy over the 36 fields.

The following sections contain variables for the number of network layers (e.g. L_P) and the layer sizes (e.g. N_P) thereof. In the following chapters, we will always refer to those variables when describing the actual networks that are being experimented with. For easier understanding, the components in architecture depictions are represented by different shapes and colours as follows:

- Components containing convolutions (rectangular box, “Conv”)
- Components containing skip connections (rectangular box, “RB”)
- Dense neural networks (circle)
- Batch normalisations (circle; rectangular box, “BN”)
- Components with rectifier nonlinearities as activation function (circle; rectangular box, “ReLu”)
- Components with tanh as activation function (circle)
- Components with softmax output (circle)

5.1 FieldNet

FieldNet (FNet) consists of several convolutional residual blocks in the board **trunk**. Both, the **value head** and the **policy head**, are connected to the trunk and consist of a convolutional block and fully connected linear layers. The FieldNet architecture is depicted in Figure 5.1.

Input The input tensor for the FieldNet architecture is inspired by the inputs used in AlphaZero. Its dimensions are given by the board’s dimensions times the number of feature planes. The feature planes are described formally in Section 5.3. They are based on the board-state and a mapping of the public cards-state (see field reachability in Section 5.3) onto the board. The significant difference to the inputs in AlphaZero for Go is the absents of the ply history and the layer that indicates the current player. We consider several sets of feature planes and compare them in Chapter 7.

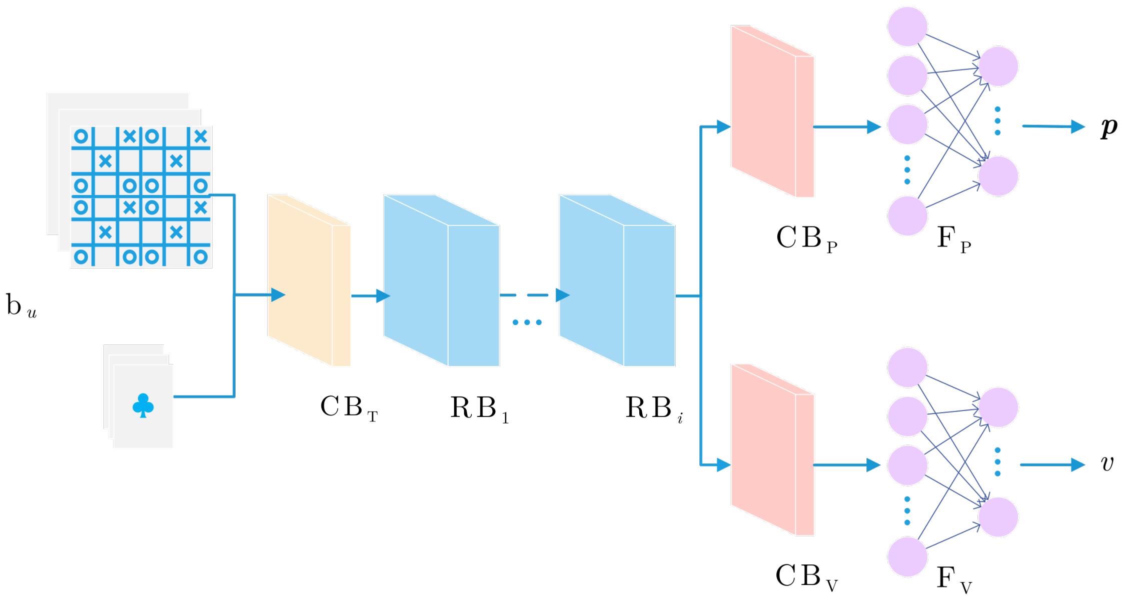


Figure 5.1: Overview of the FieldNet architecture.

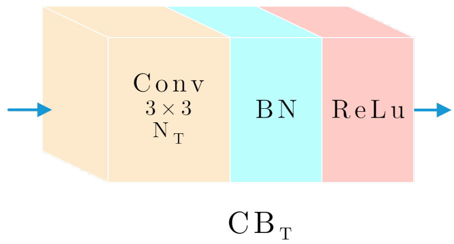


Figure 5.2: The initial convolutional block of FieldNet architecture.

Trunk The trunk is composed of one convolutional block $\mathrm{CB_T}$ followed by a series of i convolutional residual blocks RB_i. The initial convolutional block $\mathrm{CB_T}$ is depicted in Figure 5.2. It consists of a convolutional layer with a 3×3-kernel, a stride of 1 and $\mathrm{N_T}$ filters, followed by batch normalisation and a rectifier nonlinearity.

Each convolutional residual block RB_i contains a convolutional layer (3×3-kernel, stride 1 and $\mathrm{N_T}$ filters) and a batch normalisation, followed by a rectifier nonlinearity, followed by another convolutional layer (3×3-kernel, stride 1 and $\mathrm{N_T}$ filters) and batch normalisation, followed by an addition operation, that adds the input of the convolutional residual block, and finally a last rectifier nonlinearity. A convolutional residual block is shown in Figure 5.3.

The output of the trunk is the input for both, the value head and the policy head.

Value head The value head consists of a convolutional block $\mathrm{CB_V}$ and a fully connected network $\mathrm{F_V}$, which are depicted in Figure 5.4a. $\mathrm{CB_V}$ consists of a convolutional layer (1×1-

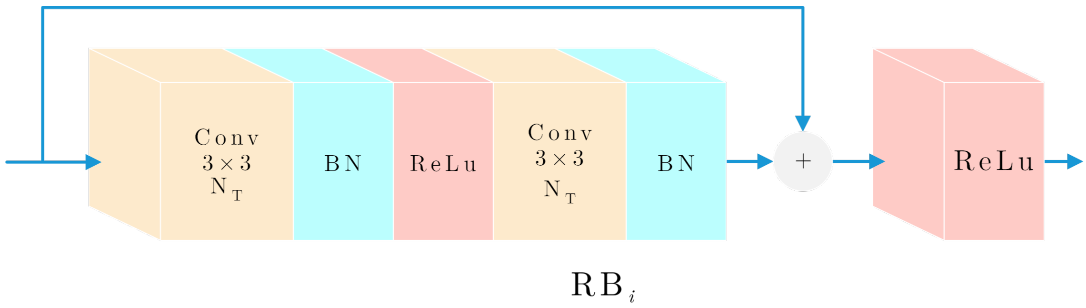


Figure 5.3: A convolutional residual block from the trunk part of the FieldNet architecture.

kernel, stride 1 and N_V filters), followed by batch normalisation and a rectifier nonlinearity. F_V consists of a fully connected layer and a hidden layer (size N_V) with a rectifier nonlinearity, followed by a fully connected layer that returns a scalar value. This value is finally transformed by a tanh-nonlinearity to the value output v (see Figure 5.1).

Policy head The policy head consists of a convolutional block CB_P and a fully connected network F_P, which are depicted in Figure 5.4b. CB_P consists of a convolutional layer (1×1-kernel, stride 1 and N_P filters), followed by batch normalisation and a rectifier nonlinearity. F_P is a fully connected linear layer that returns a vector which is finally transformed by a softmax-function to the policy output vector $\mathbf{p}$ (see Figure 5.1).

Policy output The policy output vector $\mathbf{p}$ of the FieldNet architecture is a probability distribution over the board-based action space, i.e. the 36 fields. This probability distribution is then multiplied with the binary-value mask that indicates the valid actions (i.e. the fields reachable with the player's hand and not secured by any player; see Section 3.1). The mask has a value of 1 on positions of fields that are valid and a value of 0 on the other fields' positions. After masking, $\mathbf{p}$ is re-normalised over the remaining legal moves: $|\mathbf{p}| = 36$ and $\sum_{i=1}^{36} p_i = 1$

Value output The value output v is a real number from the interval $[-1, 1]$, 1 means the current player is likely to win, 0 the game will likely end in a draw and -1 means the current player will likely lose: $v \in [-1, 1]$

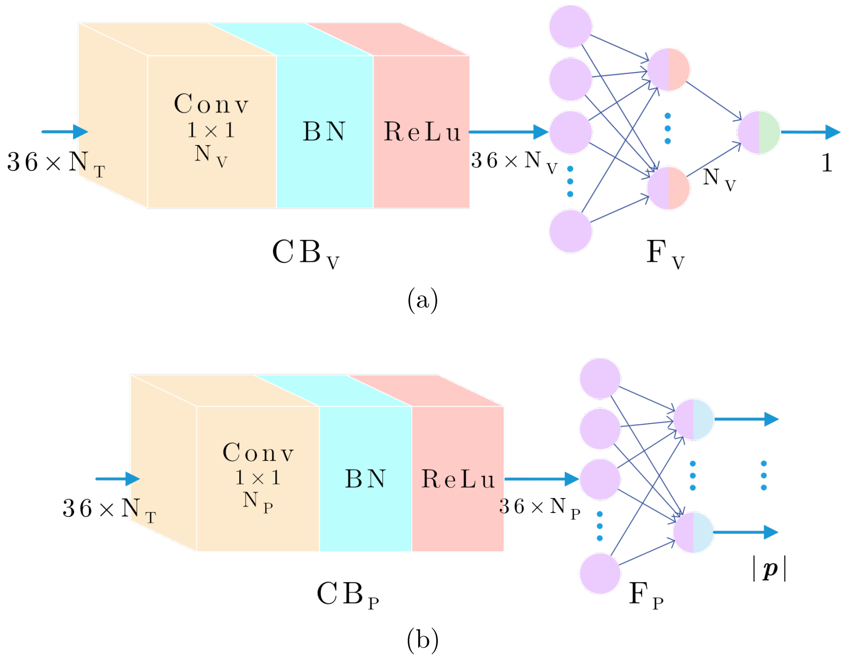


Figure 5.4: The value head (a) and the policy head (b). The input and output dimensionalities are indicated below the directional arrows.

5.2 CardFieldNet

The CardFieldNet (CFNet) architecture consists of two trunks, one **board trunk** and one **cards trunk** (for the public cards-state-based inputs). Similar to the trunk in FieldNet, the board trunk consists of several convolutional residual blocks. The cards trunk consists of a dense neural network. Both trunks are connected by the **common trunk**, which is a dense neural network. The common trunk is followed by the **value head** and the **policy head**. Both consist of dense neural networks. The CardFieldNet architecture is depicted in Figure 5.5.

Inputs The input tensor for the board trunk is similar to the input for FieldNet as described in Section 5.1 on Page 50.

The inputs for the cards trunk are vectors of cards. The feature vectors for the cards trunk are described formally in Section 5.3.

Board trunk The board trunk is composed of one convolutional block $\mathrm{CB_S}$ followed by a series of convolutional residual blocks RB_i, another convolutional block $\mathrm{CB_T}$, and a fully connected linear layer $\mathrm{F_{BT}}$. The initial convolutional block $\mathrm{CB_S}$ is the same as FieldNet architecture (see $\mathrm{CB_T}$ in Figure 5.2), with the number of filters being $\mathrm{N_S}$. The blocks RB_i are also similar to the ones in the FieldNet architecture and shown

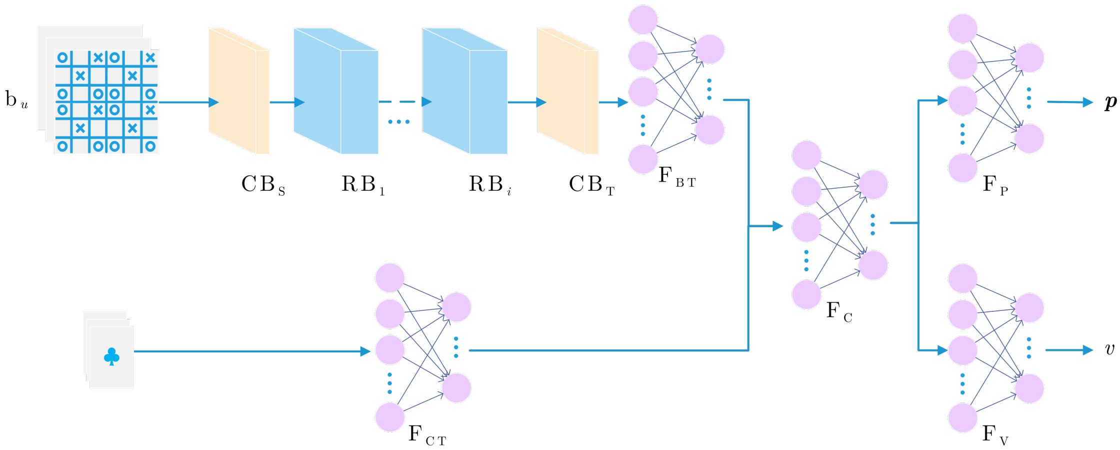


Figure 5.5: Overview of the CardFieldNet architecture.

in Figure 5.3. The final convolutional block $\mathrm{CB_T}$ consists of a convolutional layer with a 1×1-kernel, a stride of 1 and $\mathrm{N_T}$ filters, followed by batch normalisation and a rectifier nonlinearity. After $\mathrm{CB_T}$ follows the fully connected linear layer $\mathrm{F_{BT}}$, which transforms the data from $36 \times \mathrm{N_T}$ (with 36 being the size of the board) to a vector output of size $\mathrm{N_C}$. $\mathrm{CB_T}$ and $\mathrm{F_{BT}}$ are depicted in Figure 5.6. Unless otherwise mentioned, for all dense neural networks F_j, the constants N_j refer to the number of neurons and the constants L_j refer to the number of layers.

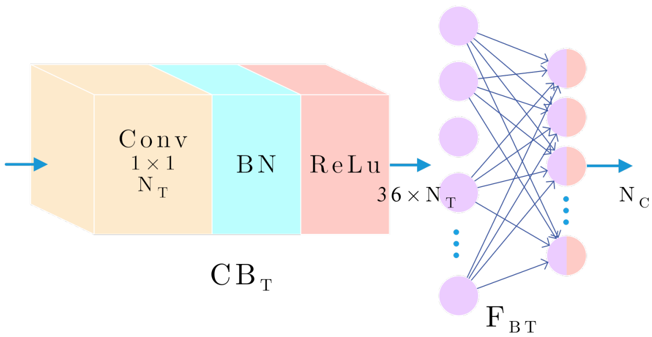


Figure 5.6: The final convolutional block and the fully connected linear layer at the end of the board trunk, which brings board trunk output to a common dimension.

Cards trunk The cards trunk is the dense neural network $\mathrm{F_{CT}}$, which is depicted in Figure 5.7. It consists of $\mathrm{L_{CT}}$ hidden layers of size $\mathrm{N_{CT}}$. The first layer converts from the input size $|\mathrm{I}|$ to size $\mathrm{N_{CT}}$. Each hidden layer incorporates batch normalisation and a nonlinear activation function. The last layer converts from size $\mathrm{N_{CT}}$ to size $\mathrm{N_C}$ and also uses a rectifier nonlinearity as the activation function.

Common trunk Figure 5.6 displays the common trunk F_C which takes the combined output of both, the cards trunk and the board trunk as an input. It consists of a dense network with L_C hidden layers of size N_C. The first layer converts the data from size $2 \times N_C$ to size N_C. Each hidden layer and the output layer incorporate batch normalisation and a nonlinear activation function.

The output of the common trunk is the input of the value head and the policy head.

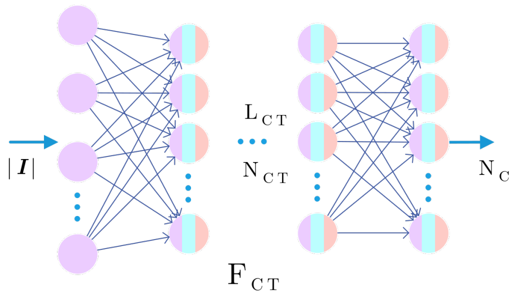


Figure 5.7: The cards trunk of the CardFieldNet architecture.

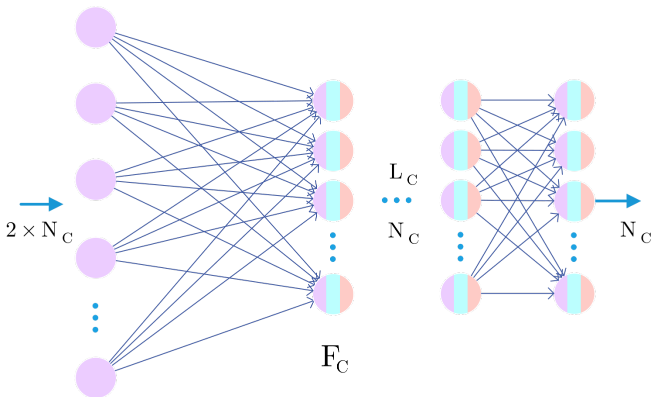


Figure 5.8: The common trunk of the CardFieldNet architecture.

Policy head The policy head is depicted in Figure 5.9, it consists of L_P hidden layers of size N_P. The first layer converts the data from size N_C to size N_P. Each hidden layer and the output layer incorporate batch normalisation and a nonlinear activation function. The last layer has the size of the policy (i.e. 36 for J4F) and is followed by a softmax function.

Value head The value head is depicted in Figure 5.10, it consists of L_V hidden layers of size N_V. The first layer converts the data from size N_C to size N_V. Each hidden layer incorporates batch normalisation and a nonlinear activation function.

The last layer outputs a scalar value and uses the *tanh* activation function.

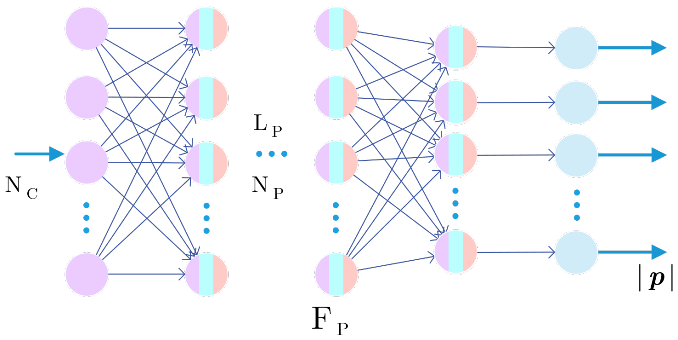


Figure 5.9: The policy head of the CardFieldNet architecture.

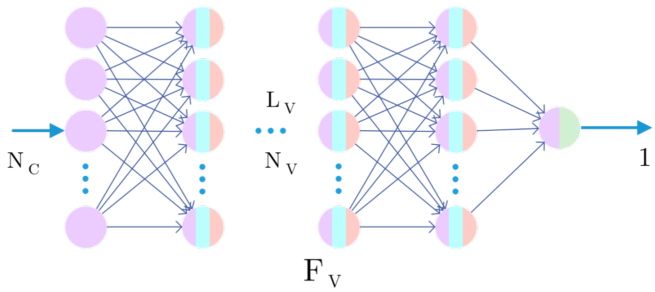


Figure 5.10: The value head (b) of the CardFieldNet architecture.

Policy output and value output The policy output vector $\mathbf{p}$ and the value output v are similar to the ones of the FieldNet architecture and described at the end of Section 5.1.

5.3 Input Features

In this section, we present the input feature planes (available for FieldNet and CardFieldNet) and feature vectors (CardFieldNet only), which we used in our experiments. We start off by defining the matrix, representing the positions of field values, and the vector of card values. Both will be referred to when the specific input feature planes and vectors are defined. For feature planes that have different values for each player, we will define them for some player $c \in \{1, \ldots, n_\mathrm{p}\}$. All the input features are in dependence on some state s (except for the field values $\mathbf{F}$), for the sake of readability, we omitted the state variable. The input tensors used in our experiments are compositions of those feature planes.

Let $\mathbf{F} = [f_{ij}]$ in Equation (5.2) be the matrix representing the field values on the Just 4 Fun board.

$$\mathbf{F} = \begin{bmatrix} 1 & 14 & 30 & 24 & 19 & 8 \\ 33 & 11 & 9 & 16 & 35 & 21 \\ 6 & 27 & 31 & 20 & 3 & 12 \\ 15 & 32 & 5 & 29 & 17 & 26 \\ 22 & 10 & 18 & 36 & 25 & 2 \\ 28 & 7 & 23 & 4 & 13 & 34 \end{bmatrix} \tag{5.2}$$

$\mathbf{c_d}$ in Equation (5.3) is an ordered vector that represents all the cards from the deck DC, and $\mathbf{c_v}$ in Equation (5.4) is a vector that represents the corresponding card values.

$$\mathbf{c_d} = (c_i)_{1 \leq i \leq 55} \tag{5.3}$$

$$\begin{aligned} \mathbf{c_v} = (&1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,\\ &6,6,6,6,7,7,7,7,8,8,8,8,9,9,9,9,10,10,10,10,\\ &11,11,11,11,12,12,12,12,13,14,15,16,17,18,19) \end{aligned} \tag{5.4}$$

Let $\mathbf{B}^c$ in Equation (5.5) be the matrix representing the number of stones on each field (i, j) of the board for some player $c \in \{1, \dots, n_\text{p}\}$, with n_p being the number of players.

$$\mathbf{B}^c = [b^c_{ij}]_{1 \leq i,j \leq 6} \tag{5.5}$$

b^c_{ij} represents the number of stones of Player c on the field with coordinates i and j and value $\mathbf{F}(i, j)$.

Number of stones of a player The most basic input feature is the number of stones on each field for a certain player. The number of stones as feature planes can be used with the FNet architecture and for the board trunk of the CFNet architecture.

For some player c, it is the 6×6 matrix $\mathbf{I_{stones}}^c = [m^c_{ij}]_{1 \leq i,j \leq 6}$ shown in Equation (5.6), where m^c_{ij} is the number of stones of Player c on the field with indices i and j.

$$\mathbf{I_{stones}}^c = \mathbf{B}^c \tag{5.6}$$

Empty field The binary plane indicates the fields, where none of the players has any stones. It can be used with the FNet architecture and for the board trunk of the CFNet architecture.

It is a 6×6 matrix $\mathbf{I_{empty}} = [m_{ij}]_{1 \leq i,j \leq 6}$ shown in Equation (5.7), where m_{ij} is 1 if the field with indices i and j has no stones.

$$\mathbf{I_{empty}} = [m_{ij}] = \begin{cases} 1, & \text{if } \forall p \in \{p \mid 1 \leq p \leq n_\text{p}\} : \mathbf{B}^p(i, j) = 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.7}$$

Minority of stones for a player This is a binary plane that indicates the fields on which a certain player has a minority. It can be used with the FNet architecture and for the board trunk of the CFNet architecture.

For some player c, it is the 6×6 matrix $\mathbf{I_{minority}}^c = [m_{ij}^c]_{1 \leq i,j \leq 6}$ shown in Equation (5.8), where m_{ij}^c equals 1, if there exists another player, which has more stones than Player c on the field with indices i and j, and 0 otherwise.

$$\mathbf{I_{minority}}^c = [m_{ij}^c] = \begin{cases} 1, & \text{if } \exists p \in \{p \mid 1 \leq p \leq n_\mathrm{p} \text{ and } p \neq c\} : \mathbf{B}^p(i,j) > \mathbf{B}^c(i,j) \\ 0, & \text{otherwise} \end{cases} \tag{5.8}$$

Majority of stones for a player This is a binary plane that indicates the fields on which a certain player has the majority of stones. It can be used with the FNet architecture and for the board trunk of the CFNet architecture.

For some player c, it is the 6×6 matrix $\mathbf{I_{majority}}^c = [m_{ij}^c]_{1 \leq i,j \leq 6}$ shown in Equation (5.9), where m_{ij}^c equals 1, if Player c has more stones than any other player on the field with indices i and j, and 0 otherwise.

$$\mathbf{I_{majority}}^c = [m_{ij}^c] = \begin{cases} 1, & \text{if } \forall p \in \{p \mid 1 \leq p \leq n_\mathrm{p} \text{ and } p \neq c\} : \mathbf{B}^c(i,j) > \mathbf{B}^p(i,j) \\ 0, & \text{otherwise} \end{cases} \tag{5.9}$$

Secured fields for a player This is a binary plane that indicates the fields which have been secured by some player, i.e. the player has two stones more on a certain field than any other player. It can be used with the FNet architecture and for the board trunk of the CFNet architecture.

For some player c, it is the 6×6 matrix $\mathbf{I_{secured}}^c = [m_{ij}^c]_{1 \leq i,j \leq 6}$ shown in Equation (5.10), where m_{ij}^c equals 1, if Player c has two stones more than any other player on the field with indices i and j.

$$\mathbf{I_{secured}}^c = [m_{ij}^c] = \begin{cases} 1, & \text{if } \forall p \in \{p \mid 1 \leq p \leq n_\mathrm{p} \text{ and } p \neq c\} : \mathbf{B}^c(i,j) \geq \mathbf{B}^p(i,j) + 2 \\ 0, & \text{otherwise} \end{cases} \tag{5.10}$$

Field values This is a constant plane equal to $\mathbf{F}$. It can be used with the FNet architecture and for the board trunk of the CFNet architecture.

Field probability The constant 6×6 plane $\mathbf{I_{fp}}$ described in Equation (5.11) contains for each field the empirical probability of getting a hand such that, that field can be reached. It can be used with the FNet architecture and for the board trunk of the CFNet architecture.

The probabilities are calculated based on repeatedly sampling hands from the deck as explained earlier in Subsection 3.1.1 and visualised in Figure 3.5. As an example, the field with value 1 at $\mathbf{F}(1,1)$, which can be reached with a probability of 26.73% with a random hand, is represented by the value 0.26729 in $\mathbf{I_{fp}}(1,1)$.

$$\mathbf{I_{fp}} = \begin{bmatrix} 0.26729 & 0.46924 & 0.19872 & 0.33259 & 0.48385 & 0.45061 \\ 0.13534 & 0.54831 & 0.48448 & 0.47750 & 0.10548 & 0.40535 \\ 0.39241 & 0.26695 & 0.17633 & 0.41948 & 0.31642 & 0.57851 \\ 0.47742 & 0.14947 & 0.36902 & 0.22352 & 0.48332 & 0.28535 \\ 0.38196 & 0.51268 & 0.48087 & 0.09019 & 0.30623 & 0.28683 \\ 0.24392 & 0.42499 & 0.36178 & 0.33798 & 0.46713 & 0.11814 \end{bmatrix} \tag{5.11}$$

Field reachability The 6×6 plane $\mathbf{I_{fr}}^c$ described by Equation (5.12) is a matrix of binary values, based on the cards state only. For some player c, it indicates the fields on the board, which Player c has the cards for in his hand of size n_{hand}, while ignoring any restrictions associated with the board state.

It can be used with the FNet architecture and for the board trunk of the CFNet architecture.

$$\mathbf{I_{fr}}^c = [m_{ij}^c] = \begin{cases} 1, & \text{if } F(i,j) \in \{\text{sum}(a) \mid a \in \mathcal{P}_h(\text{PC}^c) \text{ and } 1 \le h \le n_{\text{hand}}\} \\ 0, & \text{otherwise} \end{cases} \tag{5.12}$$

Field availability The 6×6 plane $\mathbf{I_{fa}}^c$ described in Equation (5.13) is a matrix of binary values, based on the board state only. For some player c, it indicates the fields on the board, which are neither secured by Player c w.r.t. every other player, nor any other player has secured the field w.r.t. Player c.

It can be used with the FNet architecture and for the board trunk of the CFNet architecture.

$$\mathbf{I_{fa}}^c = [m_{ij}^c] = \begin{cases} 1, & \text{if } \neg(\exists p \in \{p \mid 1 \le p \le n_{\text{p}} \text{ and } p \ne c\} : \ \mathbf{B}^c(i,j) + 1 < \mathbf{B}^p(i,j)) \\ & \ \wedge \neg(\forall p \in \{p \mid 1 \le p \le n_{\text{p}} \text{ and } p \ne c\} : \ \mathbf{B}^p(i,j) + 1 < \mathbf{B}^c(i,j)) \\ 0, & \text{otherwise} \end{cases} \tag{5.13}$$

Player hand The player hand vector $\mathbf{i_{ph}}^p$ contains binary values that indicate Player p's hand $\text{PC}^p \subset \text{DC}$ in $\mathbf{c_d}$ such that $\mathbf{i_{ph}}^p$ indicates the positions in $\mathbf{c_v}$, starting with the left most position for a given card value for each card with equal value.

It can only be used with the cards trunk of the CFNet architecture.

Let $\mathbf{i_{ph}}^p$ be Player p's hand indicator vector, initialised as:

$$\mathbf{i_{ph}}^p = (0)_{1 \le i \le |\mathbf{c_v}|}$$

For each card c_j in PC^p do the following:

Let v_j be the value of card c_j

Find the leftmost card c_k with value v_j in $\mathbf{c_v}$ such that $\mathbf{i_{ph}}^p(k) = 0$

Assign 1 to the position k in $\mathbf{i_{ph}}^p$

Used cards The player hand vector $\mathbf{i_{uc}}$ contains binary values that indicate the cards that have been already used, i.e. $\mathbf{i_{uc}}$ represents $\mathrm{UC} = \mathrm{DC} \setminus (\mathrm{SC} \cup \bigcup_{p \in \{1,\ldots,n_\mathrm{p}\}} \mathrm{PC}^p)$, in $\mathbf{c_d}$ such that $\mathbf{i_{uc}}$ indicates the positions in $\mathbf{c_v}$, starting with the left most position for a given card value for each card with equal value.

It can only be used with the cards trunk of the CFNet architecture.

Let $\mathbf{i_{uc}}$ be the used cards indicator vector, initialised as:

$$\mathbf{i_{uc}} = (0 \mid_{1 \leq j \leq |\mathbf{c_v}|})$$

For each card c_j in UC do the following:

Let v_j be the value of card c_j

Find the leftmost card c_k with value v_j in $\mathbf{c_v}$ such that $\mathbf{i_{uc}}(k) = 0$

Assign 1 to the position k in $\mathbf{i_{uc}}$

5.4 Convolutional Kernel Initialisation

The initial convolutional layer of the convolutional residual trunk of both, the FieldNet and the CardFieldNet architecture, can be supplied with custom filter kernels. The idea is to accelerate the learning by supplying the kernels that support the detection of horizontal, vertical and diagonal patterns within the input. The custom filter kernels take up the share $\mathrm{N_{CF}}$ of the overall number of filters $\mathrm{N_s}$, defined for the convolutional block, and determine the kernel size $d_{\mathrm{CF}} \times d_{\mathrm{CF}}$ for the convolutional layer. An example is depicted in Figure 5.11. In addition to the colour coding defined at the beginning of this chapter, custom convolutional filters are coloured in a slightly darker orange than randomly initialised convolutional filters.

Similar to the randomly initialised filter kernels RF_i, each custom filter kernel CF_i is applied to all feature planes, as depicted in Figure 5.12.

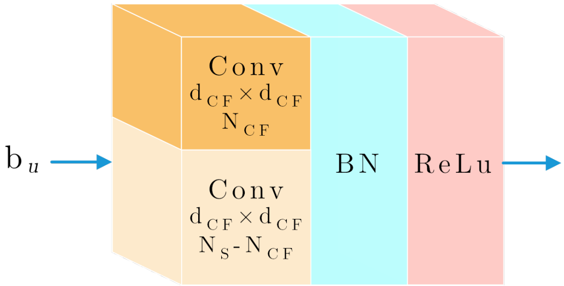


Figure 5.11: The initial convolutional block from the trunk of FieldNet or CardFieldNet with customised filter kernels.

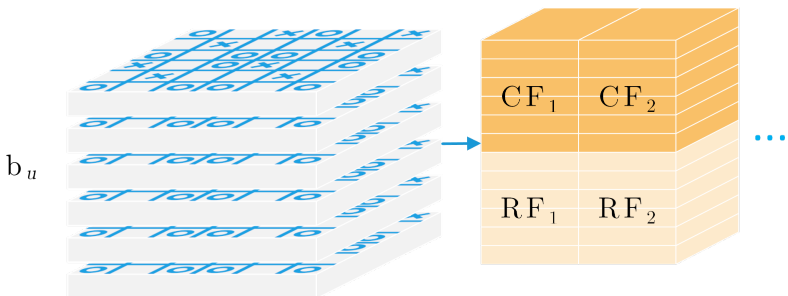


Figure 5.12: Every filter kernel RF_i and CF_i is applied to every feature plane of the input tensor b_u. In this example, there are 2 custom and 2 random filter kernels.

CHAPTER 6

Methods and Implementation

This chapter describes the methods used to construct the proposed agent and evaluate its performance. It describes the experimental setup, i.e. how experiments were performed, result data were generated, collected and processed.

6.1 Prototype Implementation

Julia To evaluate the proposed agent, we implemented a prototype in **Julia** [3] as it offers flexibility and good performance. The description in its documentation [25] reads as follows:

> Julia features optional typing, multiple dispatch, and good performance, achieved using type inference and just-in-time (JIT) compilation (and optional ahead-of-time compilation), implemented using LLVM. It is multi-paradigm, combining features of imperative, functional, and object-oriented programming. Julia provides ease and expressiveness for high-level numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming.

With Laurent's AlphaZero.jl (AZ.jl) package [33], there exists a solid and well-maintained implementation of the AlphaZero framework. AlphaZero.jl already implements mechanisms to run `Benchmarks`, collect and plot performance metrics during training, and to encapsulate hyperparameter configurations in `Experiments`. In the following, the words shown in verbatim text (e.g. `MyStructure`) refer to identifiers in the source code of either AZ.jl or our prototype implementation.

Our implementation is split into two Julia packages. The implementation of the game mechanics, i.e. the `GameInterface` used by AlphaZero.jl, is in the Just4Fun.jl package [18]. The implementation of the neural network architectures (see Chapter 5), the

SO-ISMCTS (see Chapter 4 and Section 6.4), debugging tools, benchmarking tools (Section 6.3), the network input feature planes (see Chapter 5) and the hyperparameter configurations of our experiments (see Chapter 7) are in the AlphaZeroJust4Fun.jl package [17].

The agent has been developed incrementally. We started with a setup close to AZ on Tic-Tac-Toe variants, then replaced the neural network architecture by FieldNet and implemented feature planes. Then we added the cards to the game mechanics, added SO-ISMCTS and used the CFNet architecture. Afterwards, we added the field value mechanics and further feature planes. Finally, by incorporating the multi-stone mechanics, we evaluated the proposed agent on the full two-player version of Just 4 Fun, as it is described in Section 3.1.

6.2 Benchmarking

The agent's (`AlphaJust4FunZeroPlayer`) performance was evaluated within a benchmark. As a baseline, we used random play, a vanilla SO-ISMCTS agent, a cheating MCTS agent, and human players. During hyperparameter search, to monitor the learning success of the DNN, a network-only agent was used. To determine parameters for the SO-ISMCTS within AZJ4F, the vanilla SO-ISMCTS agent was used. All are part of the AlphaZeroJust4Fun.jl package.

Random Play (`RandomPlayer`) This agent always picks its actions uniformly at random from the available actions. With $\mathrm{A}(s)$ being the available actions from a state s, the selected action a is always $a \sim \mathrm{Uniform}(\mathrm{A}(s))$. It is mostly used for monitoring the neural network performance, i.e. convergence speed, overall performance, performance specifically on pattern-win and points-win, and for hyperparameter search.

Network Agent (`NetworkOnly`) This agent selects actions by sampling according to the DNN's policy output.

Vanilla SO-ISMCTS Agent (`IsMctsRollouts`) This agent performs actions, similar to AlphaJust4Fun, as described in Algorithm 4.2. On each turn, it performs n_{iter} MCTS-iterations on n_{det} determinizations of the current game state as described in Algorithm 4.1, but initialises new nodes (see Line 16 in Algorithm 4.1) based on random playouts. The value is set to the terminal reward of the random playout, and the prior probabilities are set to a uniform distribution.

Cheating Monte Carlo Tree Search Agent (`CheatingMctsRollouts`) This agent is in full knowledge of the game's true state. It also performs actions, similar to AlphaZero, as described in Algorithm 3.2. On each turn, it performs n_{iter} MCTS-iterations from the current game state as in Algorithm 3.1, but initialises new nodes (see Line 12 in Algorithm 3.1) based on random playouts. The value is set to the terminal reward of the random playout, and the prior probabilities are set to a uniform distribution.

Human Player Besides the artificial agents above, human players on https://www.yucata.de/, and the author[1] served as a baseline. The AlphaJust4Fun agent acted as the user **AlphaJust4Fun**[2] **(CFNet)** and **AlphaJ4FZeroFNet**[3] **(FNet)** on Yucata. All its games can also be reviewed and downloaded on https://www.yucata.de/en/Ranking/AlphaJust4Fun and https://www.yucata.de/en/Ranking/AlphaJ4FZeroFNet respectively.

The benchmark was conducted during training as well as with the fully trained agent. During training, i.e. after every update of the DNN's parameters, a benchmark of the **Network agent** against the **Random agent** was conducted with 1,000 randomly initialised games. This allows to monitor the DNN's learning progress while keeping the duration of a learning cycle short in comparison to always running the benchmark with the very CPU-intensive MCTS agents. For the benchmark of the fully trained agent, the number of games was smaller, as a game between two MCTS players is a lot more resource-intensive.

The win-rate, i.e. the number of games won, divided by the number of games played between two agents, was used as the main performance metric. Other performance metrics are described in Section 6.3.

Since the skill rating system on Yucata is TrueSkill, we use TrueSkill to compare the agent's performance with human players. The win-rate of AlphaGo Zero has been evaluated on 100 games [46] and, according to the creators of TrueSkill, there are 50–100 games[4] required to reflect a significant skill change. Since J4F is a non-deterministic game, we used a minimum of 100 games to evaluate our agents. The number of games against humans will be much lower, as they are much more time-consuming.

6.3 Other Performance Metrics

We implement sets of test cases that provide insight into certain aspects of a fully trained agent's intelligence in different situations.

The test sets address the following desired abilities for an intelligent agent:

- The ability to recognise a win by pattern with the current turn.
  In one set of tests (`win-pattern`), each player has exactly one stone on each of its fields and fields are occupied by one player only. In this test set, there is only

[1] TrueSkill of 1,029 after 273 played games; https://www.yucata.de/en/User/gwario
[2] https://www.yucata.de/en/User/AlphaJust4Fun
[3] https://www.yucata.de/en/User/AlphaJ4FZeroFNet
[4] https://www.microsoft.com/en-us/research/project/trueskill-ranking-system/

one single action that forms a pattern. In another set of tests (`win-pattern-ms`), players have multiple stones on their fields, i.e. are competing for some fields of the partial pattern.

- The ability to prevent loss by pattern with the next opponent's turn (win by pattern for the opponent). Similar to the win by pattern scenarios, there are also two sets of tests. In one set, agents are competing for the fields (`prevent-loss-pattern-ms`) and in another one they are not (`prevent-loss-pattern`).
- The ability to pick a winning action (pattern win) over an action that is preventing the opponent from (possibly) scoring a win by pattern in its next turn (`prefer-win-pattern`).
- The ability to recognise a win by points (`win-points-ms`) with the current turn.
- The ability to set up double-win conditions (`double-pattern`), i.e. where the agent has two options to score a win by pattern on its next turn.

  E.g. the agent under test has secured two neighbouring fields in the middle of the board and it can secure the fields that extend the pattern to a line of 3 fields. Then the agent is expected to recognise that by extending the pattern to a line of 3, even if the opponent blocks the pattern on one side, it can win by pattern on its next turn.
- The ability to build triple-win conditions, i.e. the agent under test has two options to set up a double-win (pattern win) condition on its next turn (`triangle-pattern`). In these scenarios, the player has three stones next to each other, i.e. forming a triangle, in the middle area of the board.
- The ability to recognise a win by max field with the current turn (`win-max-field-ms`). In these test cases, players are competing for fields and there is only one single action that ends the game with a win.

Each test set is designed, such that the expected agent output metric value, is similar for each individual test case of a test set. For example, in all test cases in which the agent is close to winning, the network's value output is expected to be close to 1 and the policy should emphasise the winning action.

Each test case is described by *a sequence of actions for both players* that lead to some game state s, the agent is tested in. $A_{\mathrm{e}}(s)$ is *a set of expected actions* which an intelligent agent is expected to select for that game state. Let $(s_{\mathrm{e}}, a_{\mathrm{e}}) \in A_{\mathrm{e}}(s)$ be an arc in the game tree from s to s_{e}, by taking action a_{e}.

In every test case, the game state is created according to *the sequence of actions for both players*. Then, for outputs related to the agent's estimation of the game's outcome, it is tested whether the outputs are within some predefined intervals. For outputs related to the agent's estimated policy, we use the averaged cross-entropy for the comparison of the agents. The constructed game state ensures that the agent is in possession of the cards that are necessary to be able to play at least one of the *expected actions*.

The following **examples** illustrate how the tests work:

1. For the value network output, we define the following set of intervals: $\{[-0.3, -1.0], [0.0, 1.0], [0.3, 1.0], [0.5, 1.0], [0.8, 1.0]\}$. E.g. assume the agent's value network output for each test case from the test set, is expected to be close to 1.0, i.e. a high-value state for the current player. Now assume that, for the first test case, the actual value network output is 0.3, i.e. an underestimation of the game state, for the second test case it is 1.0, i.e. expected value estimation. At the end of each test set, for each interval, we count the number of test cases, for which the agent's output was within each interval.
2. For the policy network output, we expect the agent to select one of two sensible actions in each test case. Then the probabilities estimated for these two actions are summed up and also counted per some intervals.
3. For the value-type metrics Q and UCT on the expected actions, calculate the scaled mean and again, count them per some intervals.
4. For the tree search policy and the policy network output, we calculate the cross-entropy between the agent's estimation and an ideal distribution over the available expected actions, i.e. assigning probability 1 to the expected actions and 0 to the other actions. A value of 0 means the agent's policy meets the expectation. Subsequently, we average cross-entropy for each test set. The average cross-entropy of each test set is again averaged over all repetitions of the particular test set. We use those values to compare the agents.

For each of the test sets, we implemented multiple test cases. The `win-pattern`, `prevent-loss-pattern`, and `prefer-win-pattern` test sets, with 128–184 instances, cover almost all the win-patterns. The `double-pattern` and `triangle-pattern` sets, with only 84 and 40 instances, respectively, cover nearly all the double and triangle pattern situations. `win-points-ms` and `win-max-field-ms` have the smallest number of instances, with 19 and 10 test cases, respectively.

In most of the test sets, the scenario is constructed in the early game stages. As a reminder, a J4F game with two players takes at least 7 up to at most 40 turns. The `win-pattern`, `prevent-loss-pattern`, `prefer-win-pattern`, `double-pattern`, and `triangle-pattern` test sets are constructed between the 4th and the 6th turns. The `win-pattern-ms` and `prevent-loss-pattern-ms` test sets are constructed at the intermediate phase of the game. The `win-points-ms` and `win-max-field-ms` test sets are naturally constructed in the late stages of the game.

The number of test cases per test set, along with basic statistics on the number of turns for the test cases per test set, are summarised in Table 6.1.

Test set	Number of test cases	Turns per test case Median	$\mu \pm \sigma$
`win-pattern`	152	**6**	6 ± 0
`prevent-loss-pattern`	128	5	$\mathbf{5.44 \pm 0.50}$
`prefer-win-pattern`	184	**6**	6 ± 0
`double-pattern`	84	**4**	4 ± 0
`triangle-pattern`	40	**4**	4 ± 0
`win-pattern-ms`	136	22	14.65 ± 8.20
`prevent-loss-pattern-ms`	90	17	18.22 ± 4.92
`win-points-ms`	**19**	**39**	39 ± 0
`win-max-field-ms`	**10**	**39**	39 ± 0

Table 6.1: Basic statistics of the test sets, including the number of test cases per test set, and statistics on the number of turns per test set.

We test the following metrics:

Value estimation of the DNN ($V_{\mathbf{net}}$) This is the output v^t of the network's value head on some game state t: $\mathrm{V}^t_{\mathrm{net}} = v^t$

Mean of the scaled action value estimations of the DNN ($\overline{\mathrm{Q}}_{\mathbf{net,scaled}}$) This is the arithmetic mean of the scaled action values over all expected actions. Let $\mathrm{Q}^t_{\mathrm{net}}$ be the action value for some state t, with the reward r^t for state t, the value v^t of state t and the discount factor γ:

$$\mathrm{Q}^t_{\mathrm{net}} = r^t + \gamma \cdot v^t$$

Let with $A(t)$ being the set of all actions a_f and follow-up states s_f from some state t, let $A^t = \{(s_f, a_f) \in A(t)\}$ and $A^t_e = \{(s_f, a_f) \in A_e(t)\}$ the set of expected actions with $A_e(t) \subseteq A(t)$. The set of action values $\mathrm{Q}^{A^t}_{\mathrm{net}}$ for all follow-up states from some state t is:

$$\mathrm{Q}^{A^t}_{\mathrm{net}} = \{\mathrm{Q}^{s_f}_{\mathrm{net}} : (s_f, a_f) \in A^t\}$$

Similarly, for the follow-up states that are reached via the expected actions:

$$\mathrm{Q}^{A^t_e}_{\mathrm{net}} = \{\mathrm{Q}^{s_f}_{\mathrm{net}} : (s_f, a_f) \in A^t_e\}$$

We employ the scaling function *SCL*, defined in Equation (6.1), to normalize the action values for the follow-up states s_f corresponding to a given state t:

$$\mathrm{Q}^{A^t_e}_{\mathrm{net,scaled}} = \{\mathrm{SCL}(\mathrm{Q}^{s_f}_{\mathrm{net}}) : (s_f, a_f) \in A^t_e\}$$

$$\overline{\mathrm{Q}}^{A^t_e}_{\mathrm{net,scaled}} = \frac{1}{|\mathrm{Q}^{A^t_e}_{\mathrm{net,scaled}}|} \cdot \sum_{(s_f, a_f) \in A^t_e} \mathrm{SCL}(\mathrm{Q}^{s_f}_{\mathrm{net}})$$

Combined policy estimation of the DNN ($\mathbf{cP^{pre}_{net}}$) This is the sum of the network's policy estimations p^{a_f} over all expected actions a_f:

$$\mathrm{cP}^{\mathrm{pre},A_\mathrm{e}^t}_{\mathrm{net}} = \sum_{(s_f,a_f)\in A_\mathrm{e}^t} p^t_{a_f}$$

Combined policy estimation of the DNN after masking ($\mathbf{cP_{net}}$) This is the similar to $\mathrm{cP}^{\mathrm{pre},A_\mathrm{e}^t}_{\mathrm{net}}$, but the actions $a_{\mathrm{n/a}} \in \mathrm{NA}(t)$ that are not available from s have been removed, i.e. those components were set to 0 and the vector was subsequently renormalized:

$$p^t_a = \begin{cases} 0 & \text{if } a \in \mathrm{NA}(t) \\ \frac{p^t_a}{\sum_{b \notin \mathrm{NA}(t)} p^t_b} & \text{if } a \notin \mathrm{NA}(t) \end{cases}$$

Mean of the scaled action values ($\overline{\mathbf{Q}}_{\mathbf{mcts,scaled}}$) This is the arithmetic mean of the scaled action values over all expected actions: $\overline{\mathrm{Q}}^{A_\mathrm{e}^t}_{\mathrm{mcts,scaled}}$

The scaling is done similar to $\overline{\mathrm{Q}}_{\mathrm{net,scaled}}$, but over $\mathrm{Q}^t_{\mathrm{mcts}} = \sum_{(s_f,a_f)\in A_\mathrm{e}^t} \frac{\mathrm{W}^{a_f}}{n^{a_f}}$, with the total action value W^{a_f} and the number of visits n^{a_f} in t.

Combined MCTS policy ($\mathbf{cP_{mcts}}$) This is the sum of the MCTS based policy, over all expected actions a_f, with total visit count n^t in state t:

$$\mathrm{cP}^{A_\mathrm{e}^t}_{\mathrm{mcts}} = \sum_{(s_f,a_f)\in A_\mathrm{e}^t} \frac{n^{a_f}}{\max(1, n^t)}$$

Mean of the scaled UCT values ($\overline{\mathbf{UCT}}_{\mathbf{scaled}}$) This is the arithmetic mean of the scaled UCT scores, without Dirichlet noise, over all expected actions: $\overline{\mathrm{UCT}}^{A_\mathrm{e}^t}_{\mathrm{scaled}}$

The scaling is done similar to $\overline{\mathrm{Q}}_{\mathrm{net,scaled}}$, but over

$$\mathrm{UCT}^t = \sum_{(s_f,a_f)\in A_\mathrm{e}^t} \left[\frac{\mathrm{W}^{a_f}}{\max(1, n^t)} + C(t) \cdot \mathbf{p}^t_{a_f} \cdot \frac{\sqrt{n^t}}{n^{a_f}_{av} + 1} \right],$$

with $n^{a_f}_{av}$ being the number of times a_f was available during tree search and $C(t)$ being the exploration factor for t.

The above equations were used for SO-ISMCTS-based agents. For the MCTS-based agents, we used the vanilla AZ tree policy instead.

Cross-entropy This is the cross-entropy between the agent's estimated policy $P^{A^t}_{\mathrm{mcts}}$ and the ideal distribution $P^{A^t}_{\mathrm{I}}$ over the available expected actions $(s_f, a_f) \in A^t_\mathrm{e}$. The ideal policy is

$$P^{A^t}_{\mathrm{I}} = \left(p^{a_f}_{\mathrm{I}}\right)_{(s_f,a_f)\in A^t} = \begin{cases} 0 & \text{if } a_f \notin A^t_\mathrm{e} \\ \frac{1}{|A^t_\mathrm{e}|} & \text{if } a_f \in A^t_\mathrm{e} \end{cases}$$

and the estimated policy is $P^{A^t}_{\mathrm{mcts}} = \left(p^{a_f}_{\mathrm{mcts}}\right)_{(s_f,a_f)\in A^t}$ (and similarly for the network policies $P^{A^t}_{net}$ and $P^{\mathrm{pre},A^t}_{net}$). The cross-entropy based on the MCTS policy $CE^{A^t}_{\mathrm{mcts}}$ is then calculated with

$$CE^{A^t}_{\mathrm{mcts}} = H(P^{A^t}_{\mathrm{I}}, P^{A^t}_{\mathrm{mcts}}) = -\sum_{(s_f,a_f)\in A^t} p^{a_f}_{\mathrm{I}} \cdot \log(p^{a_f}_{\mathrm{mcts}})$$

and similarly for the network policies $CE^{A^t}_{net}$ and $CE^{\mathrm{pre},A^t}_{net}$. The cross-entropy based on a uniformly random policy is denoted by $CE^{A^t}_{rand}$. In our implementation, we group actions into expected and non-expected actions, sum up their probabilities and calculate the cross-entropy based on those. The justification is, as discusses in Section 6.4, that we want to avoid penalisation of agent estimations that only emphasises one among multiple equally valid expected actions.

Since the Q-values can get larger than 1, the Q-value metrics are scaled using the tanh-based function $\mathrm{SCL}(x)$ described by Equation (6.1) and visualised in Figure 6.1. It is close to linear for $x \in (-1, 1)$ and saturates towards -4 and 4.

$$\mathrm{SCL}(x) = 4 \cdot \tanh\left(x \cdot \tanh^{-1}\left(\frac{1}{4}\right)\right) \tag{6.1}$$

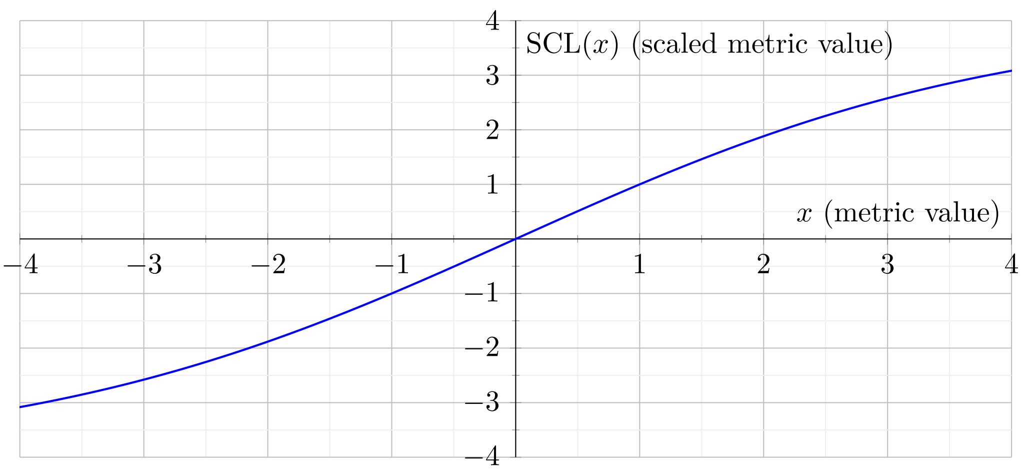


Figure 6.1: The function used to scale the metric values to the interval of $[-4, 4]$. For values from the interval $(-1, 1)$ it is close to linear.

We define the categories "Successful", "Acceptable" and "Unsuccessful". For each test set, we assign each of the metric intervals one of these categories. We repeat the tests several times and record the number of times a metric falls into each category. Then we calculate the average percentage, which reflects how close an agent is to making the expected judgement. The chosen interval to category assignments for the value metrics are listed in Table 6.2 and the ones for the policy metrics in Table 6.3.

Value metrics

win-pattern, win-pattern-ms & prefer-win-pattern Since this situation is trivial to judge, we interpret only the uppermost value range of $(0.8, 4]$ as a success. But we deem probabilities of greater than 0.6 for the single expected action as acceptable.

win-points-ms & win-max-field-ms While the value range interpreted as a success is similar to the pattern-win/loss, we even consider a value of greater than 0.45 as acceptable.

prevent-loss-pattern & prevent-loss-pattern-ms In these situations, the player is on the brink of being defeated by a pattern but can still prevent it for at least a few turns. For that reason, we consider a value range of $[-0.3, 0]$ and $[-0.45, 0.1]$ (multi-stone) as a success. In the non-multi-stone scenario, we deem the surrounding intervals of $[-0.8, -0.3)$ and $(0, 0.3]$ as still acceptable. In the multi-stone scenario, we accept values from the intervals $[-0.8, -0.45)$ and $(0.1, 0.3]$.

double-pattern & triangle-pattern Building these situations doesn't immediately lead to a reward, but it is a good mid- and long-term strategy. For that reason, we interpret a larger interval in the positive value range as a success. That is, for `double-pattern`, $(0, 0.6]$ and we accept values from $[-0.1, 0]$ and $(0.6, 1]$. For `triangle-pattern`, $(0.1, 0.6]$ is a success and acceptable are values from $[-0.1, 0.1]$ and $(0.6, 1.1]$.

Policy metrics

win-pattern, win-points-ms, prefer-win-pattern, prevent-loss-pattern & prevent-loss-pattern-ms Since these situations are trivial to judge and require early intervention, we interpret only the value range of $(0.8, 1]$ as a success.

win-points-ms & win-max-field-ms For these situations, the success-range is similar to the one for `win-pattern`, but the acceptable-range is slightly wider.

double-pattern & triangle-pattern Here we interpret an even larger value range of $(.45, 1]$ as a success.

Cross-entropy The cross-entropy is not rated on its own but compared among the agents and test sets. Generally, for agent policies closer to the expected policies, the cross-entropy is lower (ideally 0).

Test set	Interval ratings $\overline{\text{UCT}}_{\text{scaled}}$, $\overline{\text{Q}}_{\text{mcts,scaled}}$, $\overline{\text{Q}}_{\text{net,scaled}}$, V_{net}				
`win-pattern`	[-4.0, 0.6]			(0.6, 0.8]	(0.8, 4.0]
`prevent-loss-pattern`	[-4.0, -0.8)	[-0.8, -0.3)	[-0.3, 0.0]	(0.0, 0.3]	(0.3, 4.0]
`prefer-win-pattern`	[-4.0, 0.6]			(0.6, 0.8]	(0.8, 4.0]
`double-pattern`	[-4.0, -0.1)	[-0.1, 0.0]	(0.0, 0.6]	(0.6, 1.0]	(1.0, 4.0]
`triangle-pattern`	[-4.0, -0.1)	[-0.1, 0.1]	(0.1, 0.6]	(0.6, 1.1]	(1.1, 4.0]
`win-pattern-ms`	[-4.0, 0.6]			(0.6, 0.8]	(0.8, 4.0]
`prevent-loss-pattern-ms`	[-4.0, -0.8)	[-0.8, -0.45)	[-0.45, 0.1]	(0.1, 0.3]	(0.3, 4.0]
`win-points-ms`	[-4.0, 0.45]			(0.45, 0.8]	(0.8, 4.0]
`win-max-field-ms`	[-4.0, 0.45]			(0.45, 0.8]	(0.8, 4.0]

Table 6.2: Mapping of value ranges to rating categories for state-value-related metrics.

Test set	Interval rating cP_{mcts}, cP_{net}, $\text{cP}^{\text{pre}}_{\text{net}}$		
`win-pattern`	[0.0, 0.6]	(0.6, 0.8]	(0.8, 1.0]
`prevent-loss-pattern`	[0.0, 0.6]	(0.6, 0.8]	(0.8, 1.0]
`prefer-win-pattern`	[0.0, 0.6]	(0.6, 0.8]	(0.8, 1.0]
`double-pattern`	[0.0, 0.3]	(0.3, 0.45]	(0.45, 1.0]
`triangle-pattern`	[0.0, 0.2]	(0.2, 0.45]	(0.45, 1.0]
`win-pattern-ms`	[0.0, 0.6]	(0.6, 0.8]	(0.8, 1.0]
`prevent-loss-pattern-ms`	[0.0, 0.6]	(0.6, 0.8]	(0.8, 1.0]
`win-points-ms`	[0.0, 0.45]	(0.45, 0.8]	(0.8, 1.0]
`win-max-field-ms`	[0.0, 0.45]	(0.45, 0.8]	(0.8, 1.0]

Table 6.3: Mapping of value ranges to rating categories for policy-related metrics.

6.4 Implementation Details

Exploration Factor $C(s)$ The exploration factor, described by Equation (4.3), is implemented as a constant in AlphaZeroJust4Fun.jl, similar to AlphaZero.jl.

Position Averaging and Sample Weights AlphaZero.jl provides an option for position averaging (see Chapter 4, Paragraph 5), i.e. samples in memory that correspond to the same tree node are averaged. The resulting merged sample is weighted according to the sample-weight policy $\omega(n_i)$. We always use logarithmic sample weighting: $\omega(n_i) = \log_2(n_i) + 1$, with n_i being the number of samples that correspond to the same tree node.

Network Policy The policy head output is masked, i.e. the probabilities for all unavailable actions are set to 0, and renormalised before e.g. sampling an action. To prevent the divisor from being 0, when all remaining network outputs are 0, the smallest possible value of the 32-bit float type, that is being used, is added to the divisor.

Information Set Nodes The search tree in AlphaZero.jl's MCTS is implemented as a map structure. In AlphaZeroJust4Fun.jl we implement the information set tree nodes by using only the information set key-state, a subset of the full game-state, when performing node look-ups. This effectively combines all full game-states of an information set into a single node when updating node or arc statistics. The information set key-state is configurable, but not considered a hyperparameter as it is a fundamental element of the algorithm. The `CheatingMctsRollouts` agent is implemented as a vanilla MCTS agent that operates with the full state as information set key-state.

Redraw The mandatory redraw, triggered when a player has no valid card combinations, is implemented as a game mechanic. That is, no network evaluations nor MCTS-iterations are performed, only the player's hand is replaced.

Card Selection In case there are several subsets of a player's hand, that target the selected field, one is selected uniformly at random.

Training on Perfect Information When training, during SP, a perfect information version of the game, i.e. a perfect simulator, can be used. In this case, the bias term of AZ, which is described by Equation (3.2), is being used. When playing with the imperfect information version of the game, the AlphaJust4Fun bias term, which is described by Equation (4.3), is used.

Hyperparameter Schedules AlphaZero.jl only implements a parameter schedule for the replay buffer size. Any other hyperparameter scheduling is performed manually, by interrupting the training process, changing the parameters and then resuming the training process.

Self-play and Network Updates In the version of AlphaZero.jl we used, SP and network updates are not performed in parallel. Instead, each iteration or epoch consists of the three steps, self-play (SP), mini-batch updates, and training benchmark. We only used the learning-rate parameter η of the Adam optimiser, for the other parameters we always used the defaults $\beta 1 = 0.9$, $\beta 2 = 0.999$ and $\epsilon = 10^{-8}$.

Play Against Humans on Yucata When playing against humans, we use a local game with an interactive agent. On each turn, we manually apply the game state from the Yucata game to our corresponding local game, i.e. set the visible state and randomise the hidden state. Specifically, the opponent's hand is set to contain the cards of its most recent action. After setting the game state in the local game instance, the opponent's action in the Yucata (Yucata) game is applied manually as the opponent in the local game. Then the local agent's response is computed and applied locally and also in the game on Yucata.

Random Seed The random seed is set at the beginning of scripts that are performing training, benchmarking, or testing, to some value s_{init}. Additionally, before every training iteration i, the random seed is set to $s_{\text{init}} + i$. This makes the self-play and network updates reproducible, independent of the training benchmarks that are performed at the end of each training iteration, and allows for the resumption of the training from any iteration on. One drawback of our benchmark implementation is that, when multiple benchmarks are scheduled in one run, the resumption of the run starts with the initial random seed. Furthermore, each individual benchmark of a run is not based on the same randomly initialised games. To counter the impact, we set the number of games in each benchmark to at least 100.

Test Sets The multi-stone test sets (indicated by the suffix "ms" in the test set name) are in a multi-stone scenario, which means that the fields of e.g. the pattern have multiple stones of each player, but there is still only one winning action in the *win-pattern-ms* test set.

In our implementation of the test cases in the `prevent-loss-pattern` and `prevent-loss-pattern-ms` test sets, we only ensure that the acting player can prevent a potential loss, but it is not ensured that the opponent has the cards to actually win.

For the test cases of the `prevent-loss-pattern`, `prevent-loss-pattern-ms`, `double-pattern`, `triangle-pattern`, and `win-points-ms` test sets, there are multiple expected actions. Since in some game states, the agent's hand might only allow a subset of the expected actions to be played, we use the sum of probabilities of available expected action and non-expected actions for the value- and policy-metrics. For the calculation of the *cross-entropy*, we used a probability of 1 for the expected actions and 0 for the other actions as actual values. For the agent's prediction, we group the probabilities, into expected and non-expected, and sum them up. Then we calculate the cross-entropy for these grouped probabilities. Due to the implementation of the cross-entropy formula, numerical errors can occur. To avoid undefined values for 0 probabilities, an ϵ is added to the arguments of the logarithm, where $\epsilon =$ `eps(Float64)`.

Since each test set is run multiple times, we aggregate the outcomes. For the value- and policy-metrics, we determine the number of test cases with metric values that fall within the specified range for each rating category. Following this, we compute the overall percentage for every rating category. For the cross-entropy, we compare the arithmetic mean of the arithmetic mean values (mean, median, and standard deviation) of the test cases within a test set over all runs of that test set.

Number of MCTS-iterations per Determinization Our implementation provides the number of MCTS-iterations (n_{iter}) per turn and the number of determinizations (n_{det}) per turn as hyperparameters. These values can be set independently of each other. The number of MCTS-iterations per determinization $n_{\text{sim,det}}$ is calculated by dividing the number of MCTS-iterations by the number of determinizations. We use the round-to-nearest method with round-to-even as a tie-break rule for the division, i.e. if the fraction is exactly 0.5, the value is rounded to the nearest even integer. $n_{\text{sim,det}}$ is then used for every determinization. It should be noted that due to the rounding, this can lead to a different number of overall MCTS-iterations than defined by n_{iter} (at most $\frac{n_{\text{det}}}{2}$).

CardFieldNet Architecture Details As we will mention later in Subsection 7.6.2, we needed to fine-tune some aspects of the architecture to make it work. After continuous issues with exploding and vanishing gradients, we made the following adjustments:

- The convolutional layers in the board trunk were all initialised with values from the *Glorot-uniform* distribution [20].
- After each convolutional layer in the board trunk we added a *dropout layer* [24].
- As rectifier nonlinearities after convolutional layers in the board trunk, we used the *Randomised Leaky Rectified Linear Unit* [52].
- In the cards trunk, after the last dense layer, but before batch normalisation, we added a dropout layer.
- All dense layers were initialised with values from the *Glorot-normal* distribution [20].
- As rectifier nonlinearities after all dense layers, we used the *Scaled Exponential Linear Unit* [29].
- In the common trunk, the value head and the policy head, after the last dense layer and after batch normalisation and activation, we added a *dropout layer*.

CHAPTER 7

Experimental Findings

In this chapter, we present the results on benchmarks and test sets and other experimental findings.

7.1 Baseline Agent Performance

We chose several ways of determining an agent's strength. The first one is to compare the performance against the `RandomPlayer`. The second way is to compare the performance against a MCTS agent that does its random playouts with full knowledge of the hidden information (`CheatingMctsRollouts`). The `CheatingMctsRollouts` baseline allows us to benchmark not only AlphaJust4Fun but also the isolated performance of an `IsMctsRollouts` agent. This in turn allows tuning of the hyperparameter configuration of the SO-ISMCTS part of AlphaJust4Fun, i.e. the information set key-state and the number of determinizations, for follow-up tests and benchmarks.

7.1.1 `RandomPlayer` baseline

We first analyse the random baseline in isolation by letting two `RandomPlayer` agents compete against each other. In addition to the findings briefly mentioned in Subsection 3.1.1, we find that there was no advantage for the starting player in this scenario. However, as we will describe later, there is an advantage for a non-random starting player. We also observe, that it is not unlikely, even with the restriction imposed by the cards, for randomly played games to end with patterns and thus, games ending with patterns are not necessarily an indicator of intelligence or successful strategies. *Conversely, we hypothesise that preventing the opponent from achieving a pattern is an important element of successful strategies.*

The results are summarised in Table 7.1. The randomness seeds we used for this benchmark are listed in Table C.3c.

		When Started (%)		When Not Started (%)	
Won	Pattern	5,938	(23.76)	5,387	(21.54)
	Points	6,532	(26.14)	6,801	(27.19)
	Max-Field	54	(0.22)	48	(0.19)
	Total	12,524	(50.12 won)	12,236	(48.92)
Lost	Pattern	5,427	(21.72)	6,032	(24.12)
	Points	6,979	(27.93)	6,686	(26.73)
	Max-Field	59	(0.24)	57	(0.23)
	Total	12,465	(49.88 lost)	12,775	(51.08)
Total		24,989	(49.98 started)	25,011	(50.02)

Table 7.1: The results of Player 1 in the `RandomPlayer` vs. `RandomPlayer` benchmark are based on 50,000 games. It shows that winning through pattern is, despite the constraint imposed by the cards, fairly easy if the opponent does not actively try to hinder it.

We also ran the `RandomPlayer` agent, which is always using a uniform distribution as a policy over the available actions, on our test sets. When running our test sets 120 times from randomly initialised initial game states, we observe an average mean cross-entropy values of around 2.5 with average standard deviations of around 0.93 on most test sets. For `win-points-ms` it performs noticeably better, with 0.93 ± 0.75. Due to the higher share of expected ("correct") actions in some states, the probability of guessing correctly can become fairly high. Thus, correct behaviour in those scenarios becomes less indicative of intelligence. This can be observed especially in the `win-points-ms` test set. It is often the case that several game-ending actions are available that let a player have more overall points than the opponent. The average median cross-entropy values are around 2.3. For `prevent-loss-pattern`, the average median is slightly lower with 1.91, and for `win-points-ms`, it is significantly lower with 0.75. The average mean values and the average standard deviations (formatted as $\overline{\mu} \pm \overline{\sigma}$) are summarised in Table 7.2, the average median values are visualised in Figure 7.9 and the randomness seeds are listed in Table C.1.

Test set	**`RandomPlayer`** $\mathbf{CE_{rand}}$
`win-pattern`	2.81 ± 0.98
`prevent-loss-pattern`	2.09 ± 0.84
`prefer-win-pattern`	2.76 ± 0.98
`double-pattern`	2.74 ± 0.96
`triangle-pattern`	2.50 ± 0.91
`win-pattern-ms`	2.73 ± 0.98
`prevent-loss-pattern-ms`	2.49 ± 0.94
`win-points-ms`	$\mathbf{0.93 \pm 0.75}$
`win-max-field-ms`	2.48 ± 0.89

Table 7.2: Cross-entropy metrics for the `RandomPlayer`, averaged ($\overline{\mu} \pm \overline{\sigma}$) over 120 repetitions per test set.

7.1.2 **`CheatingMctsRollouts`** baseline

The cheating MCTS agent is our main baseline for benchmarking our AlphaJust4Fun agents. The main parameter for the `CheatingMctsRollouts` agent is the number of playouts per turn. To estimate its skill-level, we recorded the number of playouts at which its win-rate matched one of a `RandomPlayer`. We did the same also against a human player. The author, an experienced[1,2] player, was used as a human baseline. Note that the search tree has been reset after each game, which might weaken the cheating player in comparison to one that maintains its search tree over many games. For future work, it would be interesting to compare the performance over a larger number of competitive games without resetting the search tree after every single game. However, the "fresh" `CheatingMctsRollouts` agent is a fair comparison during training.

Benchmark performance Our benchmark shows that a `CheatingMctsRollouts` agent with only 2 playouts can beat a `RandomPlayer` agent convincingly. With **50 MCTS-iterations**, a `CheatingMctsRollouts` agent wins approximately **80% of the games against a `RandomPlayer` agent**.

To achieve a **win-rate of 40% in 50 games against our baseline human player**, a `CheatingMctsRollouts` agent needs to perform **8,000 playouts per turn**.

The full configurations used for the benchmark are displayed in Tables C.3a and C.3b.

Test performance The `CheatingMctsRollouts` agent, which we ran against the test sets, performed 900 MCTS-iterations. The full configuration used for the tests is displayed in Table C.2a.

It is typically capable of recognising actions that result in a win, which are represented by the `win-pattern`, `prefer-win-pattern`, `win-pattern-ms`, and `win-points-ms` test sets. This is indicated by average median values of close to zero as well as the low average mean values compared to the `RandomPlayer`. For the `win-max-field-ms` test set, which consists of rather trivial game situations, the typical cross-entropy is also close to zero. However, the high average mean value of 3.26 and the very high average standard deviation of 7.76 indicate that there are still instances where the agent's estimation significantly diverged from the optimal policy. *The bad performance in situations, in which an action would prevent a potential loss on the following turn by the opponent (the `prevent-loss-pattern` and the `prevent-loss-pattern-ms` test sets), is most likely because the opponent has a low probability of having a winning hand (as we described in Section 6.4). A cheating player is aware of the opponent's hand during playouts, and might thus choose a different, potentially more effective, policy than the one we would expect from a player unaware of the opponent's hand in these situations. The low performance on the double- and triangle-pattern test sets (`double-pattern`*

[1] Player profile: https://www.yucata.de/en/User/gwario

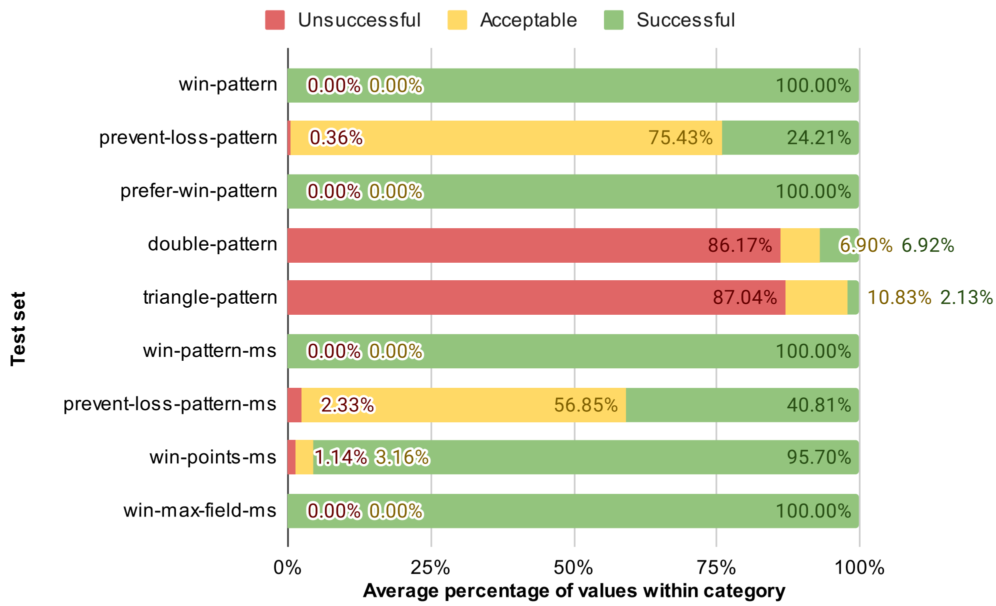
[2] μ and σ are displayed in the ranking table for Just 4 Fun: https://www.yucata.de/en/Ranking/Game/Just4Fun; TrueSkill=1,029, $\mu = 1133.122$ and $\sigma = 34.66538$ after 273 games at the time of writing

and `triangle-pattern`) is not surprising, as it requires long-term planning, for which 900 playouts are likely not enough. Additionally, the knowledge of the full game state also allows for more exploitation and earlier wins, which is especially the case for the cheating MCTS agent. The average standard deviations of the `CheatingMctsRollouts` agent's cross-entropy are relatively high on all test sets. Table 7.3 summarieses the average mean values and the average standard deviations of the cross-entropy, with the `CheatingMctsRollouts` agent's values highlighted in gray. The average medians are visualized in Figure 7.9.

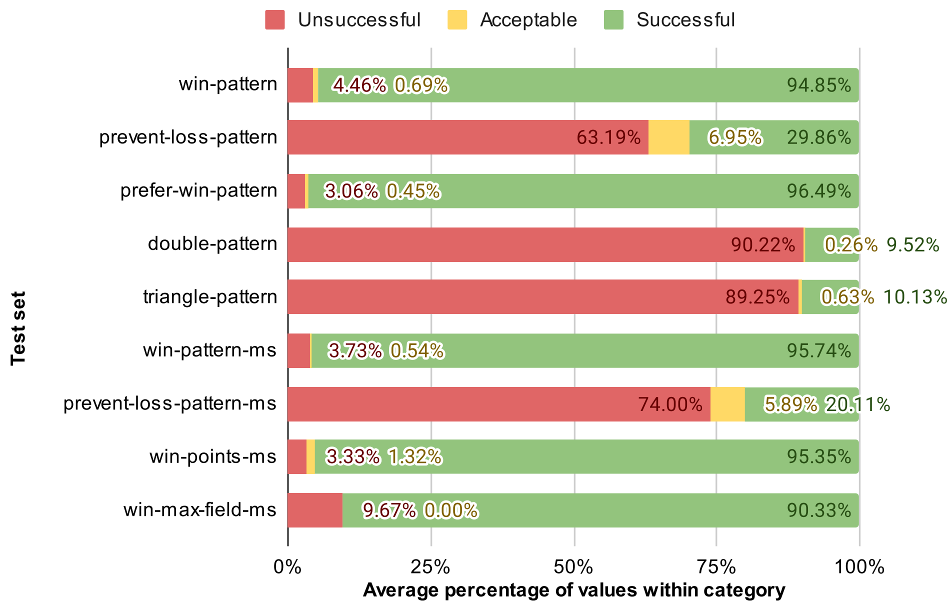
Test set	**`CheatingMctsRollouts`** $\mathbf{CE_{mcts}}$	**`RandomPlayer`** $\mathbf{CE_{rand}}$
`win-pattern`	$\mathbf{0.65 \pm 4.16}$	2.81 ± 0.98
`prevent-loss-pattern`	3.21 ± 2.84	$\mathbf{2.09 \pm 0.84}$
`prefer-win-pattern`	$\mathbf{0.45 \pm 3.40}$	2.76 ± 0.98
`double-pattern`	5.92 ± 2.19	$\mathbf{2.74 \pm 0.96}$
`triangle-pattern`	5.44 ± 2.21	$\mathbf{2.50 \pm 0.91}$
`win-pattern-ms`	$\mathbf{0.69 \pm 4.54}$	2.73 ± 0.98
`prevent-loss-pattern-ms`	4.06 ± 2.98	$\mathbf{2.49 \pm 0.94}$
`win-points-ms`	$\mathbf{0.70 \pm 2.91}$	0.93 ± 0.75
`win-max-field-ms`	3.26 ± 7.76	$\mathbf{2.48 \pm 0.89}$

Table 7.3: Comparison of the cross-entropy for the `CheatingMctsRollouts` player (MCTS policy; 900 MCTS-iterations in each turn; 60 repetitions) and the `RandomPlayer` (120 repetitions), averaged ($\overline{\mu} \pm \overline{\sigma}$) over multiple repetitions of each test set.

The suboptimal performance in situations represented by the `prevent-loss-pattern` and `prevent-loss-pattern-ms` test sets is also reflected in our rating of the policy-based metric cP_{mcts} on the expected actions. The $\overline{UCT}_{scaled}$ values are mostly in the acceptable range, as Figure 7.1a shows. The weakness in longer-term strategies (`double-pattern` and `triangle-pattern`) is also expressed clearly by the bad ratings of both metrics. Figure 7.1 shows the rating for the value- and policy-metrics.



(a) $\overline{\mathrm{UCT}}_{\mathrm{scaled}}$



(b) $\mathrm{cP}_{\mathrm{mcts}}$

Figure 7.1: Rating of UCT-quality (a) and policy-quality (b) of `CheatingMcts-Rollouts` on the test sets, averaged over 60 repetitions of each test set. The agent performed 900 playouts.

7.2 Information Set Key-State

The high branching factor introduced by the cards has a big impact on the computational costs of exploration depth. The information set key-state introduced in Section 6.4 allows us to change the amount of game states that are combined in the information set nodes, which impacts the branching factor. This in turn impacts the exploration depth for a given number of SO-ISMCTS iterations per turn. We evaluated different information set key-states in a benchmark, using the vanilla SO-ISMCTS agent (`IsMctsRollouts`) against the cheating MCTS agent (`CheatingMctsRollouts`).

For the information set key-states, we consider combinations of the player-cards-state (HC - the cards in the player's hand), the public cards-state (UC - used cards) and the board-state (BS).

We compare the performance of the `IsMctsRollouts` with the following information set key-states:

BS, HC, and UC This effectively combines the statistics of all game states, collected during the MCTS-iterations, that share the same board-state, player-cards-state, and public cards-state. This is the largest information set key-states. It combines the least number of (full) game states.

BS and HC This combines the statistics of all game states that share the same player-cards-state and board-state. This information set key-states is much smaller than BS, HC, and UC as it combines all permutations of the pile of used cards, the stack of cards and the cards in the opponent's hand.

BS This combines the statistics of all games states that share the same board-state. It is the smallest information set key-states and combines all permutations of the full cards-state.

In a first benchmark on 100 randomly initialised games, we used a moderate number of playouts per turn (900), and a rather large number of determinizations (120). This results, after rounding, in 8 playouts per determinization and 960 overall playouts for the `IsMctsRollouts` agent. The `CheatingMctsRollouts` baseline still performs exactly 900 playouts.

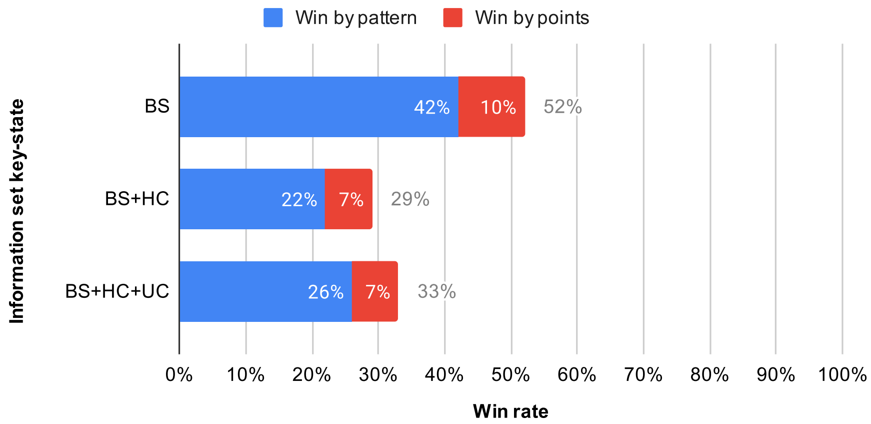
The result indicates that the `IsMctsRollouts` agent with the smallest information set key-state performs best against the `CheatingMctsRollouts` baseline. With a win rate of 52%, its performance is similar to the baseline's performance. Most wins were due to win by pattern. The fact that `IsMctsRollouts` performed even slightly better than `CheatingMctsRollouts` also might indicate that the number of played games is not sufficient for comparison, or that the `CheatingMctsRollouts` baseline does not perform optimally with only 900 playouts per turn, or a suboptimal exploration/exploitation balance. The second-best performance was observed for the largest information set key-state, BS, HC, and UC. Figure 7.2a shows the benchmark performance for different information set key-states.

For a second benchmark, we increased the number of playouts per turn (8,000) and the number of determinizations per turn (700) significantly. This results in 11.4 playouts per determinization and should lead to a deeper search per determinization for the `IsMctsRollouts` agent.

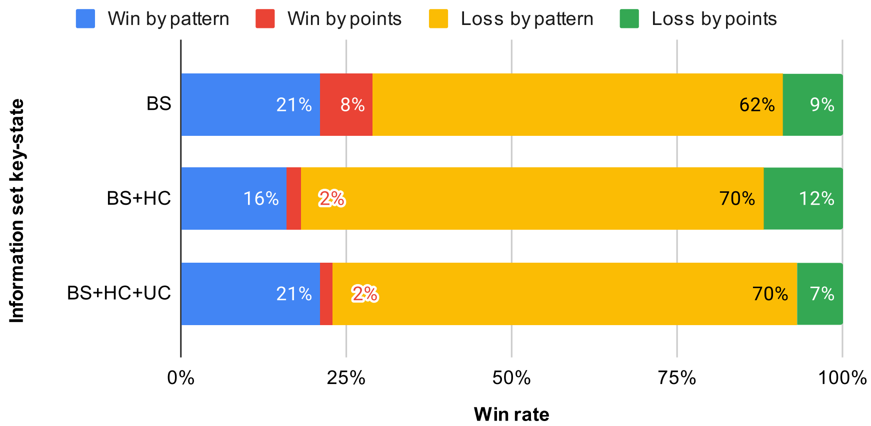
The results of this benchmark show overall a much lower win rate for the `IsMctsRollouts`. The `IsMctsRollouts` agent with BS, HC, and UC and the one with *BS* performed best, but with only 29% and 23% win rate, respectively. The number of win by pattern outcomes was similar for both, the agent with BS and the agent with BS, HC, and UC. The agent with BS scored more win by points. Comparing the performance of the agent with BS, HC, and UC for the higher and the lower number of playouts, it appears that its performance decreased less, relative to the BS agent. Figure 7.2b shows the results of this benchmark.

The results suggest that the smaller information set key-state works better for `IsMctsRollouts` agents with a lower number of MCTS-iterations, at least without the support of a DNN. *The reason might be that the search depth with* 900 *MCTS-iterations was not sufficient for the largest information set key-state. The results also suggest that agent performance scales better with w.r.t. the number of MCTS-iterations for BS, HC, and UC.* For a more meaningful comparison of MCTS based agents, more playouts per turn might be necessary. This would also be the case in competitive play against humans.

The complete set of hyperparameters for this experiment is listed in Table C.6.



(a) Win rate of `IsMctsRollouts` (120 determinizations) against `Cheating-MctsRollouts` with 900 of playouts based on 100 games.



(b) Win rate of `IsMctsRollouts` (700 determinizations) against `Cheating-MctsRollouts` with 8,000 of playouts based on 100 games.

Figure 7.2: Win rate of SO-ISMCTS against the baseline with different information set key-states.

7.3 Determinization/MCTS-Iteration Balance

To investigate the relationship between the number of playouts per turn n_{iter} and the number of determinizations per turn n_{det}, we conduct experiments similar to those presented by Cowling et al. [12]. We benchmark the `IsMctsRollouts` agent against the `CheatingMctsRollouts` baseline on 100 games for each pair of $n_{\text{det}} = 900$ and n_{iter}. For the first benchmark, we used the minimal information set key-state BS (Figure 7.3) and for a second benchmark, the biggest information set key-state BS, HC, and UC (Figure 7.4).

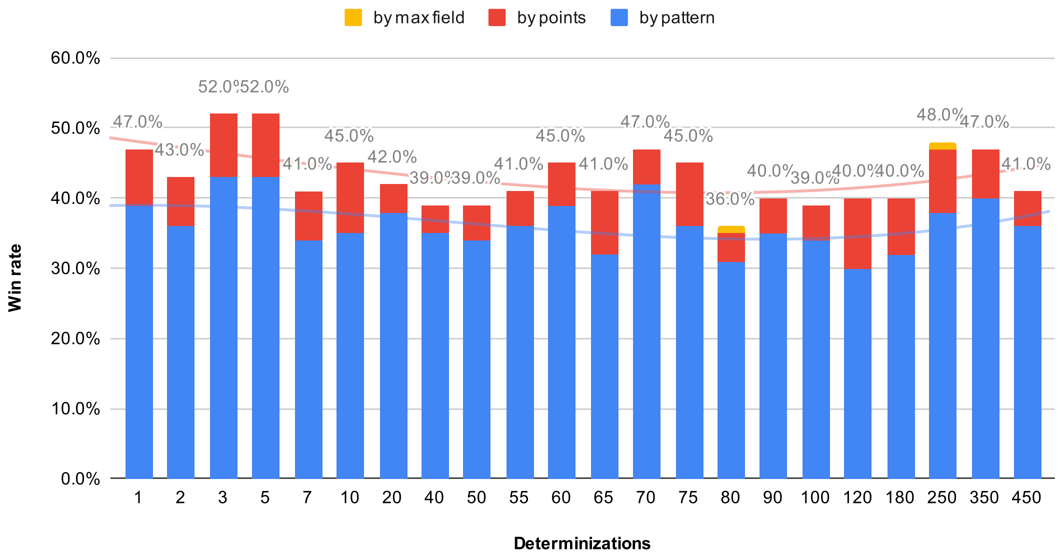


Figure 7.3: Win rate of the `IsMctsRollouts` agent (900 playouts, BS) against the `CheatingMctsRollouts` agent (900 playouts) in a benchmark of 100 games. The trend lines are polynomial with a degree of 3.

The win rate of the `IsMctsRollouts` agent with BS only against the baseline (Figure 7.3) mostly varies between 40% and 50%, with the best performance of 52% occurring between 3 and 5 determinizations per turn.

For the `IsMctsRollouts` agent with BS, HC, and UC (Figure 7.4), the win rate decreases noticeably with an increasing number of determinizations. This is most likely due to the decreased exploration depth as the number of playouts on each determinization decreases. For example, the agent with only one determinization performs 900 playouts on a single determinization in each turn. However, the determinization is still different in each turn, as the known game state changes with each turn. The agent with 450 determinizations performs only two playouts per determinization in each turn. This results in an exploration depth of at most 2 turns. It should be noted that the `CheatingMctsRollouts` baseline with 900 playouts per turn does lose in 33% and 41%

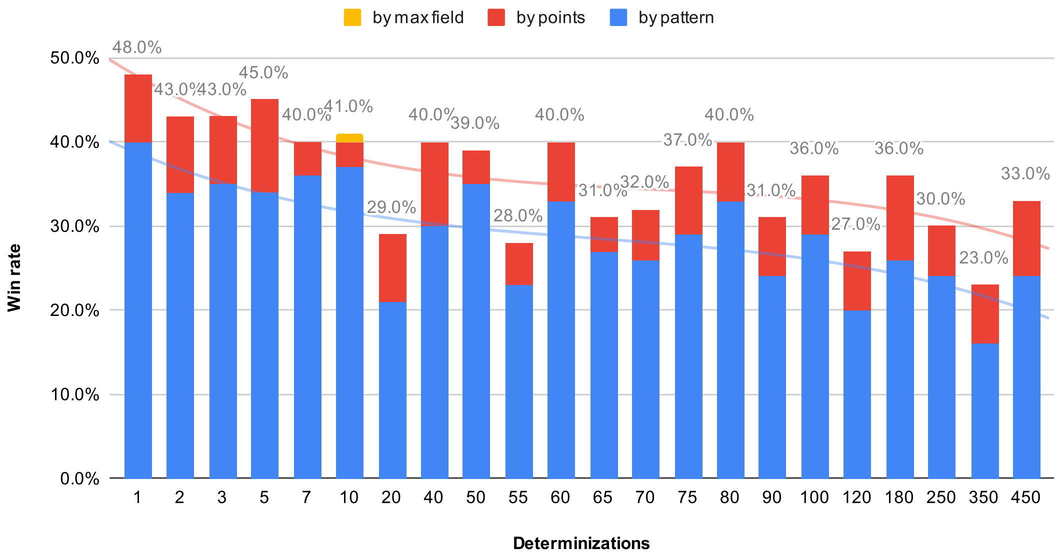


Figure 7.4: Win rate of the `IsMctsRollouts` agent (900 playouts, BS, HC, and UC) against the `CheatingMctsRollouts` agent (900 playouts) in a benchmark of 100 games. The trend lines are polynomial with a degree of 3.

of the games against the SO-ISMCTS with at most 2 turns look-ahead. We hypothesise that this can be attributed to the combined effects of the following factors:

- Due to the high number of determinizations, the true state (and thus the opponent's follow-up turn) is more likely to be explored.
- Due to the branching factor, even 900 playouts of the `CheatingMctsRollouts` baseline do not lead to a significantly higher exploration depth.
- The high impact of randomness in J4F still prevents a clearly superior but not perfect player from winning all or almost all the games, as indicated by the benchmark of the `CheatingMctsRollouts` agent against the `RandomPlayer` in Subsection 7.1.2.

The best performance for the `IsMctsRollouts` agent with BS, HC, and UC, with a win rate of 48%, was observed with only one determinization per turn.

Fast self-play games for training For training, to generate enough samples without taking a long time in each training iteration, we reduced the number of MCTS-iterations to numbers between 75 and 150. To find good values for the number of determinizations, we repeated the benchmarks described above with 150 playouts on 200 games. Figure 7.5 shows the performance for different determinizations using the minimal information set key-state BS and Figure 7.6 the performance for the largest information set key-state BS, HC, and UC.

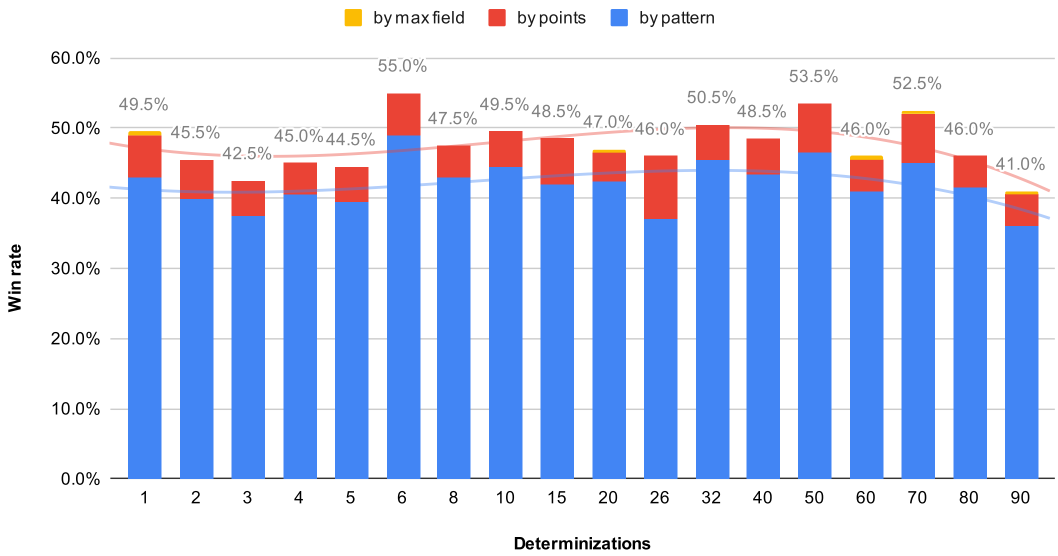


Figure 7.5: Win rate of the `IsMctsRollouts` agent (150 playouts, BS) against the `CheatingMctsRollouts` agent (150 playouts) in a benchmark of 200 games. The trend lines are polynomial with a degree of 3.

In the benchmark of the `IsMctsRollouts` agent with BS and 150 playouts per turn against a `CheatingMctsRollouts` with equal number of playouts, the performance mostly varies between 45% and 50%. Even though the best performance of 55% was observed with 6 determinizations, the most consistent range appears to be between 6 and 15 determinizations.

Similarly to the benchmark with 900 playouts, the performance of the `IsMctsRollouts` agent with BS, HC, and UC generally decreases with an increasing number of playouts. The range with the highest, most consistent performance of 47% to 54% also appears to be in the lower range between 1 and 4 determinizations. There is a slight peak at 8 determinizations which might be an outlier.

In summary, the best performance was observed with a very low number of determinizations, and conversely, a high number of playouts per determinization.

The full experiment configuration is displayed in Table C.7.

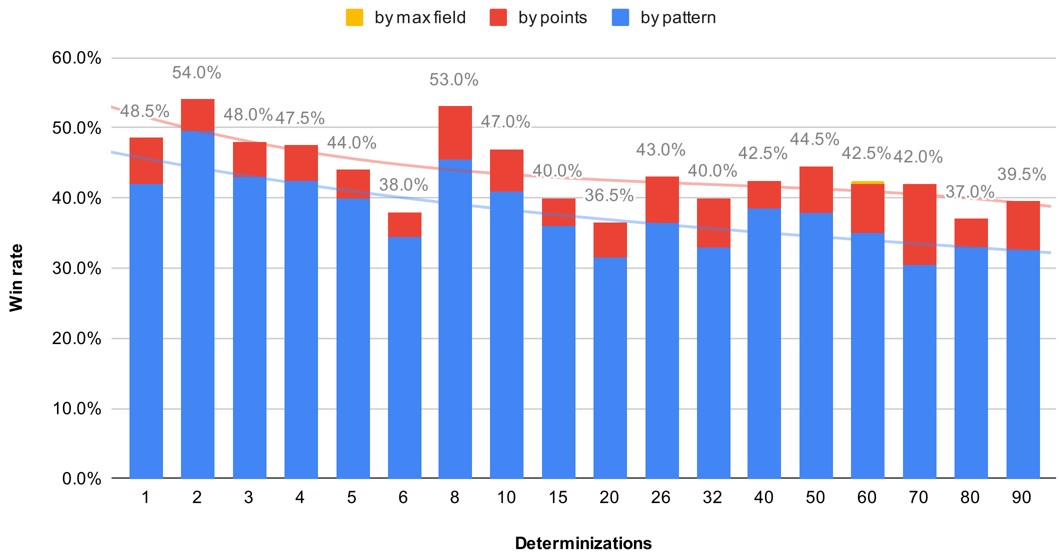


Figure 7.6: Win rate of the `IsMctsRollouts` agent (150 playouts, BS, HC, and UC) against the `CheatingMctsRollouts` agent (150 playouts) in a benchmark of 200 games. The trend lines are polynomial with a degree of 3.

For future work, it would be interesting to investigate the impact of guidance of the DNN on the performance for different numbers of determinizations. As the performance varies considerably throughout the tested range of determinizations, more benchmark games might be beneficial to determine the optimal range.

7.4 Training on Imperfect Information

The initial assumption was that training on perfect information will produce higher quality training samples right from the start and thus accelerate the training process. To verify this hypothesis, we compare the training process of the FieldNet architecture on perfect information and on imperfect information.

Interestingly, this is not the case. A comparison of the training progress of the FieldNet (FNet) architecture over 70 iterations shows that the training benchmark performance when self-play is performed on perfect information increases noticeably slower against the baselines than when also self-play is performed on imperfect information. The training benchmarks, of the `NetworkOnly` against the `RandomPlayer` and the `CheatingMctsRollouts`, are depicted in Figure 7.7a and Figure 7.7b. In each iteration, a total of 600 games was played against the `RandomPlayer`, while `CheatingMctsRollouts` with 50 playouts engaged in 60 games.

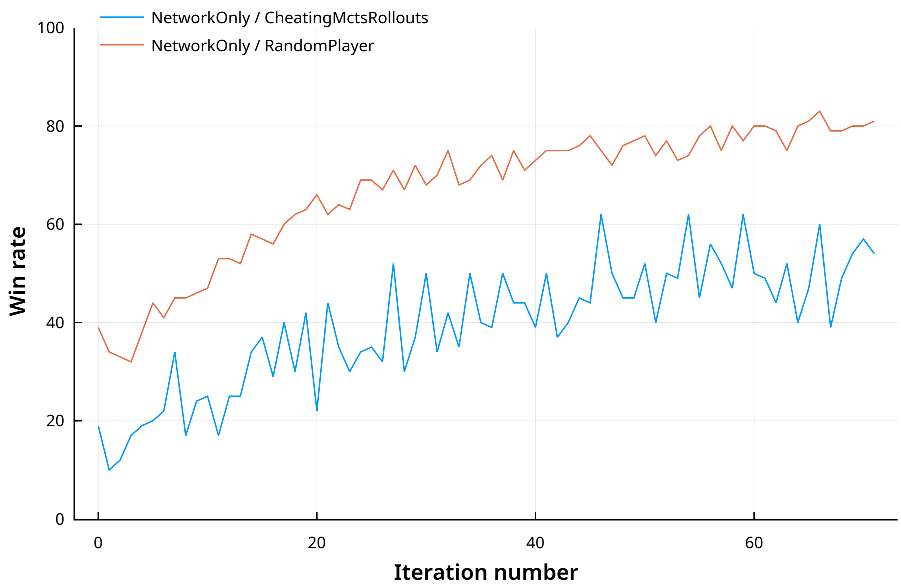
While in the perfect information case, the network reaches a win rate of about 40% against

`CheatingMctsRollouts` between iterations 30 and 40, in the imperfect information case, the network's win rate is already close to 60%.

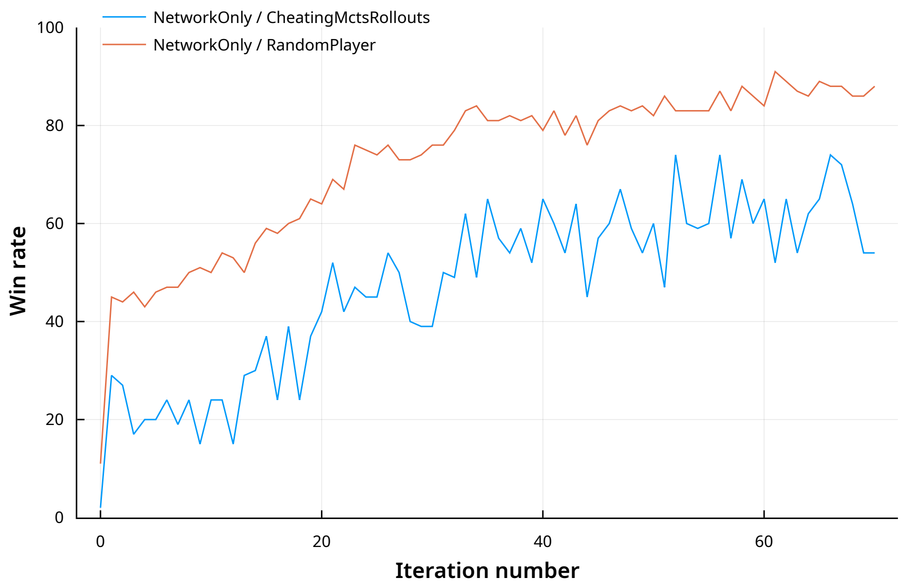
The significant impact on the branching factor introduced by the cards, i.e. the randomness, might be beneficial to the learning process of the network. We suspect this is especially the case for FNet architecture, as it has no direct access to the known public cards-state.

Future work could investigate the peak performance achieved after any number of training iterations, and how quickly this performance is reached in both cases.

The configuration for both agents are in Tables C.8 and C.9.



(a) Self-play with perfect information.



(b) Self-play with imperfect information.

Figure 7.7: Win rate of the `NetworkOnly` agent against `CheatingMctsRollouts` agent, with 50 playouts (based on 60 games) and the `RandomPlayer` (based on 600 games), after each network update. The randomly initialised network was used in training iteration 0.

7.5 Custom Convolutional Filter Kernel Initialisation

We evaluated the effect of using **custom, non-random values** for a subset of the convolutional **kernels of the initial convolution (CKI)** of the networks' board trunk. Our hypothesis is that the training process can be shortened by initialising them with basic patterns.

In our experiments, we use reasonably good hyperparameters for the agent, s.t. its performance against the random baseline (on 800 random games) as well as a simple cheating Monte Carlo Tree Search baseline (on 80 random games) still increases with training. The cheating Monte Carlo Tree Search was configured to do 50 random playouts, which equates to an approximately 90% win rate against the random baseline. Keeping the number of MCTS-iterations per turn fairly low allowed us to get the results in a reasonable amount of time. They were used for two training sessions, one session with randomly initialised and one with CKI.

We chose kernels for detection of horizontal, vertical and diagonal lines. Each kernel k is quadratic with dimension 3×3 and the values sums up to zero, i.e. $\sum_{i=1}^{3}\sum_{j=1}^{3} CF_{k,ij} = 0$. CF_1 for vertical patterns, CF_2 for horizontal patterns and CF_3 and CF_4 for diagonal patterns. They are visualised in Equation (7.1):

$$
\begin{aligned}
\mathrm{CF}_1 &= \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}, \qquad \mathrm{CF}_2 = \begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}, \\
\mathrm{CF}_3 &= \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}, \qquad \mathrm{CF}_4 = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}
\end{aligned} \tag{7.1}
$$

In addition, we trained the FieldNet architecture one time with a minimal set of input features and one time with all the input features we described in Section 5.3. For the training with fewer network inputs, we used the player *stones* plane $\mathbf{I_{stones}}$ (one for the player and one for the opponent), the *empty* fields plane $\mathbf{I_{empty}}$, the *field values* plane $\mathbf{F}$, the *field availability* plane $\mathbf{I_{fa}}$, the *field probability* plane $\mathbf{I_{fp}}$ and the *field reachability* plane $\mathbf{I_{fr}}$. For the training with more network input feature planes, we used the $\mathbf{I_{stones}}$ (for both players), $\mathbf{I_{empty}}$, $\mathbf{I_{minority}}$, $\mathbf{I_{majority}}$, $\mathbf{I_{secured}}$ (for both players), $\mathbf{F}$, $\mathbf{I_{fa}}$, $\mathbf{I_{fp}}$ and $\mathbf{I_{fr}}$.

The results indicate that learning improves significantly in some configurations, i.e. the agent performance increases more quickly when CKI was used.

The win rate against the `RandomPlayer` after the 80th training iteration increased by 4.75% when trained with CKI. Even though the network reached only a win rate of 59.75% (with CKI) against the `RandomPlayer`, the performance curve was not saturated after 80 training iterations. The big difference is that the win rate of about 60% was reached after only 40 iterations with CKI, while without CKI, the win rate was only at 40%.

In the benchmark against `CheatingMctsRollouts`, the win rate even increased by 16.25% when trained with CKI, but only reached 43.75%. The significantly faster performance increase is also reflected in the win rate curve. While the network without CKI reached a win rate of 20% against `CheatingMctsRollouts` after 40 iterations, the network trained with CKI reached the same win rate after only 18 iterations.

When all feature planes were used as network inputs, both networks reached a significantly higher win rate. CKI did not change the overall win rate of 89% against the `RandomPlayer`, but resulted in a slight increase in *wins by points*. Against `CheatingMctsRollouts`, the win-rate even decreased by 7.5% from 75% to 67.5%. When all feature planes were used, with CKI, the network win rate of 40% against `CheatingMctsRollouts` was reached after 20 iterations while without CKI it took 30 training iterations. The benchmark performance against the baselines is summarised in Table 7.4 (`RandomPlayer`) and Table 7.5 (`CheatingMctsRollouts`).

Brief experiments showed that freezing the kernel values, i.e. excluding them from updates, with our custom initialisation led to slightly lower performance.

The complete agent configuration is displayed in Table C.5.

`NetworkOnly` vs. `RandomPlayer` (min)							
CKI	by pattern		by points		by max field	total win rate	
No	395		45		0	55.00%	
Yes	**434**	**(+9.87%)**	43	(-4.44%)	1	**59.75%**	**(+4.75%)**

(a) In 800 games, using minimal input features, the performance in the last iteration against the `RandomPlayer` did increase when CKI was used.

`NetworkOnly` vs. `RandomPlayer` (full)							
CKI	by pattern		by points		by max field	total win rate	
No	687		28		0	89.38%	
Yes	671	(-2.33%)	**41**	**(+46.43%)**	1	89.13%	(-0.25%)

(b) In 800 games, using all input features, the overall performance against `RandomPlayer` did not change when CKI was used. The number of wins by points increased significantly.

Table 7.4: Benchmark performance of `NetworkOnly` against the `RandomPlayer`, with and without CKI, once with minimal network input feature planes (a) and once with all feature planes (b).

NetworkOnly vs. CheatingMctsRollouts (min)				
CKI	by pattern	by points	by max field	total win rate
No	21	1	0	27.50%
Yes	**34** (+61.90%)	1	0	**43.75%** (+16.25%)

(a) In 80 games, using minimal input features, the performance in the last iteration against the `CheatingMctsRollouts` did increase when CKI was used.

NetworkOnly vs. CheatingMctsRollouts (full)				
CKI	by pattern	by points	by max field	total win rate
No	59	1	0	**75.00%**
Yes	**53** (-10.17%)	1	0	**67.50%** (-7.50%)

(b) In 80 games, using all input features, the performance against `CheatingMctsRollouts` even decreased when CKI was used.

Table 7.5: Benchmark performance of `NetworkOnly` against the `CheatingMcts-Rollouts`, with and without CKI, once with minimal network input feature planes (a) and once with all feature planes (b).

7.6 Agent Performance

In this section, we present the results of the AlphaJust4Fun agent's performance evaluation. First, we briefly investigate the test performance of the `IsMctsRollouts` agent on the test sets. We then analyse the AlphaJust4Fun agent's test performance, supporting the assumption that the DNN indeed improves the tree search. Next, we present a full benchmark of all agents, followed by a performance comparison against human players on Yucata.

7.6.1 Test performance

`IsMctsRollouts` Agent Similar to the configuration of the `CheatingMctsRollouts` agent (Subsection 7.1.2, Paragraph: Test performance), we also used 900 playouts per turn. Those were evenly distributed over 4 determinizations. The agent used BS, HC, and UC as information set key-state. The entire configurations of both agents are listed in Tables C.2a and C.2b.

The `IsMctsRollouts` player's average median cross-entropy values are generally very similar to the `CheatingMctsRollouts` player's performance. They are close to zero on the `win-pattern`, `prefer-win-pattern`, `win-pattern-ms`, `win-points-ms`, and `win-max-field-ms` test sets. On the `double-pattern`, `triangle-pattern`, and `prevent-loss-pattern-ms` test sets, the average median values are similarly bad or slightly worse. The only exception is the `prevent-loss-pattern`, where the `CheatingMctsRollouts` player achieves a notably lower average median cross-entropy of 3.24 compared to the `IsMctsRollouts` agent with 5.27. *The `IsMcts-`*

`Rollouts` agent's performance on the `prevent-loss-pattern-ms` test set, which has more diverse action sequences leading to the prevent-loss situations, is closer to the `CheatingMctsRollouts` agent's performance. The similarity suggests that the `IsMctsRollouts` agent might have slightly better adaptability to a wide range of prevent-loss situations. However, the fact that the average median cross-entropy on both prevent-loss test sets is still worse than the one the `RandomPlayer`, suggests that the weakness is likely rooted in the exploration/exploitation/determinization configuration of the Monte Carlo Tree Search-based agents and the low probability of the opponent actually having a winning hand (as we described in Section 6.4). The average median cross-entropy values are compared visually in Figure 7.9.

The average mean cross-entropy for the `CheatingMctsRollouts` player is better than that of the `IsMctsRollouts` agent on most test sets. Only on the `win-max-field-ms` test set, the `IsMctsRollouts` agent performs slightly better than the `CheatingMctsRollouts` agent. Table 7.6 shows the average mean values and average standard deviations of the cross-entropy on each test set for the `IsMctsRollouts` player (highlighted in gray) and the baselines. The values that represent either the best performance or an improvement over either of the baselines are emphasised in bold.

In summary, even with the disadvantage of not knowing the stack of cards and opponent's hands, the `IsMctsRollouts` agent performs surprisingly similarly to the `CheatingMctsRollouts` agent.

Test set	**CheatingMcts-Rollouts** $\mathbf{CE_{mcts}}$	**IsMctsRollouts** $\mathbf{CE_{mcts}}$	**RandomPlayer** $\mathbf{CE_{rand}}$
`win-pattern`	$\mathbf{0.65 \pm 4.16}$	$\mathbf{1.57 \pm 7.12}$	2.81 ± 0.98
`prevent-loss-pattern`	3.21 ± 2.84	3.80 ± 2.85	$\mathbf{2.09 \pm 0.84}$
`prefer-win-pattern`	$\mathbf{0.45 \pm 3.40}$	$\mathbf{1.02 \pm 5.76}$	2.76 ± 0.98
`double-pattern`	5.92 ± 2.19	5.96 ± 2.34	$\mathbf{2.74 \pm 0.96}$
`triangle-pattern`	5.44 ± 2.21	5.49 ± 2.17	$\mathbf{2.50 \pm 0.91}$
`win-pattern-ms`	$\mathbf{0.69 \pm 4.54}$	$\mathbf{1.60 \pm 7.16}$	2.73 ± 0.98
`prevent-loss-pattern-ms`	4.06 ± 2.98	4.63 ± 2.87	$\mathbf{2.49 \pm 0.94}$
`win-points-ms`	$\mathbf{0.70 \pm 2.91}$	1.39 ± 5.29	0.93 ± 0.75
`win-max-field-ms`	3.26 ± 7.76	$\mathbf{2.95 \pm 7.05}$	$\mathbf{2.48 \pm 0.89}$

Table 7.6: Comparison of the mean values and standard deviations of the cross-entropy for the `IsMctsRollouts` player (MCTS policy; 900 MCTS-iterations on 4 determinizations in each turn; 60 repetitions), the `CheatingMctsRollouts` agent (MCTS policy; 900 MCTS-iterations in each turn; 60 repetitions) and the `RandomPlayer` (120 repetitions), averaged ($\overline{\mu} \pm \overline{\sigma}$) over multiple repetitions of each test set.

FieldNet-based AlphaJust4Fun Agent We evaluated an FieldNet-based AlphaJust4Fun agent that uses, and also has used during training, BS, HC, and UC as information set key-state. It also performed 900 playouts, but distributed over 3 determinizations. The entire configurations of the cheating MCTS agent and the FNet-based AZJ4F agent are in Tables C.2a and C.13b.

Similar to the other MCTS-based agents, it is typically capable of recognising actions that immediately result in a win, which are represented by the `win-pattern`, `prefer-win-pattern`, `win-pattern-ms`, `win-points-ms`, and `win-max-field-ms` test sets. This is indicated by average median cross-entropy values of zero. On the `prevent-loss-pattern`, `prevent-loss-pattern-ms`, and `double-pattern` test sets, it's MCTS-based policies outperform the baselines with average median cross-entropy values of 0.91, 1.53, and 0.36, respectively. On the `triangle-pattern` test set, the MCTS-based policies achieve a significantly better average median cross-entropy than the two MCTS-only agents, but it is still slightly worse than the random baseline. As a result, it is not clear whether it can exploit this strategy. The average median cross-entropys of the network-based policies is generally higher than those of the MCTS-based policies, except for the `triangle-pattern`, where it is slightly lower. On the `win-pattern`, `prefer-win-pattern`, and `win-pattern-ms` test sets, it is comparable to the MCTS-based values, close to zero. Figure 7.9 visualises the average median cross-entropy.

The average mean values and average standard deviations of the cross-entropy, calculated from MCTS-policies, show improvement over the baselines on most test sets. On the `triangle-pattern` it's performance is in a similar range as the random baseline. However, in some instances of the `win-max-field-ms` test set, suboptimal policies are generated, which is reflected in the relatively high average mean value and average standard deviation of 5.25 ± 11.52. `AlphaJust4FunZeroPlayer`'s network-based average mean values and average standard deviations are mostly in a similar range or better than the baselines. On the `triangle-pattern` test set, it is even slightly better than the agent's MCTS-based values. Table 7.7 shows the average mean values and the average standard deviations on each test set for the FNet-based `AlphaJust4FunZeroPlayer` (BS, HC, and UC) and the baselines. The values that represent an improvement over the baselines and, for the network-based cross-entropy, the values that represent an improvement over the MCTS-based cross-entropy, are highlighted in bold.

In summary, the FNet-based AlphaJust4Fun agent only showed notable deficiencies on the `triangle-pattern` and `win-max-field-ms` test sets, with suboptimal average median values on both and a particularly high average standard deviation on the `win-max-field-ms` test set. These results clearly show that the DNN substantially enhances the tree search policy over the baselines and the `IsMctsRollouts` agent.

Test set	**AlphaJust4FunZero-Player** (FNet) $\mathbf{CE_{mcts}}$	**CheatingMcts-Rollouts** $\mathbf{CE_{mcts}}$	**RandomPlayer** $\mathbf{CE_{rand}}$
`win-pattern`	$\mathbf{0.02 \pm 0.20}$	0.65 ± 4.16	2.81 ± 0.98
`prevent-loss-pattern`	$\mathbf{1.13 \pm 1.09}$	3.21 ± 2.84	2.09 ± 0.84
`prefer-win-pattern`	$\mathbf{0.01 \pm 0.10}$	0.45 ± 3.40	2.76 ± 0.98
`double-pattern`	$\mathbf{1.11 \pm 1.76}$	5.92 ± 2.19	2.74 ± 0.96
`triangle-pattern`	2.81 ± 1.65	5.44 ± 2.21	$\mathbf{2.50 \pm 0.91}$
`win-pattern-ms`	$\mathbf{0.02 \pm 0.26}$	0.69 ± 4.54	2.73 ± 0.98
`prevent-loss-pattern-ms`	$\mathbf{1.73 \pm 1.71}$	4.06 ± 2.98	2.49 ± 0.94
`win-points-ms`	0.80 ± 3.29	$\mathbf{0.70 \pm 2.91}$	0.93 ± 0.75
`win-max-field-ms`	5.25 ± 11.52	3.26 ± 7.76	$\mathbf{2.48 \pm 0.89}$

(a) Cross-entropy of the MCTS policy, averaged ($\overline{\mu} \pm \overline{\sigma}$) over multiple repetitions of each test set.

Test set	**AlphaJust4FunZeroPlayer** (FNet) $\mathbf{CE_{net}}$	$\mathbf{CE_{net}^{pre}}$
`win-pattern`	0.18 ± 0.40	0.18 ± 0.40
`prevent-loss-pattern`	2.21 ± 1.03	2.21 ± 1.03
`prefer-win-pattern`	0.17 ± 0.36	0.17 ± 0.36
`double-pattern`	1.99 ± 0.98	1.99 ± 0.98
`triangle-pattern`	$\mathbf{2.42 \pm 0.99}$	$\mathbf{2.42 \pm 0.99}$
`win-pattern-ms`	0.21 ± 0.41	0.21 ± 0.41
`prevent-loss-pattern-ms`	2.43 ± 1.02	2.43 ± 1.02
`win-points-ms`	1.07 ± 1.11	1.07 ± 1.12
`win-max-field-ms`	$\mathbf{2.70 \pm 2.01}$	$\mathbf{2.70 \pm 2.01}$

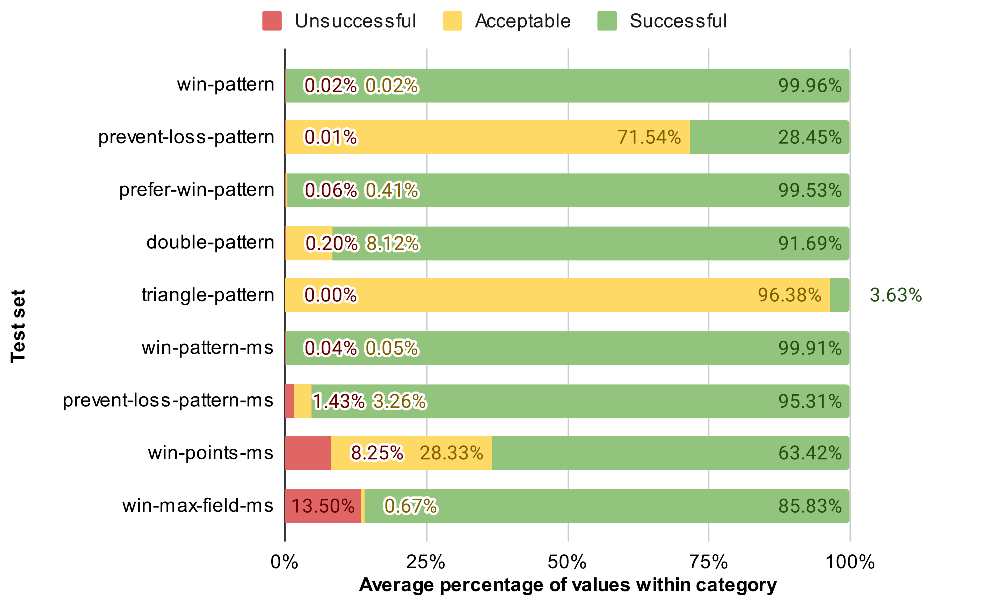
(b) Cross-entropy of the network policy, averaged ($\overline{\mu} \pm \overline{\sigma}$) over multiple repetitions of each test set.

Table 7.7: Comparison of the cross-entropy metrics for the FNet-based AlphaJust4Fun agent (MCTS policy (a) and network policy (b); 900 MCTS-iterations over 3 determinizations in each turn; using BS, HC, and UC; 60 repetitions), the `CheatingMctsRollouts` (MCTS policy; 900 MCTS-iterations in each turn; 60 repetitions) and the `RandomPlayer` (120 repetitions).

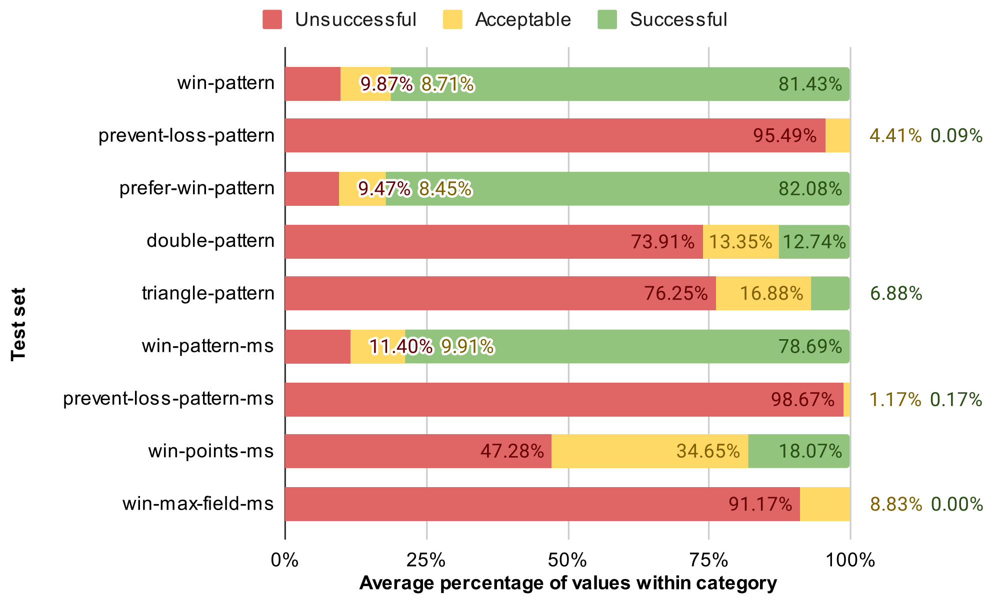
UCT-value and $\mathbf{P_{net}}$ quality After investigating the deficiencies mentioned before, we found the network's policies to be the primary cause. This becomes apparent in our rating of $\mathrm{cP_{net}}$-values, where a high fraction of the values were rated unsuccessful, and small fractions acceptable and successful. Most of the $\mathrm{cP_{net}}$-values rated successful were in situations where a pattern-win is imminent. We suspect that overfitting might be one issue, which is supported by the surprising fact that the cross-entropy metrics of the network policies, before and after masking, were the same. The action values $\overline{\mathrm{Q}}_{\mathrm{net,scaled}}$ are not included in this work, but were reasonably good, with most of the values rated successful and only very little rated unsuccessful. However, the network's value output turned out to be less sensitive to state changes than expected during real game play. This is also reflected by our rating of the V_{net}-values. For each test set, all the values fall into the same rating category. Figure 7.8 shows our rating of $\overline{\mathrm{UCT}}_{\mathrm{scaled}}$ and $\mathrm{cP_{net}}$, and

Figure 7.10c shows the rating of V_{net} for all test sets.

We also evaluated a FieldNet-based AlphaJust4Fun agent that uses BS as information set key-state. However, the MCTS-based cross-entropy values are not displayed, but were mostly in ranges similar to the agent that uses BS, HC, and UC. On the `triangle-pattern` and the `prevent-loss-pattern-ms` test sets, the BS-version performed only slightly better, but on the `double-pattern` test set, it performed significantly worse. In the remainder of this work, we will exclusively refer to the FieldNet-based AlphaJust4Fun agent that uses BS, HC, and UC as information set key-state.



(a) $\overline{\mathrm{UCT}}_{\mathrm{scaled}}$



(b) $\mathrm{cP}_{\mathrm{net}}$

Figure 7.8: Percentage of values for the expected actions, calculated by the **FNet-based AlphaJust4Fun** agent (with 900 MCTS-iterations over 3 determinizations in each turn, using BS, HC, and UC), per rating category for each test set (based on 60 repetitions of each test set).

CardFieldNet-based AlphaJust4Fun Agent Our final AlphaJust4Fun agent is based on the CardFieldNet and uses, and also has used during training, BS, HC, and UC as information set key-state. It performs 900 playouts per turn, like the other tested agents, distributed over only 3 determinizations. The entire configurations of the cheating MCTS agent and the CFNet-based AZJ4F agent are in Tables C.2a and C.11b.

The average median cross-entropy of its MCTS-policies outperforms all other agents in almost all test sets. The most significant difference can be observed on the test sets, where the terminal reward for a win was further from the tested state, i.e. the `prevent-loss-pattern`, `prevent-loss-pattern-ms`, `double-pattern`, and `triangle-pattern`. Despite having a lower average median cross-entropy of 2.34 than the other MCTS-based agents, it is still not quite lower than the `RandomPlayer` baseline with 2.27 on the `triangle-pattern` test set. In contrast to the FNet-based agent, the average mean cross-entropy of the network-based policies of the CFNet, before and after masking, are different on all test sets. We therefore conclude that overfitting is much less of an issue for this network and training configuration. In fact, the average cross-entropy metrics of the network policy before masking is lower than the one of the policy after masking in all test sets. On the `triangle-pattern`, the network-based policies have the lowest average median cross-entropy, as well as the lowest average mean value and the average standard deviation, among all tested agents. Also on the `double-pattern` test set, at least the average mean value and the average standard deviation of the cross-entropy of the network-based policies, before masking, are also among the lowest of all agents and close to those of the MCTS-based policies. On the `win-pattern`, `prefer-win-pattern`, `win-pattern-ms`, `win-points-ms`, and `win-max-field-ms` test sets, where a win can be achieved with the next action, the average median cross-entropy values of the network-based policies are high in comparison to the other agents. This hints that the network might have better captured strategies that are more successful in longer games. Figure 7.9 visualises the performance, based on the averaged median cross-entropy, of every tested agent on every test set.

The average mean values and the average standard deviations of the cross-entropy values of the MCTS-based policies are generally lower than the other agents' MCTS-based cross-entropy. Only on the `prefer-win-pattern` test set, FNet-based agent's average mean cross-entropy is the lowest. While the average median cross-entropy is 0 on the `win-max-field-ms` test set, the average mean and average standard deviation of between 3.95 and 10.03 are still slightly higher than those of the `CheatingMctsRollouts` agent. Table 7.8 shows the average cross-entropy metrics on each test set for the CFNet-based `AlphaJust4FunZeroPlayer` and the baselines. The values that represent an improvement over the baselines and, for the network-based cross-entropy, the values that represent an improvement over the MCTS-based cross-entropy, are highlighted in bold.

In summary, the CFNet-based AlphaJust4Fun agent only shows notable deficiencies on the `triangle-pattern` tests, with the median values still only at the same cross-entropy as the `RandomPlayer`. On all other test sets, this agent shows lower median values and also mostly lower means and standard deviations of the cross-entropy than the

Test set	AlphaJust4FunZero-Player (CFNet) $\mathbf{CE_{mcts}}$	CheatingMcts-Rollouts $\mathbf{CE_{mcts}}$	RandomPlayer $\mathbf{CE_{rand}}$
win-pattern	$\mathbf{0.08 \pm 0.80}$	0.65 ± 4.16	2.81 ± 0.98
prevent-loss-pattern	$\mathbf{1.04 \pm 1.52}$	3.21 ± 2.84	2.09 ± 0.84
prefer-win-pattern	0.72 ± 4.85	$\mathbf{0.45 \pm 3.40}$	2.76 ± 0.98
double-pattern	$\mathbf{1.04 \pm 1.99}$	5.92 ± 2.19	2.74 ± 0.96
triangle-pattern	$\mathbf{2.59 \pm 2.63}$	5.44 ± 2.21	$\mathbf{2.50 \pm 0.91}$
win-pattern-ms	$\mathbf{1.03 \pm 5.88}$	0.69 ± 4.54	2.73 ± 0.98
prevent-loss-pattern-ms	$\mathbf{2.29 \pm 4.87}$	4.06 ± 2.98	2.49 ± 0.94
win-points-ms	$\mathbf{0.51 \pm 2.06}$	0.70 ± 2.91	0.93 ± 0.75
win-max-field-ms	3.92 ± 10.12	3.26 ± 7.76	$\mathbf{2.48 \pm 0.89}$

(a) Cross-entropy of the MCTS policy, averaged ($\overline{\mu} \pm \overline{\sigma}$) over multiple repetitions of each test set.

Test set	AlphaJust4FunZeroPlayer (CFNet) $\mathbf{CE_{net}}$	$\mathbf{CE_{net}^{pre}}$
win-pattern	1.47 ± 1.24	1.02 ± 1.09
prevent-loss-pattern	$\mathbf{1.56 \pm 0.77}$	$\mathbf{1.47 \pm 1.09}$
prefer-win-pattern	1.98 ± 1.37	1.45 ± 1.25
double-pattern	$\mathbf{1.63 \pm 1.15}$	$\mathbf{1.18 \pm 1.08}$
triangle-pattern	$\mathbf{1.90 \pm 0.73}$	$\mathbf{1.56 \pm 0.95}$
win-pattern-ms	1.71 ± 1.20	1.20 ± 1.12
prevent-loss-pattern-ms	$\mathbf{2.11 \pm 0.98}$	$\mathbf{1.69 \pm 1.16}$
win-points-ms	1.01 ± 0.58	0.94 ± 0.92
win-max-field-ms	3.30 ± 0.61	$\mathbf{2.24 \pm 0.72}$

(b) Cross-entropy of the network policy, averaged ($\overline{\mu} \pm \overline{\sigma}$) over multiple repetitions of each test set.

Table 7.8: Comparison of the cross-entropy metrics for the CFNet-based AlphaJust4Fun agent (MCTS policy (a) and network policy (b); 900 MCTS-iterations over 3 determinizations in each turn; using BS, HC, and UC; 60 repetitions), the `CheatingMctsRollouts` player (MCTS policy; 900 MCTS-iterations in each turn; 60 repetitions) and the `RandomPlayer` (120 repetitions).

baselines. *The results indicate that the DNN significantly improves the tree search policy and even outperforms the baselines on its own on some test sets.* The general superiority over the baselines is displayed more clearly in the benchmark in Subsection 7.6.3.

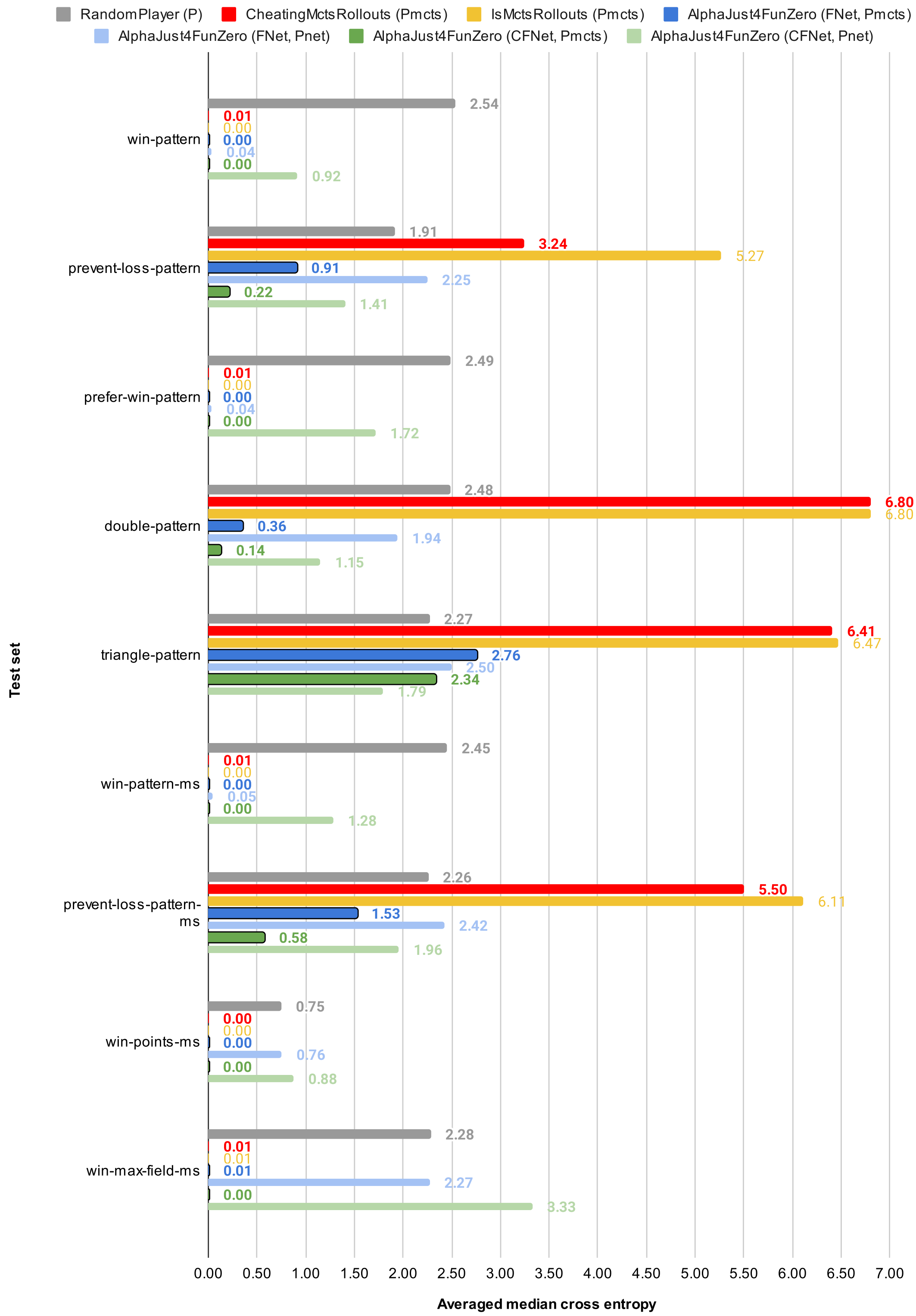
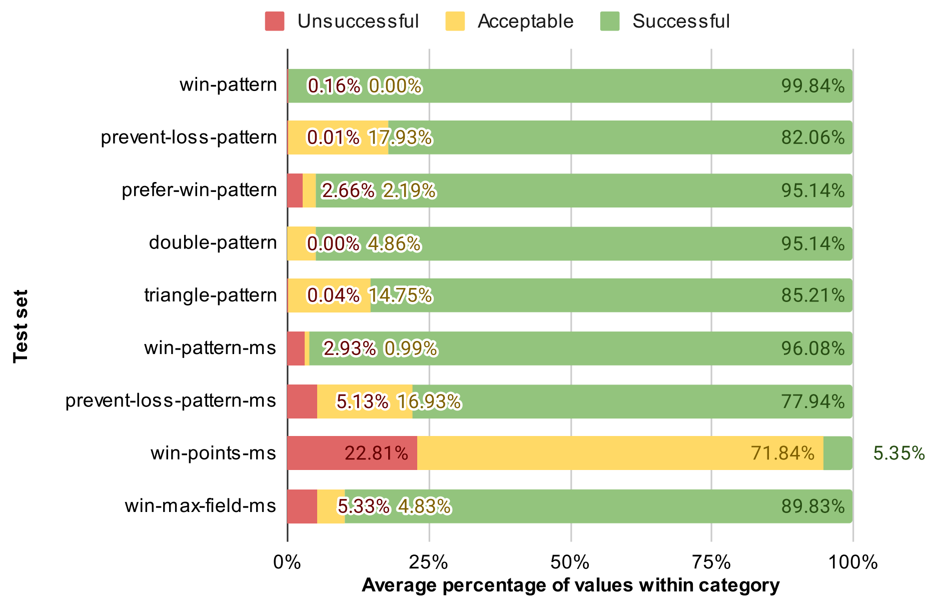


Figure 7.9: Median cross-entropy of agent policies, averaged per test set over of all repetitions of the test set.

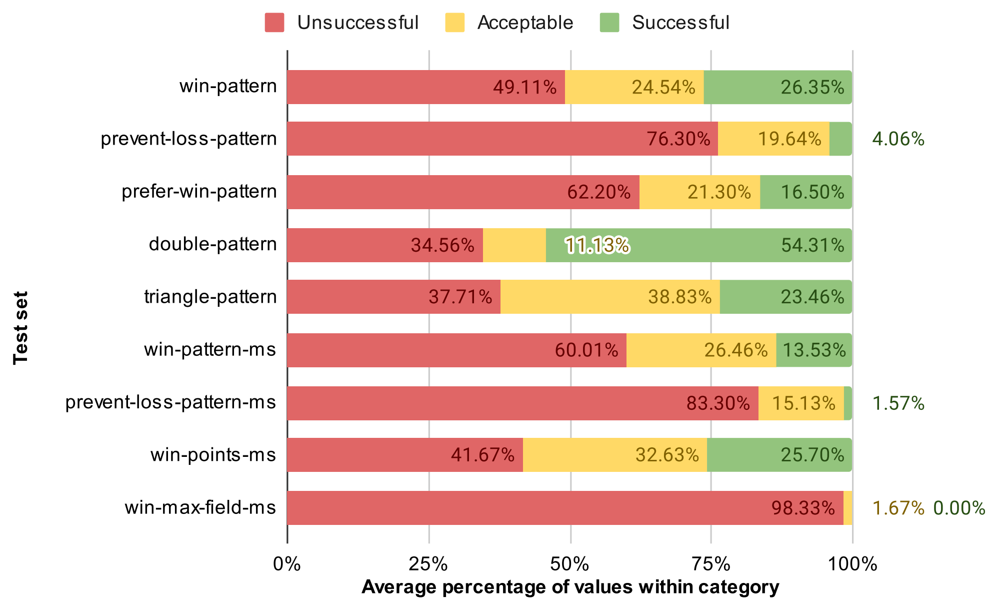
UCT-value quality On the `prevent-loss-pattern` and `triangle-pattern` test sets, the CFNet-based agent performs is notably better than the FNet-based agent. On the test sets that require the agent to find an action that results in a win by pattern, the FNet-based agent performs slightly better. On the `win-max-field-ms` test set, the CFNet-based agent preforms slightly better, but our rating on the `win-points-ms` test set is, with 23% rated unsuccessful, 72% acceptable, and only 5% successful, notably worse for the CFNet-based agent. The FNet-based agent did significantly better, with only 8% unsuccessful, 28% acceptable and 63% successful. According to this metric, the `win-points-ms` test set is the most challenging for the AZJ4F agents. This indicates general difficulties in estimating the value of those rather trivial situations appropriately. *The reason is that in those game states, there are usually several possible actions that make the agent lose the game. In combination with an overly exploratory agent configuration, this can result in lower than expected action values.* Figure 7.10a show the rating of $\overline{\text{UCT}}_{\text{scaled}}$ on the test sets.

$\mathbf{P_{net}}$ quality The ratings of the network-policy values follow a similar trend as $\overline{\text{UCT}}_{\text{scaled}}$. However, the performance on the `prevent-loss-pattern`, `triangle-pattern`, but also the `prevent-loss-pattern-ms` and `double-pattern` test sets, show improvements over the FNet-based agent. The performance on the `win-pattern`, `prefer-win-pattern`, and `win-pattern-ms` has worsened notably.

On the `prevent-loss-pattern` test set, the CFNet-based agent's values were rated unsuccessful in 76% of the test cases, whereas the FNet-based agent's were rated unsuccessful in 95% of the test cases. On the `prevent-loss-pattern-ms` test set, CFNet's values were rated unsuccessful in 83%, and FNet's values in 99% of the test cases. On the `double-pattern` and `triangle-pattern` test sets, the differences are more significant with only 35% and 38% rated unsuccessful for CFNet, and 74% and 76% unsuccessful test cases for FNet. In game states that are closer to a win, i.e., `win-pattern`, `prefer-win-pattern`, and `win-pattern-ms`, the FNet-based agent performed significantly better. *A reason for the advantage of the FNet-based agent on the pattern-win test sets might be the training duration or the number of parameters. However, it might have come at the cost of overfitting in the situations mentioned above, at the end of the section describing the FNet-based agent.* The FieldNet (26,143 parameters) was trained for 223 MCTS-iterations, whereas the CardFieldNet (64,445 parameters) for 128. We ended the training after 128 MCTS-iterations as the performance in the training benchmark (against `CheatingMctsRollouts` with 50 MCTS-iterations) did not increase anymore. Furthermore, the number of games that ended in a win by points did not increase in the self-play games nor the training benchmark. Figure 7.10b shows the ratings of the network-policy values.



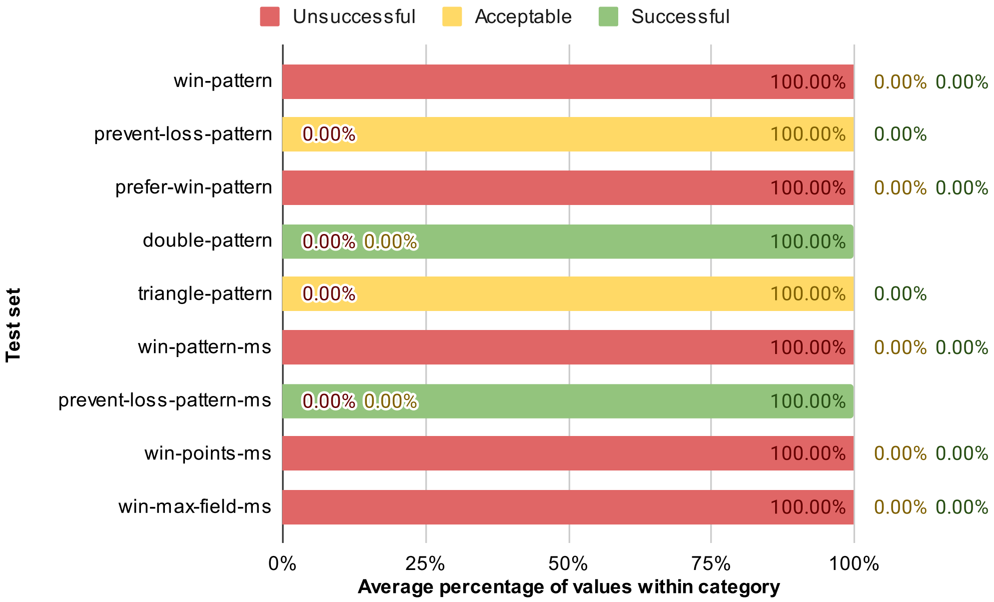
(a) $\overline{\mathrm{UCT}}_{\mathrm{scaled}}$



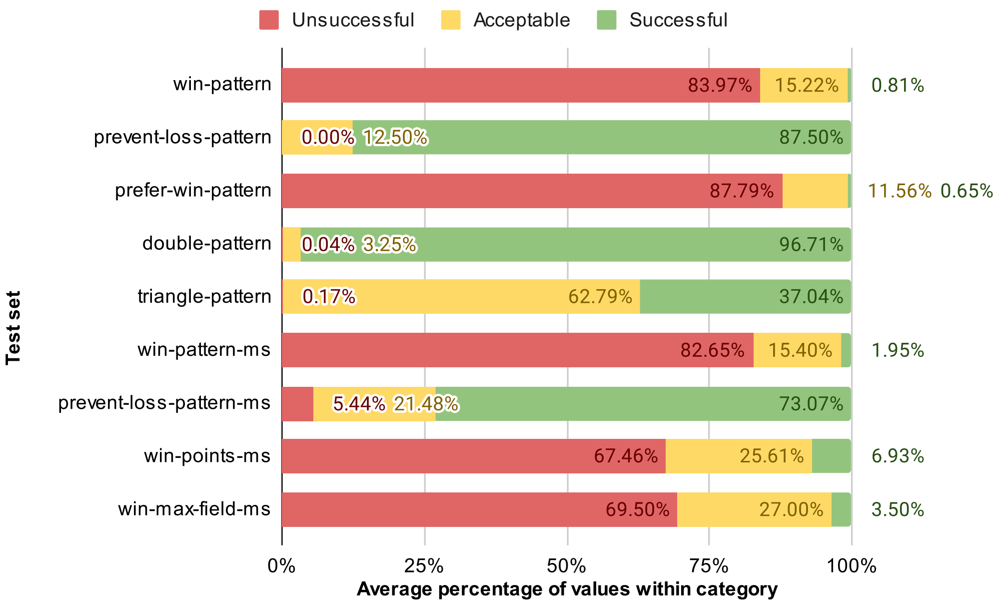
(b) $\mathrm{cP}_{\mathrm{net}}$

Figure 7.10: Percentage of values for the expected actions, calculated by the **CFNet-based AlphaJust4Fun** agent (with 900 MCTS-iterations over 3 determinizations in each turn, using BS, HC, and UC), per rating category for each test set (based on 60 repetitions of each test set).

$V_{\mathbf{net}}$-value quality From our ratings of the network's value estimations V_{net}, it is clear that the CardFieldNet is overall able to differentiate better between game state values compared to the FieldNet. We also observed this major issue of FNet during interactive game simulations, where V_{net} appeared to be almost constant for most of the states. Figure 7.10 shows the rating of the network-values.



(c) FieldNet



(d) CardFieldNet

Figure 7.10: Percentage of V_{net}-values for the expected actions, calculated by the AlphaJust4Fun agent (with 900 MCTS-iterations over 3 determinizations in each turn, using BS, HC, and UC), per rating category for each test set (based on 60 repetitions).

7.6.2 Training

It was very challenging to find a working hyperparameter combination for the CFNet architecture. Only after experimenting with different activation functions, weight initialisation functions, gradient clipping, dropout, batch normalisation, and employing the network aspects described in Section 6.4, we were able to make the training work. We mainly attribute this success to the mix of convolutional and dense networks and the structures that combine both of them into a single dense network.

The FNet architecture, on the other hand, is less sensitive to the hyperparameter configuration and also more stable during the first 50 training iterations.

The discount factor γ appears to be vital to the training process, as it emphasises the uncertainty towards the beginning of the game. We suspect that it aids the FNet-based agent more than the CFNet-based agent, as the latter already receives direct information about the cards state as an input.

7.6.3 Benchmark performance

In this section, we present the results of the extensive benchmark among all agents and the baselines. The win rate in each agent match-up per game end reason is visualised in Figure 7.11, Table 7.9a shows the number of games for each agent match-up, and Table 7.9b the key hyperparameters for each agent. Since the starting player was selected

<table>
<tr><th>Agent</th><th>IsMctsRollouts</th><th>NetworkOnly</th><th>RandomPlayer</th><th>CheatingMctsRollouts</th></tr>
<tr><td>AlphaJust4FunZeroPlayer</td><td>100</td><td colspan="2" rowspan="2">200</td><td rowspan="2">100</td></tr>
<tr><td>IsMctsRollouts</td><td></td></tr>
<tr><td>NetworkOnly</td><td colspan="2"></td><td>900</td><td rowspan="2">200</td></tr>
<tr><td>RandomPlayer</td><td colspan="3"></td></tr>
</table>

(a) The number of benchmark games for each agent match-up.

Agent	MCTS-iterations	Information set size	Determinizations
AlphaJust4FunZeroPlayer	900	BS, HC, and UC	3
IsMctsRollouts	900	BS, HC, and UC	3
NetworkOnly	n/a		
CheatingMctsRollouts	900	n/a	
RandomPlayer	n/a		

(b) Key hyperparameters for the agents in the benchmark.

Table 7.9: Settings of the FieldNet and CardFieldNet benchmarks.

randomly, the number of starts each player had, was not always equal. Figure 7.12 shows the number of times each agent was the first one to act.

We performed a similar benchmark for the FieldNet with information set key-state being BS only, but its performance was slightly worse than the one with BS, HC, and UC.

`AlphaJust4FunZeroPlayer` vs. `CheatingMctsRollouts` Both, the FieldNet- and the CardFieldNet-based AlphaJust4Fun agents performed solidly against the cheating MCTS agent with an equal number of MCTS-iterations.

The CardFieldNet-based agent performed especially well with a win rate of 94%. The FieldNet-based agent won at least in 74% of the games.

The FieldNet-based agent lost in 22% of the games by pattern, whereas CardFieldNet-based agent lost only 3% by pattern. The lesser ability of the FieldNet-based agent to prevent opponent wins by pattern, compared to the CardFieldNet-based, is also reflected in their performance on the `prevent-loss-pattern` and `prevent-loss-pattern-ms` test sets shown in Figure 7.9.

`AlphaJust4FunZeroPlayer` vs. `IsMctsRollouts` From this confrontation, it is clear that the DNN significantly enhances the performance of the SO-ISMCTS. The CardFieldNet-based agent won 92% and the FieldNet-based agent won 89% of the games. The main difference between the two agents was that the CardFieldNet-based agent won more games by pattern.

`AlphaJust4FunZeroPlayer` vs. `NetworkOnly` This confrontation illustrates that the DNN without the SO-ISMCTS is considerably weaker than the combined system. The FieldNet agent won 12% of the games, and the CardFieldNet agent won only 5%.

`IsMctsRollouts` vs. `CheatingMctsRollouts` The isolated performance of the SO-ISMCTS against the baseline is not nearly as good as the combination with a DNN. The `IsMctsRollouts` agent won 45.5% of the games.

`NetworkOnly` vs. `CheatingMctsRollouts` Also, the isolated performance of the DNN against the baseline is not as good as the combination. The FieldNet agent won 46% of the games and the CardFieldNet agent won at least 59% of the games. This also shows the superiority of the CardFieldNet architecture and its learning success.

`RandomPlayer` vs. `CheatingMctsRollouts` In the Just 4 Fun game setting, with the constraints imposed by the cards, even a random policy can win in 9.25% of the games against a cheating MCTS agent. Interestingly, half of those games were even won by pattern. Conversely, in line with expectations, the primary cause of losses was a pattern win by the cheating MCTS agent.

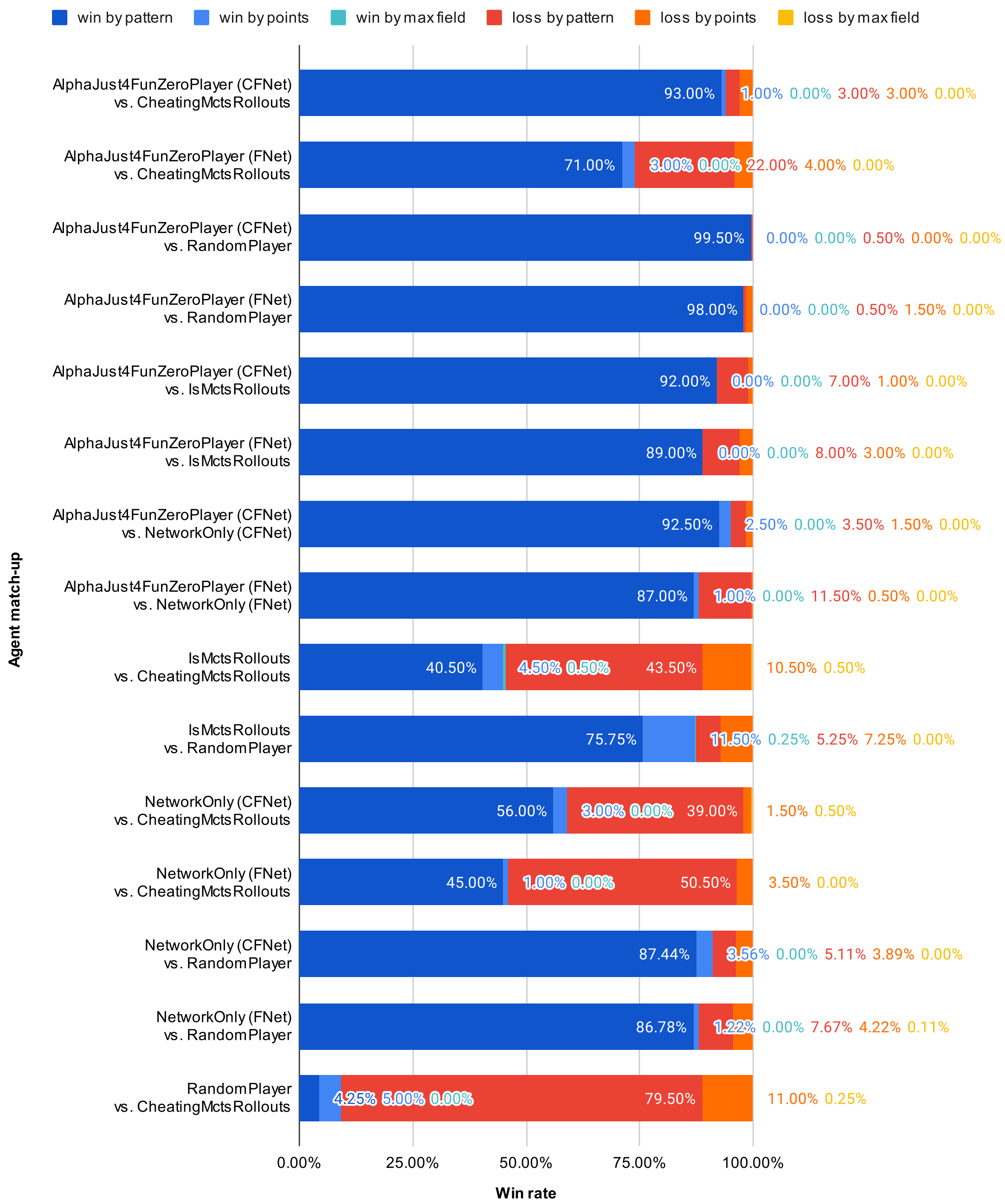


Figure 7.11: Benchmarks of all agents. The agents performed 900 MCTS-iterations (MCTS and SO-ISMCTS) and 3 determinizations (SO-ISMCTS only) per turn and BS, HC, and UC as information set key-state (SO-ISMCTS only). The percentages are for the agent named first in the match-up.

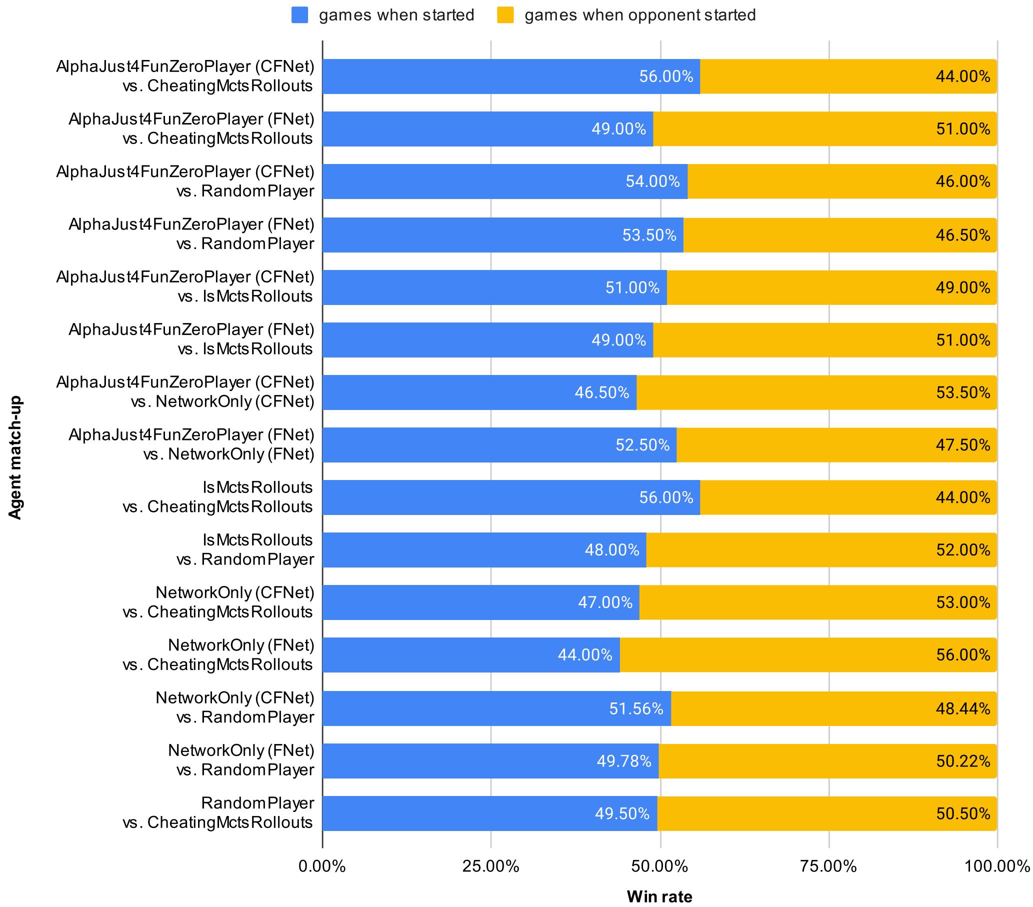


Figure 7.12: Percentage of the randomly initialised games in which each agent was the first player to act. The left bar represents the agent named first in the match-up.

Impact of the number of playouts in the `RandomPlayer` vs. `CheatingMcts-Rollouts` match-up To determine the number of MCTS-iterations a cheating MCTS agent needs to achieve a 100% win rate against the `RandomPlayer`, we ran benchmarks with various numbers of iterations. Besides the complete benchmark with 900 MCTS-iterations presented at the beginning of this section, we also ran benchmarks using 50, 1500, 3000, and 8000 MCTS-iterations, testing only the `CheatingMctsRollouts` agent and the `RandomPlayer`.

The cheating MCTS agent reached a win rate of above 96% only after performing $8,000$ MCTS-iterations per turn. This is still worse than the AlphaJust4Fun agent with either network. *The high number of MCTS-iterations, the cheating MCTS agent required to achieve a very high win rate against the `RandomPlayer` might partly due to suboptimal exploitation behaviour.* The number of iterations and the overall win rate is shown in Table 7.10.

MCTS-iterations per turn	Win rate	Games	Percentage of games in which the `RandomPlayer` started
50	79.17%	600	51.67%
900	90.75%	400	49.50%
1,500	91.00%	200	51.00%
3,000	93.33%	120	53.33%
8,000	96.67%	120	55.00%

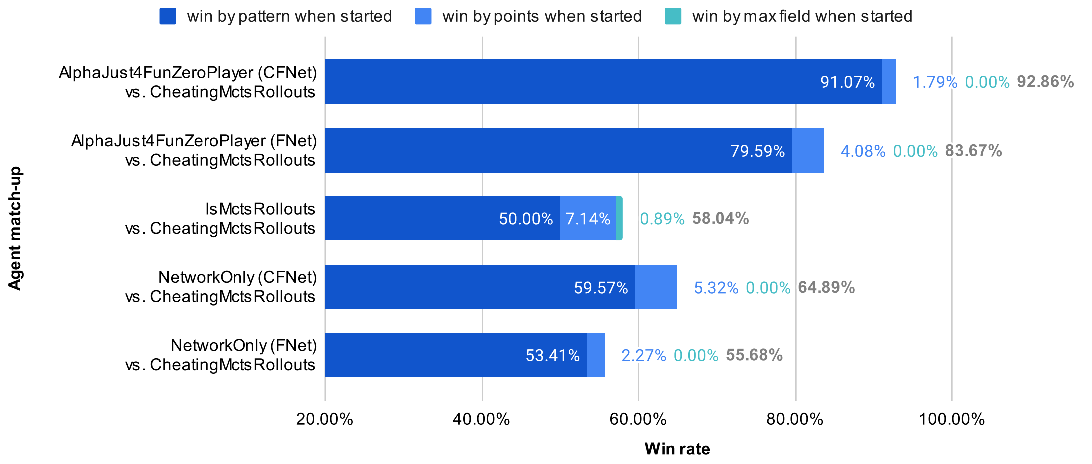
Table 7.10: Results of the benchmarks of a `CheatingMctsRollouts` agent with different numbers of MCTS-iterations against the `RandomPlayer`.

Impact of being the starting player for different agent match-ups Being the starting player gave the cheating MCTS agent an advantage in most match-ups. The number of wins by points decreased in all match-ups when the cheating MCTS agent was the player who started.

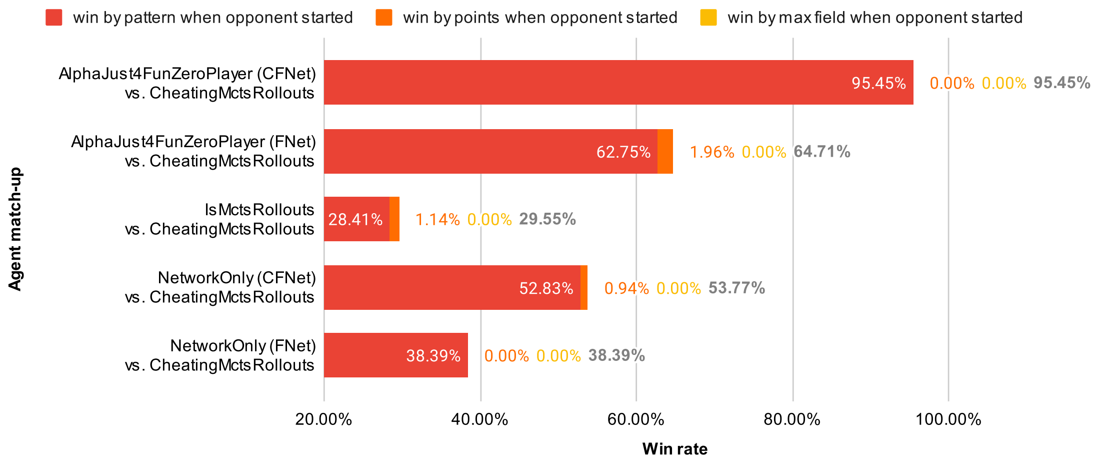
Being the starting player had the least impact on the match-up against the CFNet-based AlphaJust4Fun agent. Interestingly, when not being the starting player, its win rate even increased by 2.59%. However, this might be a coincidence and a result of the limited number of games in this match-up. The FNet-based AlphaJust4Fun agent's win rate was 20% less when the cheating MCTS agent started. The most significant reduction in win rate of 28.49%, when being the follow-up player, was observed in the `IsMctsRollouts` match-up. In the match-up against the CardFieldNet alone, the win rate decreased by only 11.12%. The FieldNet agent's win rate decreased by about 17%.

The win rates for the agent named first in the match-up, depending on when they started or were the follow-up player, are visualised in Figure 7.13. The training configuration of the FieldNet-based AlphaJust4Fun agent is listed in Table C.12 and the configuration used in the benchmark is listed in Table C.13b. The training configuration of the

CardFieldNet-based AlphaJust4Fun agent is listed in Table C.10 and the configuration used in the benchmark is listed in Table C.11b. The configuration of the cheating MCTS agent and the SO-ISMCTS agent that were used for the benchmark are listed in Tables C.4a and C.4b.



(a)



(b)

Figure 7.13: Win rate based on 100 randomly initialised games for the agent named first in each match-up, depending on whether it started (a) the game or was the follow-up player, i.e. the opponent started (b).

7.6.4 Performance against human players

We set up public games with the required configuration options on Yucata. The opponents, except for one player, were not selected by us, but chose by themselves to join our games. Any player at any skill level was free to join the games. To speed up the benchmark, a colleague of the author played a few games. He is an experienced chess player, but was new to Just 4 Fun. He won 5 out of 5 games against the FNet-based agent and 4 out of 9 games against the CFNet-based agent (Player G in Table 7.11).

FieldNet-based AlphaJust4Fun Agent For play against humans, using the FieldNet-based AlphaJust4Fun agent, the account **AlphaJ4FZeroFNet**[3] was used. The agent has been configured with 8,000 MCTS-iterations, 700 determinizations and to utilise BS, HC, and UC in the tree search. This means that 11 MCTS-iterations per determinization were performed, which results in 7,700 overall MCTS-iterations per turn.

The agent can beat human players, but the policy was often objectively bad. It is not good enough to compete with skilled human players. *This agent configuration uses more determinizations relative to the number of MCTS-iterations, compared to the configuration used in the benchmark of agents. This might also contribute to a weaker performance, as it results in more exploration.*

The agent won 4 out of 15 games, i.e. the win rate was 26.67%. The performance in terms of wins/losses and TrueSkill-change is displayed in Figure 7.14a. The agent configuration is listed in Table C.13a.

CardFieldNet-based AlphaJust4Fun Agent For play against humans, using the CardFieldNet-based AlphaJust4Fun agent, the account **AlphaJust4Fun**[4] was used. Since the 8,000 MCTS-iterations of the FieldNet-based agent did not appear to be enough to outperform experienced players, we configured CardFieldNet-based agent to perform 16,000 MCTS-iterations on 50 determinizations. This resulted in 320 MCTS-iterations per determinization. The information set key-state has been BS, HC, and UC.

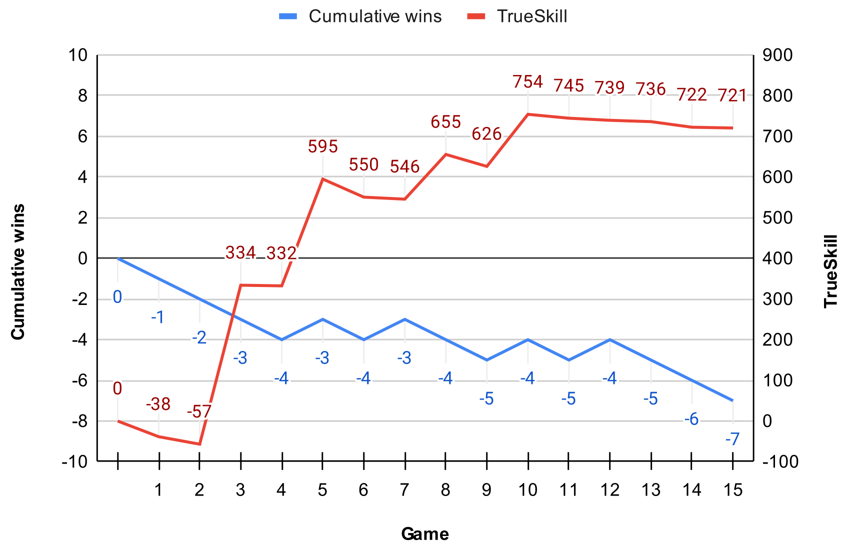
While exchanging the state between the local Just 4 Fun instance running a AZJ4F agent and the game on Yucata via their web interface, we observed overall, from our experience, solid policies in most of the situations. The only aspect that remained unclear from our observations was the behaviour near a win by points or loss for that matter. Despite that, the agent performed well even against skilled human players. Table 7.11 shows the human opponents that AlphaJust4Fun competed with.

The agent won 9 out of 15 games, i.e. the win rate was 60%. The performance in terms of wins/losses and TrueSkill-change is displayed in Figure 7.14b. The agent configuration is listed in Table C.11a.

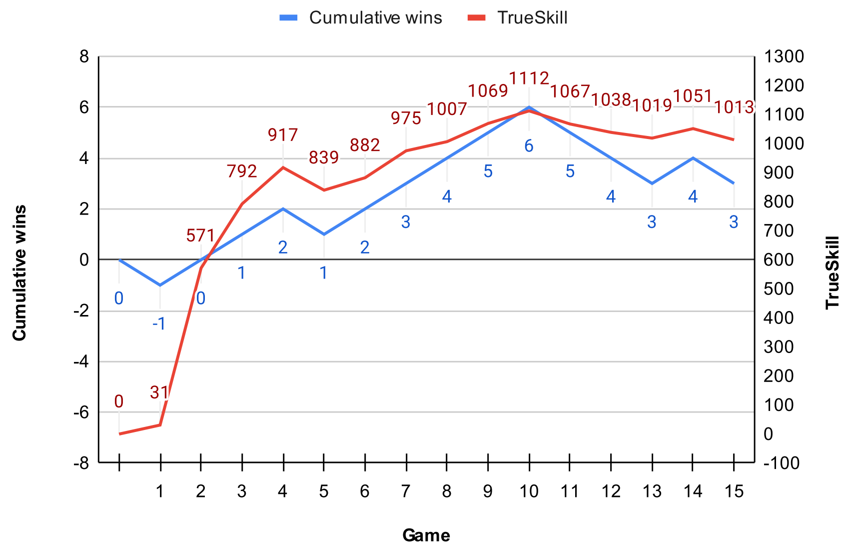
[3] https://www.yucata.de/en/User/AlphaJ4FZeroFNet (account required to view)
[4] https://www.yucata.de/en/User/AlphaJust4Fun (account required to view)

Human player		A	B	C	D	E	F	G
On Yucata	**Win rate**	47.63%	44.20%	**61.30**%	32.51%	**60.52**%	43.75%	**68.75**%
	#Games	359	**2663**	491	**3273**	**3181**	16	16
	TrueSkill	1,049	1,059	1,122	911	1,161	636	1,063
Vs. AZJ4F	**Wins**	1	1	1	1	1	1	4
	#Games	1	1	1	1	1	1	9

Table 7.11: The players on Yucata, AlphaJust4Fun (CFNet) played against. TrueSkill, win rate and number of games were recorded at the time of the last game AZJ4F played against the particular player.



(a) Game results and TrueSkill progression of `AlphaJust4FunZeroPlayer` (FNet, 8,000 MCTS-iterations, 700 determinizations) against human players resulted in 4 wins and 11 losses. This equates to a win rate of 26.67%. All games ended in a win win by pattern.



(b) Game results and TrueSkill progression of the AlphaJust4Fun agent (CFNet, 16,000 MCTS-iterations, 50 determinizations) against human players resulted in 9 wins and 6 losses. This equates to a win rate of 60%. All games ended in a win win by pattern.

Figure 7.14: Results when competing against human players on Yucata.

CHAPTER 8

Conclusions and Future Work

This last chapter concludes this thesis with a summary of the key findings. We highlight the importance, in relation to other research, and several limitations. At the end, we give a perspective on what future works might tackle.

8.1 Summary and Key Findings

AlphaZero combines Monte Carlo Tree Search with a deep neural network that estimates prior probabilities and state values. It has found great success in the deterministic perfect information games Go, shogi and chess. In this thesis, we described a modification of the AlphaZero framework, termed AlphaJust4Fun, to handle environments with imperfect information and non-determinism. As a benchmark problem, we used the game Just 4 Fun, which has a board with stones that is visible to all players and cards, of which each player only has knowledge of their own hand.

We replaced the Monte Carlo Tree Search algorithm that is being used in AlphaZero by the Single-Observer Information Set MCTS. Single-Observer Information Set MCTS performs search on concrete environment states, where the hidden information is determinized randomly. AlphaZero uses a ResNet-based neural network architecture which relies on input based on the visible geometric board state. It features a single trunk and two heads, one for value output and one for policy output. The cards in Just 4 Fun do not fit naturally as network inputs. For that reason, we propose a second network architecture, the CardFieldNet, which features two trunks and two heads. A residual network trunk takes the board state as an input, and a dense network trunk takes the visible part of the cards state as an input. A dense network combines both trunks into one common trunk. The output of this common trunk is the input for the value head and the policy head, which are both dense networks as well.

We evaluated AlphaJust4Fun in a set of game scenarios and in a benchmark. The main baseline in this benchmark is a MCTS algorithm, similar to the one used in AlphaZero, that is performing search in full knowledge of the hidden part of the game state.

The results indicate that AlphaJust4Fun successfully handles hidden information and non-determinism in Just 4 Fun. It outperforms the baseline and can also compete with experienced human players.

In our experiments, we investigate the ResNet-based architecture, using the board state and inputs that are derived from the cards and mapped to the board. The resulting agent was stable regarding the hyperparameters and outperformed the baseline.

A second agent, that uses the CardFieldNet architecture, significantly outperformed the baseline, reaching an even higher win rate than the agent with the ResNet-based architecture. We found the hyperparameter search for the CardFieldNet to be very challenging, as the training was very sensitive to various parameters.

Also in the artificial test scenarios, the CardFieldNet AlphaJust4Fun agent outperformed the baseline and the FieldNet AlphaJust4Fun agent.

Our experiments strongly suggest that the AlphaJust4Fun agents benefit from the DNN compared to a vanilla SO-ISMCTS agent. The network's policy alone performed at least similar to the baseline.

Similar to other research on ISMCTS, we find the determinization/MCTS-iteration balance to be less important, the range of reasonable ratios appears particularly wide for Just 4 Fun.

Combining the statistics of information set nodes worked well for Just 4 Fun. Increasing the information set size by omitting even the known cards state did also lead to reasonably good performance, which might be caused by the resulting greater exploration depth.

In summary, we were able to successfully apply our modification of the AlphaZero framework to the non-deterministic game Just 4 Fun. Replacing its Monte Carlo Tree Search algorithm with the Single-Observer Information Set MCTS turned out to handle the uncertainty in Just 4 Fun well.

8.2 Comparison with Previous Research

Not until recently, there were no general frameworks for non-deterministic environments with imperfect information that were able to achieve superhuman performance. Two prominent approaches, MuZero and Stochastic MuZero, have been very successful but add multiple neural networks, making training far more resource-intensive. Our approach may be slightly more computationally expensive during play, but requires a lot less resources for neural network training. With a default residual network, our approach only adds the number of determinizations to AlphaZero's hyperparameters, which we found to have a minor impact, at least in Just 4 Fun.

This makes AlphaJust4Fun almost as easy to configure as, e.g. AlphaZero, while still being able to handle a wider range of problems.

While we apply minor modifications to the MCTS algorithm that is used in AlphaZero, to turn it into a SO-ISMCTS, other researchers [56, 32, 5, 6, 7] report success by applying CFR-based tree search in this domain.

8.3 Limitations

Even though the residual network-based architecture was not nearly as good as the CardFieldNet in our benchmark, we think there might still be a margin for improvement, especially with more effort put into hyperparameter optimisation and addressing overfitting.

Surprisingly, the determinization/MCTS-iteration ratio had so little influence, with a total of 900 MCTS-iterations. We suspect that, with a much higher number of iterations, the number of determinizations might also have more impact on the performance.

The two-trunk network architecture is a valid approach for inputs of different types. However, the fact that it was such a challenge to find hyperparameters leading to reasonably good learning success makes it less appealing than architectures closer to ResNet.

Even our most capable agent, which also competed against human players, still shows some weaknesses when dealing with points-wins and max-field-wins.

AlphaJust4Fun significantly outperformed our baseline. However, we spent less time on tuning the baseline agent. Instead, we used sensible defaults for most other parameters and only increased the number of playouts. With more tuning, the baseline performance might still improve.

While the capabilities of our final agent were evident in the limited sample of games against human players, a substantially larger number of games would be necessary for a conclusive comparison.

8.4 Future Work

For future research, we want to make AlphaJust4Fun able to handle scenarios with more than two agents. Just 4 Fun is still a good benchmark problem, as it supports up to four players.

The cards-based inputs used in this work are still very basic. The four cards in hand could also be encoded as four binary sequences, each indicating the position of a card in the vector of all cards.

It would also be interesting to experiment with additional cards-based features. For example, features could express the general rarity of cards, the rarity relative to already used cards, or the probability of the opponent being in possession of certain cards. Further input features for the board-based trunk, such as the probability of reaching specific fields when replacing 1, 2, 3, or 4 cards, or the probability of the opponent reaching certain fields based on the unknown cards, could augment the neural network's capabilities.

While the field reachability (probability distribution) and the most easy to achieve patterns, were determined empirically based on a large amount of data, it would be interesting to calculate the exact conditional probabilities.

Another interesting characteristic of Just 4 Fun, that is worth further investigation, is the change of the branching factor over the duration of a game. The possible reshuffles during a game might have effects that could be exploited by further enhancements of the MCTS-algorithm.

The output of our neural network architecture was the fields on the board. However, since multiple card combinations can often reach the same field, another interesting approach would be to output the exact card combination from the player's hand. This could also be implemented as the output of another network head.

Another interesting approach, though not generally applicable, would be to implement heuristics for card selection. For example, this could involve prioritizing combinations with lower-value cards, or choosing the combination with the largest or smallest number of cards.

In all our evaluations, we reset the agent's search tree after every game. In future work, it would be interesting to investigate the impact of not resetting the tree, particularly how performance would scale, and how it would comparison to our baseline.

It would be interesting to implement the multiple-observer information set MCTS (MO-ISMCTS) [12], which uses a different search tree for each player and should reduce the problem of strategy fusion.

Given the success of CFR-based algorithms, it would also be interesting to replace MCTS with CFR.

APPENDIX A

Yucata.de & Just 4 Fun

Yucata.de [55] implements the game Just 4 Fun with slight modifications to the original rules. A digital copy of the original rules can be downloaded on spielen.de [27]. One modification is the fact that, when played with more than two players, the played cards are not shown. In two player-mode, the cards that have been used in the most recent action are displayed. Intuitively, with the game progressing, with every turn more information about the hidden stack of cards becomes available (e.g. information about the card that is necessary to complete a pattern, likely still being available or not). Conversely, the used cards not being visible, makes the game more difficult. Even more so with 4 players, the used cards not being visible can have a bigger impact.

Apart from the original field-value distribution, there is also an ordered-distribution A.1 (in row-wise ascending order) and a random distribution (the values of 1-36 are distributed in random order).

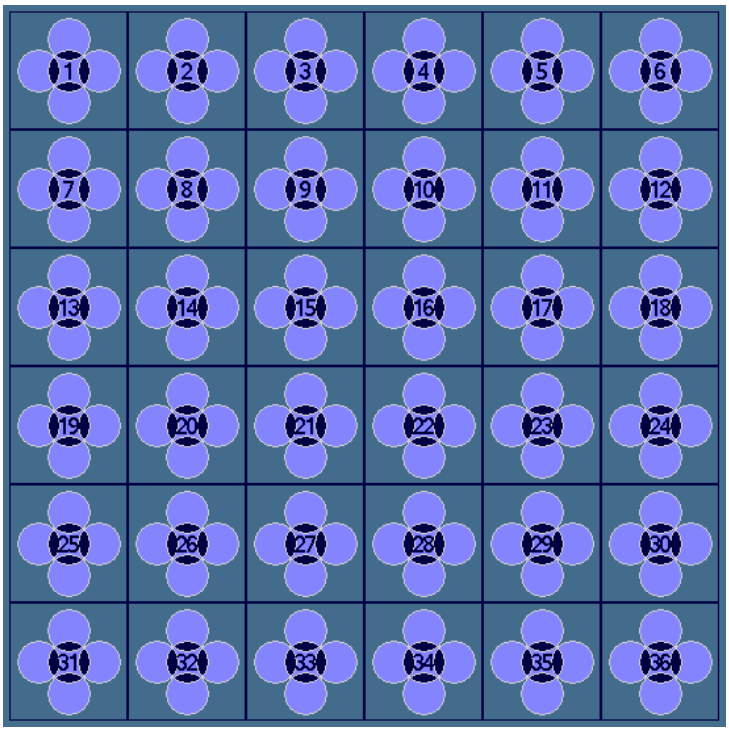


Figure A.1: Just 4 Fun board with the ordered field-value distribution [55].

1 26.73%	**2** 28.68%	**3** 31.64%	**4** 33.80%	**5** 36.90%	**6** 39.24%
7 42.50%	**8** 45.06%	**9** 48.45%	**10** 51.27%	**11** 54.83%	**12** 57.85%
13 46.71%	**14** 46.92%	**15** 47.74%	**16** 47.75%	**17** 48.33%	**18** 48.09%
19 48.39%	**20** 41.95%	**21** 40.54%	**22** 38.20%	**23** 36.18%	**24** 33.26%
25 30.62%	**26** 28.54%	**27** 26.70%	**28** 24.39%	**29** 22.35%	**30** 19.87%
31 17.63%	**32** 14.95%	**33** 13.53%	**34** 11.81%	**35** 10.55%	**36** 9.02%

Figure A.2: Approximate field (field value is in bold) reachability (probability in percent below the field value) calculated by sampling 10^6 times 4 cards from the deck at random, drawn on the game board with ordered field-value distribution as heat-map.

APPENDIX B

AlphaJust4Fun

Algorithm B.1: Detailed description of AlphaJust4Fun's MCTS algorithm

Input: s_0 - the a root node of a game subtree, composed of nodes s_w and arcs (s_x, a_y), where some player z is about to play and which corresponds to the root information set I_0^i of that subtree
n_{iter} - the number of MCTS-iterations
n_{det} - the number of determinizations
C - the exploration-constant function

for n_{iter} *iterations* **do**
  if *First iteration* ***OR*** *every* $\frac{n_{iter}}{n_{det}}$*-th iteration* **then**
    Select determinization d_j from $D(\mathrm{H}_0^i)$ at random
  end
  Start from the root node by assigning $s_k = s_0$
  repeat // Selection
    Descend the subtree by selecting arcs (s_k, a_l) with $a_l \in A^{d_j}(s_k)$, that are available from s_k under determinization d_j, and maximise the upper confidence bound $Q(s_k, a_l) + C(s_0) \cdot P(s_k, a_l) \cdot \sqrt{\frac{N_a(s_k,a_l)}{1+N(s_k,a_l)}}$, and updating s_k to the selected child node $s_k = s_l$
  until *an arcs (s_k, a_l) is reached, that leads to a node (corresponding to an information set), which is not in the tree yet* ***or until*** *arc (s_k, a_l) leads to a terminal node*
  if *arc (s_k, a_l) leads to a node, which is not in the tree yet* **then**
    Add the child node s_l to s_k's arc (s_k, a_l), that is corresponding to the information set $\mathrm{I}_{s_l}^i$
  end
  if s_l *is a terminal node under* d_j **then** // Simulation
    Initialise s_l's value using the terminal reward: $V(s_l) = \mathrm{r}^{d_j}(s_l)$
  else
    Initialise s_l and its arcs (s_l, a_m) with $a_m \in A(s_l)$ using the estimation of the current DNN checkpoint f_{θ_c}:
    $V(s_l) = v_l$ with $(v_l, \mathbf{p_l}) = f_{\theta_c}(s_l)$
    $P(s_l, a_m) = p_{l,a_m}$
    $W(s_l, a_m) = 0$
    $N_a(s_l, a_m) = \begin{cases} 1 & \text{for } (s_l, a_m) \in A^{d_j}(s_l) \\ 0 & \text{for } (s_l, a_m) \notin A^{d_j}(s_l) \end{cases}$
    $N(s_l, a_m) = 0$
  end
  for *each arc (s_k, a_l) visited during this iteration* **do** // Backpropagation
    Update (s_k, a_l)'s visit count $N(s_k, a_l)$ and total action value $W(s_k, a_l)$
    for *each sibling (s_k, a_m) with $(s_k, a_m) \in A^{d_j}(s_k)$, that was available for selection when (s_k, a_l) was selected, including itself* **do**
      Increment (s_k, a_m)'s availability count $N_\mathrm{a}(s_k, a_m)$
    end
  end
end

APPENDIX C

Experiment Setup

`RandomPlayer`	
Random seeds (120)	123, 124, . . . , 242

Table C.1: Randomness seeds used for the randomly acting agent `RandomPlayer` on the tests.

MCTS Parameters	
Reward discount (γ)	1
Exploration constant	1
MCTS-iterations per turn	900
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	Yes
Randomness seeds (60)	123, 124, . . . , 182

(a) Configuration of `CheatingMcts-Rollouts` agent used for the tests.

MCTS Parameters	
Determinizations per turn	4
Reward discount (γ)	1.0
Exploration constant	0.5
MCTS-iterations per turn	900
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.4
Reset search tree	After each game
Information set key-state	player's stones, opponent's stones, player's hand, used cards
Randomness seeds (60)	123, 124, . . . , 182

(b) `IsMctsRollouts` agent

Table C.2: MCTS configuration that was used in the benchmark against the other agents.

MCTS Parameters	
Reward discount (γ)	1
Exploration constant	1
MCTS-iterations per turn	8,000
Temperature Schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	After each game
Randomness seed	1,234

(a) Configuration of the cheating MCTS agent that was used to play against the human baseline.

MCTS & Benchmark Parameters	
Reward discount (γ)	1
Exploration constant	1
MCTS-iterations per turn	2, 50, 8,000
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	Yes
Randomness seed	1,234
Games	50

(b) Configuration of the cheating MCTS that was used for the benchmark against the `RandomPlayer`.

Benchmark Parameters	
Randomness seed	4,321
Games	50,000

(c) Configuration of the random against random benchmark.

Table C.3: Evaluation configurations of baseline benchmarks.

MCTS Parameters	
Reward discount (γ)	1.0
Exploration constant	1.0
MCTS-iterations per turn	900, 150
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	After each game

(a) `CheatingMctsRollouts` agent

MCTS Parameters	
Determinizations per turn	3
Reward discount (γ)	1.0
Exploration constant	0.5
MCTS-iterations per turn	900
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.4
Reset search tree	After each game
Information set key-state	player's stones, opponent's stones, player's hand, used cards

(b) `IsMctsRollouts` agent

Table C.4: MCTS configuration that was used in the benchmark against the other agents.

Self-play Parameters	
Determinizations per turn	20
Information set key-state	player's stones, opponent's stones, player's hand
Reward discount (γ)	0.95
Exploration constant	0.7
MCTS-iterations per turn	100
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Games per iter.	70
Fill batches	Yes
Reset search tree	After each iteration
Learning Parameters	
Position averaging	Yes
Sample weighing policy	Logarithmic
Sample merging policy	stones, hand (player), used cards
Adam learning rate	7×10^{-5}
L2 regularisation	0.0001
Non-validity penalty	0.7
Batch size	64
Maximum # batch updates per iter.	6250
Training Parameters	
Training iterations	80
Ternary game outcome	Yes
Linear replay buffer size schedule	Iter. 1: 10,000, Iter. 14: 20,000, Iter. 34: 60,000, Iter. 80: 120,000
Randomness seed	11232

(a) Configuration of reinforcement learning.

`NetworkOnly` Benchmark	
Games against `CheatingMcts Rollouts`	80
Games against `RandomPlayer`	800
MCTS Parameters (`CheatingMctsRollouts`)	
Reward discount (γ)	1.0
Exploration constant	1.0
MCTS-iterations per turn	50
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	After each game

(b) Configuration of the benchmark against the random and the cheating MCTS baseline.

Network Parameters	
Custom kernel init. (CKI only)	See Equation (7.1)
Custom kernels frozen (CKI only)	No
Trunk kernel size	3×3
Dropout	No
# trunk blocks	2
# trunk filters	16
# policy head filters	6
# value head filters	8
Batch norm. momentum	0.8
Input feature planes (full)	empty, stones (player), stones (opponent), minority (player), majority (player), secured (player), secured (opponent), field values, field availability, field probability, field reachability
Input feature planes (min)	empty, stones (player), stones (opponent), field values, field availability, field probability, field reachability

(c) Configuration of the neural networks.

Table C.5: Configuration of the experiments for the evaluation of the benefit of custom convolutional kernel initialization over random initialization.

Common `IsMctsRollouts` Parameters	
MCTS-iterations per turn	900 and 8,000
Determinizations per turn (900 MCTS-iterations per turn)	120
Determinizations per turn (8,000 MCTS-iterations per turn)	700
Reward discount (γ)	0.88
Exploration constant	1.0
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.4
Reset search tree	After each game

(a) The common parameters used for the `IsMctsRollouts` agent.

Benchmark Parameters	
Games against `CheatingMctsRollouts`	100
MCTS Parameters (`CheatingMctsRollouts`)	
MCTS-iterations per turn	900 and 8,000
Reward discount (γ)	1.0
Exploration constant	1.0
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	After each game
`IsMctsRollouts` (BS)	
Information set key state	player's stones, opponent's stones
Common `IsMctsRollouts` Parameters	See Table C.6a
`IsMctsRollouts` (BS and HC)	
Information set key state	player's stones, opponent's stones, player's hand
Common `IsMctsRollouts` Parameters	See Table C.6a
`IsMctsRollouts` (BS, HC, and UC)	
Information set key state	player's stones, opponent's stones, player's hand, used cards
Common `IsMctsRollouts` Parameters	See Table C.6a

(b) Benchmark settings for the agents to be tested and the baseline.

Table C.6: Configuration of the experiments for the evaluation of the effect of different information set key-states.

Common `IsMctsRollouts` Parameters	
MCTS-iterations per turn	150 and 900
Reward discount (γ)	1.0
Exploration constant	1.0
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	After each game
Information set key-state (BS)	player's stones, opponent's stones
Information set key-state (BS, HC, and UC)	player's stones, opponent's stones, player's hand, used cards
Randomness seed	14

(a) The common parameters used for the `IsMctsRollouts` agent.

Benchmark Parameters	
Games against `CheatingMctsRollouts` (900 MCTS-iterations per turn)	100
Games against `CheatingMctsRollouts` (150 MCTS-iterations per turn)	200
MCTS Parameters (`CheatingMctsRollouts`)	
MCTS-iterations per turn	150 and 900
Reward discount (γ)	1.0
Exploration constant	1.0
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	After each game
`IsMctsRollouts`	
Determinizations per turn	1, 2, 3, 4, 5, 6, 8, 10, 15, 20, 26, 32, 40, 50, 60, 70, 80, 90
Common `IsMctsRollouts` Parameters	See Table C.7a

(b) Agent configuration used for the benchmark.

Table C.7: Configuration of the experiments for the evaluation of the effect of different ratios of MCTS-iterations to determinizations.

Self-play Parameters	
Reward discount (γ)	0.88
MCTS-iterations per turn	100
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Games per training iteration	70
Fill batches	Yes
Reset search tree	After each training iteration
Learning Parameters	
Position averaging	Yes
Sample weighing policy	Logarithmic
Sample merging policy	stones, hand (player), used cards
L2 regularisation	0.0001
Non-validity penalty	0.7
Batch size	64
Maximum # batch updates per iter.	6,250
Training Parameters	
# iterations	70/71
Ternary game outcome	Yes
Linear replay buffer size schedule	Iter. 1: 10,000, Iter. 14: 20,000, Iter. 34: 60,000, Iter. 99: 90,000
Randomness seed	Iter. 0-22: 11,232, Iter. 23-70/71: 132

(a) The reinforcement learning configuration.

Common Network Parameters	
Trunk kernel size	3×3
Dropout	No
# trunk blocks	2
# trunk filters	16
# policy head filters	6
# value head filters	8
Batch norm. momentum	0.8
Input feature planes	empty, stones (player), stones (opponent), minority (player), majority (player), secured (player), secured (opponent), field values, field availability, field probability, field reachability
Imperfect Info. Specific Parameters	
Adam learning rate	Iter. 1-22: 0.00007, Iter. 23-199: 0.0006
Determinizations per turn	2
Exploration constant	0.7
Information set key-state	player's stones, opponent's stones, player's hand, used cards
Perfect Info. Specific Parameters	
Adam learning rate	Iter. 1-22: 0.0003, Iter. 23-199: 0.0006
Exploration constant	1
Information set key-state	player's stones, opponent's stones, player's hand, opponent's hand, stack cards, used cards

(b) The common neural network configuration and specific parameters.

Table C.8: Configuration of the FieldNet-based AlphaJust4Fun agents used to compare training with perfect information and with imperfect information.

Training Benchmark of `NetworkOnly`	
Games against `RandomPlayer`	600
Games against `CheatingMctsRollouts`	60
MCTS Parameters (`CheatingMctsRollouts`)	
Reward discount (γ)	1.0
Exploration constant	1.0
MCTS-iterations per turn	50
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Reset search tree	After each game

(a) Configuration of the baseline agents in the benchmark after each training iteration.

Table C.9: Configuration of the FieldNet-based AlphaJust4Fun agent benchmark, used to compare training with perfect information and with imperfect information.

Self-play Parameters	
Determinizations per turn	2
Information set key-state	player's stones, opponent's stones, player's hand, used cards
Reward discount schedule (γ)	Iter. 0-33: 0.93, Iter. 34-129: 0.89
Exploration constant	2
MCTS-iterations per turn	150
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	Iter. 0-4: 0.7, Iter. 5-21: 0.7, Iter. 22-33: 0.35, Iter. 34-63: 0.5, Iter. 64-129: 0.4
Games per training iteration	75
Fill batches	Yes
Reset search tree	After each training iteration

Learning Parameters	
Position averaging	Yes
Sample weighing policy	Logarithmic
Sample merging policy	stones, hand (player)
Adam learning rate	Iter. 0-4: 0.00005, Iter. 5-21: 0.0001, Iter. 22-33: 0.0005, Iter. 34-63: 0.0008, Iter. 64-88: 0.0005, Iter. 89-129: 0.0001
L2 regularisation	0.0003
Non-validity penalty	Iter. 0-4: 0.5, Iter. 5-129: 0.3
Batch size	Iter. 0-4: 64, Iter. 5-129: 128
Maximum # batch updates per iter.	Iter. 0-4: 4,000, Iter. 5-21: 6,250, Iter. 22-63: 12,500, Iter. 64-129: 6,250

Training Parameters	
Training iterations	129
Ternary game outcome	Yes
Linear replay buffer size schedule	Iter. 1: 10,000, Iter. 5: 20,000, Iter. 20: 40,000, Iter. 99: 80,000, Iter. 199: 90,000, Iter. 300: 170,000
Randomness seed	Iter. 0-5: 29, Iter. 6-22: 20, Iter. 23-34: 35, Iter. 35-64: 65, Iter. 65-129: 99

Network Parameters	
Custom kernel init.	-
Trunk kernel size	3×3
Dropout	20%
# trunk blocks	3
# trunk filters	16
# board trunk neurons	16
# cards trunk layers	2
# cards trunk neurons	32
Cards Trunk Batch norm. momentum	0.8
# common trunk layers	3
# common trunk neurons	48
Common Trunk Batch norm. momentum	0.98
# Policy head layers	3
# Policy head neurons	32
Policy head Batch norm. momentum	0.8
# Value head layers	4
# Value head neurons	16
Value head Batch norm. momentum	0.98
Board Trunk Batch norm. momentum	0.8
Board Input feature planes	empty, stones (player), stones (opponent), minority (player), majority (player), secured (player), secured (opponent), field values, field availability, field probability, field reachability
Cards Input feature vectors	player's hand, used cards

Table C.10: Configuration of the CardFieldNet-based AlphaJust4Fun agent used in tests, the benchmark and in play against humans.

MCTS Parameters	
Determinizations per turn	50
Reward discount (γ)	1
Exploration constant	0.5
MCTS-iterations per turn	16,000
Temperature Schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.6
Reset search tree	After each game
Information set key-state	player's stones, opponent's stones, player's hand, used cards
Randomness seed	1,234

(a) MCTS configuration used for play against humans.

MCTS Parameters	
Determinizations per turn	3
Reward discount (γ)	1
Exploration constant	1 for benchmarks, 0.5 for tests
MCTS-iterations per turn	900
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.4
Reset search tree	Yes
Information set key-state	player's stones, opponent's stones, player's hand, used cards
Randomness seed (benchmark)	1,234
Randomness seeds (tests; 60)	123, 124, ..., 182

(b) MCTS configuration used for the tests and the agent benchmark.

Table C.11: Evaluation configurations of the CardFieldNet-based AlphaJust4Fun agent used in tests, benchmarks and play against humans.

Self-play Parameters	
Determinizations per turn	20
Information set key-state	player's stones, opponent's stones, player's hand, used cards
Reward discount (γ)	0.88
Exploration constant	0.7
MCTS-iterations per turn	150
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.666667
Games per training iteration	75
Fill batches	Yes
Reset search tree	After each training iteration
Learning Parameters	
Position averaging	Yes
Sample weighing policy	Logarithmic
Sample merging policy	stones, hand (player), used cards
Adam learning rate	Iter. 1-22: 0.00007, Iter. 23-199: 0.0006, Iter. 200-300: 0.0001
L2 regularisation	0.0001
Non-validity penalty	0.7
Batch size	64
Maximum # batch updates per iter.	6,250
Training Parameters	
Training iterations	80
Ternary game outcome	Yes
Linear replay buffer size schedule	Iter. 1: 10,000, Iter. 14: 20,000, Iter. 34: 60,000, Iter. 99: 90,000, Iter. 199: 180,000, Iter. 300: 280,000
Randomness seed	Iter. 0-22: 11,232, Iter. 23-199: 132, Iter. 200-300: 13

(a) The reinforcement learning configuration.

Network Parameters	
Custom kernel init.	See Equation (7.1)
Custom kernels frozen	No
Trunk kernel size	3×3
Dropout	No
# trunk blocks	3
# trunk filters	16
# policy head filters	6
# value head filters	8
Batch norm. momentum	0.8
Input feature planes	empty, stones (player), stones (opponent), minority (player), majority (player), secured (player), secured (opponent), field values, field availability, field probability, field reachability

(b) The neural network configuration.

Table C.12: Configuration of the FieldNet-based AlphaJust4Fun agent used in the benchmark and in play against humans.

MCTS Parameters	
Determinizations per turn	700
Reward discount (γ)	0.95
Exploration constant	0.7
MCTS-iterations per turn	8,000
Linear temperature schedule	Turn 1: 0.9, Turn 10: 0.8, Turn 20: 0.6, Turn 34: 0.4, Turn 40 0.01
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.6
Reset search tree	After each game
Information set key-state	player's stones, opponent's stones, player's hand, used cards
Randomness seed	1,234

(a) MCTS configuration used for play against humans.

MCTS Parameters	
Determinizations per turn	3
Reward discount (γ)	1.0
Exploration constant	0.5
MCTS-iterations per turn	900
Temperature schedule	1 (constant)
Prior temp.	1
Dirichlet noise ϵ	0.25
Dirichlet noise α	0.4
Reset search tree	After each game
Information set key-state	player's stones, opponent's stones, player's hand, used cards
Randomness seed (benchmark)	1,234
Randomness seeds (tests; 60)	123, 124, . . . , 182

(b) MCTS configuration used for the tests and the agent benchmark.

Table C.13: Evaluation configurations of the FieldNet-based AlphaJust4Fun agent used in tests, benchmarks and play against humans.

List of Figures

List of Tables

List of Algorithms

Glossary

MCTS-iteration *See* playout. Pages: 28, 29, 35, 36, 38, 43, 44, 47, 64, 73, 74, 75, 79, 80, 82, 83, 87, 91, 94, 96, 98, 100, 102, 103, 105, 106, 107, 108, 110, 112, 114, 116, 117, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 139, 140, 141, 142, 145, 146

full game-state Contains the board-state and the full cards-state. Pages: 42, 73, 146

public cards-state Contains the pile of used cards, visible to all players. Pages: 42, 50, 53, 82, 89, 145

board-state Contains the number of stones of each player on each field. Pages: 42, 46, 50, 82, 145

BS This combines the statistics of all games states that share the same board-state. It is the smallest information set key-states and combines all permutations of the full cards-state. Pages: 82, 83, 85, 87, 97, 107, 128, 129, 139

BS, HC, and UC This effectively combines the statistics of all game states, collected during the MCTS-iterations, that share the same board-state, player-cards-state, and public cards-state. This is the largest information set key-states. It combines the least number of (full) game states. Pages: 82, 83, 85, 86, 87, 88, 93, 94, 95, 96, 97, 98, 99, 100, 103, 105, 106, 107, 108, 112, 128, 129, 139, 140, 141, 142

determinization A determinization of a state (determinization for short) in context of imperfect information games is the full game state, with the unknown portion of a state being sampled from some distribution. Pages: 19, 36, 37, 38, 39, 41, 43, 44, 47, 64, 74, 75, 77, 82, 83, 84, 85, 86, 87, 88, 93, 94, 96, 98, 99, 100, 103, 105, 106, 108, 112, 114, 116, 117, 124, 125, 126, 127, 128, 129, 130, 132, 133, 134, 135, 139, 140, 141, 142

end in draw The extremely rare case in which all players put all their stones on the same fields and thus did not hold a majority on any field. Page: 24

full cards-state Contains the player's own cards, the pile of used cards, the (hidden) stack of cards and the (hidden) opponent's cards. Pages: 42, 82, 145

information set key-state The part of the full game-state that is common to all possible full game-states within an Information Set. Pages: 42, 73, 77, 82, 83, 84, 85, 87, 93, 94, 97, 99, 107, 108, 112, 125, 126, 127, 128, 129, 130, 132, 133, 134, 135, 139, 140, 142, 145

non-locality During search, only a particular subtree is used to evaluate the payoff of a strategy. This subtree, however, might not be relevant, as the subtree from the actual state is in a different part of the game tree and has a different set of payoffs. Page: 37

player-cards-state Contains the player's own cards which are only visible to the player they belong to. Pages: 42, 82, 145

playout In the literature, the terms simulation, playout and rollout are often used interchangeably. In this thesis we will use the term **MCTS-iteration** for a single iteration of a MCTS algorithm as described in Subsection 2.1.2, i.e. the combination of selection, expansion, simulation and backpropagation. The value estimation in the simulation phase, can be an **evaluation** in the case of AlphaZero, where the neural network estimates the value, or a **playout** in the case of vanilla MCTS, where the games are continued according to some policy until they terminate. Pages: 2, 16, 28, 39, 64, 65, 77, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 91, 93, 94, 99, 110, 117, 139, 145, 146

rollout *See* playout. Pages: 13, 16

strategy fusion An agent erroneously assumes, it can decide on the right strategy in different states within an information set. This leads to incorrect decisions, as the states within an information set cannot be distinguished from each other. Pages: 37, 45, 118

win by max field None of the players were able to construct a pattern and all players even had the same number of points. However, one player held the majority on a field with a higher value than all the fields of the other players. Pages: 24, 66

win by points None of the players were able to construct a pattern, but either player had more points than the other players and won the game that way. Pages: 24, 66, 83, 112

win by pattern Either of the players won by constructing a pattern. Pages: 24, 65, 66, 82, 83, 102, 114

board A 6×6 grid of 36 fields. The field-value distribution is predefined. The board, as generated by the Yucata.de-implementation, is depicted in Figure 3.1. They also provide an ordered (Yucata.de & Just 4 Fun, Figure A.1) and a random field-value distribution. Pages: 21, 24, 33, 47, 54, 57, 59, 66, 120, 140, 147

card A card is represented by a value between 1 and 19. An example is depicted in Figure 3.2. Pages: 21, 22, 24, 25, 41, 42, 46, 47, 59, 64, 67, 146, 147, 148

deck The set of all cards. The cards with values 1-12 are contained 4 times each and the ones with values 13-19 once. Pages: 21, 24, 25, 42, 49, 57, 58, 138

field A field is represented by a position – the x and y coordinates on the board and a board-wide unique value. The value can be any integer between 1 and 36. The fields contain the number of stones of all players. Pages: 21, 22, 23, 24, 25, 27, 42, 46, 49, 50, 52, 57, 58, 59, 64, 65, 66, 73, 119, 120, 138, 140, 145, 146, 147, 148

hand A set of 4 cards in a player's possession. Pages: 22, 24, 25, 41, 52, 58, 59, 60, 73, 74, 94, 138, 147

majority A player has majority on a field if they have at least one stone more than every other player on this field. See also Figure 3.3. Pages: 22, 23, 24, 145, 146, 147

pattern A pattern in the context of J4F is the alignment of stones on the board. A player wins, if they have the majority on at least 4 fields aligned in a horizontal, vertical or diagonal line. An example for a winning pattern in a diagonal is depicted in Figure 3.3. Pages: 22, 24, 25, 46, 146

redraw action Putting down all four cards and drawing four new ones from the stack of cards, without placing a stone. This action has to be played and can only be played, in case no regular action is possible according to the rules (i.e. on every field reachable with the particular hand, there is a player who has a majority). Page: 22

regular action Putting down between 1 and 4 cards of the player's hand and putting a stone on the field with the number equal to the sum of the played card values. An example for a regular action is depicted in Figure 3.2. Page: 21

stack of cards The stack of cards a player is drawing from to refilling their hand after an action. Pages: 22, 41, 42, 82, 94, 146, 147, 148

stone A unit to be placed on (linked to) a field. Each player starts with 20 stones. Pages: 21, 22, 23, 24, 42, 46, 57, 58, 64, 65, 66, 145, 147

used cards Cards that have been played and are not available until the stack of cards is empty. In that case, the used cards are shuffled and become the new stack of cards. Pages: 22, 41, 42, 146

Acronyms

$\overline{\mathbf{Q}}_{\mathbf{mcts,scaled}}$ Mean of the scaled action values. Pages: 69, 72

$\overline{\mathbf{Q}}_{\mathbf{net,scaled}}$ Mean of the scaled action value estimations of the DNN. Pages: 68, 69, 72, 96

$\overline{\mathbf{UCT}}_{\mathbf{scaled}}$ Mean of the scaled UCT values. Pages: 69, 72, 80, 81, 96, 98, 102, 103

$\mathbf{cP}_{\mathbf{net}}^{\mathbf{pre}}$ Combined policy estimation of the DNN. Pages: 69, 72

$\mathbf{cP}_{\mathbf{mcts}}$ Combined MCTS policy. Pages: 69, 72, 80, 81

$\mathbf{cP}_{\mathbf{net}}$ Combined policy estimation of the DNN after masking. Pages: 69, 72, 96, 98, 103

CE cross-entropy. Pages: 66, 67, 69, 70, 71, 74, 78, 79, 80, 93, 94, 95, 96, 97, 99, 100, 101, 139, 141, 142

$V_{\mathbf{net}}$ Value estimation of the DNN. Pages: 68, 72, 96, 97, 104, 105, 140

AG AlphaGo. Pages: 2, 16, 28

AGZ AlphaGo Zero. Pages: 2, 16, 28, 65

AI artificial intelligence. Pages: xiii, 1, 2, 4, 5, 7, 15, 17

AZ AlphaZero. Pages: xi, xii, xiii, 2, 3, 4, 5, 15, 16, 17, 18, 19, 21, 28, 29, 31, 35, 41, 43, 45, 46, 47, 49, 50, 63, 64, 69, 73, 115, 116, 117, 146

AZ.jl AlphaZero.jl. Pages: 63, 72, 73

AZJ4F AlphaJust4Fun. Pages: xi, xii, xiii, xvi, 5, 19, 21, 41, 43, 45, 46, 47, 64, 65, 73, 77, 79, 93, 94, 95, 96, 97, 98, 99, 100, 102, 103, 105, 107, 110, 111, 112, 113, 114, 115, 116, 117, 123, 124, 130, 131, 132, 133, 134, 135, 139, 140, 141, 142, 143

AZJ4F.jl AlphaZeroJust4Fun.jl. Pages: 64, 72, 73

CFNet CardFieldNet. Pages: xv, 49, 53, 54, 55, 56, 57, 58, 59, 60, 61, 64, 65, 75, 99, 100, 102, 103, 104, 105, 106, 107, 110, 111, 112, 113, 114, 115, 116, 117, 132, 133, 138, 139, 140, 142

CFR Counterfactual Regret Minimization. Pages: 3, 14, 15, 17, 19, 117, 118

chess chess. Pages: xiii, 1, 2, 8, 9, 15, 16, 18, 28, 29, 31, 34, 35, 46, 112, 115

CKI custom kernel initialisation. Pages: 91, 92, 93, 141

DNN deep neural network. Pages: xi, xiii, 2, 4, 15, 16, 28, 30, 32, 33, 38, 43, 45, 46, 64, 65, 83, 88, 93, 95, 100, 107, 115, 116, 124, 149

Dou dizhu Dou dizhu. Pages: 3, 43

FNet FieldNet. Pages: xv, 49, 50, 51, 52, 53, 56, 57, 58, 59, 60, 61, 64, 65, 88, 89, 91, 94, 95, 96, 97, 98, 99, 102, 104, 105, 106, 107, 110, 112, 114, 116, 130, 131, 134, 135, 138, 139, 141, 142

HULHE heads-up limit Texas hold'em poker. Page: 14

HUNL heads-up no-limit Texas hold'em poker. Pages: 3, 17

ISMCTS Information Set Monte Carlo Tree Search. Pages: 5, 10, 14, 21, 37, 38, 41, 116

J4F Just 4 Fun. Pages: xi, xiii, xvi, 3, 4, 5, 9, 11, 15, 16, 21, 37, 41, 46, 49, 55, 57, 64, 65, 67, 86, 107, 112, 115, 116, 117, 118, 119, 120, 147

J4F.jl Just4Fun.jl. Page: 63

LB Libratus. Pages: 3, 17

LOTR:C Lord of the Rings: The Confrontation. Page: 43

m,n,k m,n,k. Page: 11

MCCFR Monte Carlo Counterfactual Regret Minimisation. Pages: 3, 14

MCTS Monte Carlo Tree Search. Pages: xi, xiii, 2, 4, 10, 12, 14, 15, 16, 17, 18, 28, 29, 31, 35, 37, 43, 47, 64, 65, 69, 70, 73, 77, 79, 80, 82, 83, 91, 94, 95, 96, 97, 99, 100, 107, 108, 110, 111, 115, 116, 117, 118, 124, 125, 126, 127, 128, 129, 131, 133, 135, 140, 141, 142, 143, 145, 146, 149

MZ MuZero. Pages: xv, 3, 17, 18, 19, 116

PB Pluribus. Pages: 3, 17

PUCB Predictor + UCB. Pages: 14, 28

ReBeL Recursive Belief-based Learning. Pages: xv, 17, 19

ResNet residual network. Pages: 15, 33, 115, 116, 117

RL reinforcement learning. Pages: xi, xiii, 2, 3, 15, 16, 17, 18, 28, 31, 127, 130, 134

shogi shogi. Pages: xi, xiii, 2, 16, 18, 34, 46, 115

SMZ Stochastic MuZero. Pages: xv, 3, 18, 19, 116

SO-ISMCTS Single-Observer Information Set MCTS. Pages: xi, xiii, 19, 38, 41, 43, 64, 69, 77, 82, 84, 86, 107, 108, 111, 115, 116, 117, 139, 140

SP self-play. Pages: xi, xiii, 2, 3, 14, 15, 16, 17, 28, 29, 31, 32, 36, 38, 45, 47, 73, 74, 87, 88, 90, 102, 127, 130, 132, 134

TS TrueSkill. Pages: 15, 16, 65, 112, 113, 114, 142

TTT Tic-Tac-Toe. Pages: 1, 8, 11, 64

UCB1 Upper confidence bound 1 policy by Auer et al.[2]. Pages: 12, 13, 38

UCT UCB1 applied to trees. Pages: 13, 37, 38, 69, 96, 102, 149

Yucata Yucata. Pages: 21, 65, 73, 93, 112, 113, 114, 140, 142

Bibliography

[1] Ioannis Antonoglou et al. "Planning in Stochastic Environments with a Learned Model". In: *Proceedings of the 10th International Conference on Learning Representations (ICLR)*. Virtual, 2022. URL: https://openreview.net/forum?id=X6D9bAHhBQ1.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.2 (2002), pp. 235–256. DOI: 10.1023/A:1013689704352.

[3] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671.

[4] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. "Lower Bounding Klondike Solitaire with Monte-Carlo Planning". In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*. Thessaloniki, Greece: AAAI Press, 2009, pp. 26–33. DOI: 10.1609/icaps.v19i1.13363.

[5] Michael Bowling et al. "Heads-up limit hold'em poker is solved". In: *Science* 347.6218 (2015), pp. 145–149. DOI: 10.1126/science.1259433.

[6] Noam Brown and Tuomas Sandholm. "Superhuman AI for heads-up no-limit poker: Libratus beats top professionals". In: *Science* 359.6374 (2018), pp. 418–424. DOI: 10.1126/science.aao1733.

[7] Noam Brown and Tuomas Sandholm. "Superhuman AI for multiplayer poker". In: *Science* 365.6456 (2019), pp. 885–890. DOI: 10.1126/science.aay2400.

[8] Noam Brown et al. "Combining Deep Reinforcement Learning and Search for Imperfect-Information Games". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS)*. Vancouver, Canada: Curran Associates Inc., 2020, pp. 17057–17069.

[9] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[10] S. Barry Cooper and Jan Van Leeuwen. "Digital Computers Applied to Games". In: *Alan Turing: His Work and Impact*. Elsevier, 2013, pp. 623–650. DOI: 10.1016/B978-0-12-386980-7.50024-1.

[11] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Proceedings of the 5th International Conference on Computers and Games (CG)*. Turin, Italy: Springer Berlin Heidelberg, 2007, pp. 72–83. DOI: 10.1007/978-3-540-75538-8_7.

[12] Peter I. Cowling, Edward J. Powley, and Daniel Whitehouse. "Information Set Monte Carlo Tree Search". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.2 (2012), pp. 120–143. DOI: 10.1109/TCIAIG.2012.2200894.

[13] Peter I. Cowling, Colin D. Ward, and Edward J. Powley. "Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.4 (2012), pp. 241–257. DOI: 10.1109/TCIAIG.2012.2204883.

[14] Arpad E. Elo. *The rating of chessplayers, past and present*. 2nd Edition. Arco Pub, 1978.

[15] Alhussein Fawzi et al. "Discovering faster matrix multiplication algorithms with reinforcement learning". In: *Nature* 610.7930 (2022), pp. 47–53. DOI: 10.1038/s41586-022-05172-4.

[16] Ian Frank and David Basin. "Search in games with incomplete information: a case study using Bridge card play". In: *Artificial Intelligence* 100.1 (1998), pp. 87–123. DOI: 10.1016/S0004-3702(97)00082-9.

[17] Mario Gastegger. *AlphaJust4Fun, a modification of AlphaZero (AlphaZero.jl), that replaces the MCTS algorithm by SO-ISMCTS to solve board game Just 4 Fun (Just4Fun.jl)*. https://gitlab.com/gwario/AlphaZeroJust4Fun.jl. 2024.

[18] Mario Gastegger. *Just 4 Fun implementation for AlphaZero.jl and AlphaZero-Just4Fun.jl*. https://gitlab.com/gwario/Just4Fun.jl. 2024.

[19] Matthew L. Ginsberg. "GIB: Imperfect Information in a Computationally Challenging Game". In: *Journal of Artificial Intelligence Research* 14 (2001), pp. 303–358. DOI: 10.1613/jair.820.

[20] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 9. Proceedings of Machine Learning Research. Sardinia, Italy: PMLR, 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[21] Jürgen P. Grunau. *Jürgen P. Grunau - Board Game Designer*. BoardGameGeek. URL: https://boardgamegeek.com/boardgamedesigner/272/jurgen-p-grunau (visited on 06/20/2023).

[22] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, USA: IEEE, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[23] Ralf Herbrich, Tom Minka, and Thore Graepel. "TrueSkill™: A Bayesian Skill Rating System". In: *Proceedings of the 19h International Conference on Neural Information Processing Systems (NIPS)*. Vancouver, Canada: MIT Press, 2006, pp. 569–576. URL: https://papers.nips.cc/paper_files/paper/2006/hash/f44ee263952e65b3610b8ba51229d1f9-Abstract.html.

[24] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012. DOI: 10.48550/arXiv.1207.0580. arXiv: 1207.0580 [cs.NE].

[25] *Julia Documentation · The Julia Language*. URL: https://docs.julialang.org/en/v1.9/ (visited on 09/23/2023).

[26] *Just 4 Fun - Overview*. BoardGameGeek. URL: https://boardgamegeek.com/boardgame/17534/just4fun (visited on 06/20/2023).

[27] *Just 4 Fun - Rules (scan as PDF) on spielen.de*. spielen.de. URL: https://gesellschaftsspiele.spielen.de/alle-brettspiele/just-4-fun/ (visited on 05/22/2024).

[28] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980 [cs.LG].

[29] Günter Klambauer et al. *Self-Normalizing Neural Networks*. 2017. DOI: 10.48550/arXiv.1706.02515. arXiv: 1706.02515 [cs.LG].

[30] Donald E. Knuth and Ronald W. Moore. "An analysis of alpha-beta pruning". In: *Artificial Intelligence* 6.4 (1975), pp. 293–326. DOI: 10.1016/0004-3702(75)90019-3.

[31] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Proceedings of the 17th European Conference on Machine Learning (ECML)*. Berlin, Germany: Springer Berlin Heidelberg, 2006, pp. 282–293. DOI: 10.1007/11871842_29.

[32] Marc Lanctot et al. "Monte Carlo Sampling for Regret Minimization in Extensive Games". In: *Proceedings of the 23rd International Conference on Neural Information Processing Systems (NIPS)*. Vol. 22. Vancouver, Canada: Curran Associates, Inc., 2009, pp. 1078–1086. URL: https://papers.nips.cc/paper_files/paper/2009/hash/00411460f7c92d2124a67ea0f4cb5f85-Abstract.html.

[33] Jonathan Laurent. *AlphaZero.jl: A generic, simple and fast AlphaZero implementation*. https://github.com/jonathan-laurent/AlphaZero.jl. 2021.

[34] Min Lin, Qiang Chen, and Shuicheng Yan. *Network In Network*. 2014. arXiv: 1312.4400 [cs.NE].

[35] Jeffrey Long et al. "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search". In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. Vol. 24. 1. Atlanta, USA: AAAI Press, 2010, pp. 134–140. DOI: 10.1609/aaai.v24i1.7562.

[36] Daniel J. Mankowitz et al. "Faster sorting algorithms discovered using deep reinforcement learning". In: *Nature* 618.7964 (2023), pp. 257–263. DOI: 10.1038/s41586-023-06004-9.

[37] Matej Moravčík et al. "DeepStack: Expert-level artificial intelligence in heads-up no-limit poker". In: *Science* 356.6337 (2017), pp. 508–513. DOI: 10.1126/science.aam6960.

[38] Martin J. Osborne. *An Introduction to Game Theory*. Oxford University Press, 2009.

[39] Martin J. Osborne and Ariel Rubinstein. *A course in game theory*. MIT Press, 1994.

[40] Aske Plaat. *Learning to Play: Reinforcement Learning and Games*. Springer Cham, 2020. DOI: 10.1007/978-3-030-59238-7.

[41] Christopher D. Rosin. "Multi-armed bandits with episode context". In: *Annals of Mathematics and Artificial Intelligence* 61.3 (2011), pp. 203–230. DOI: 10.1007/s10472-011-9258-6.

[42] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. 3rd Global Edition. Pearson Education Limited, 2016.

[43] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4.

[44] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404.

[45] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[46] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (2017), pp. 354–359. DOI: 10.1038/nature24270.

[47] David Silver et al. "Supplementary Materials for 'A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play'". In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404.

[48] Maciej Świechowski et al. "Monte Carlo Tree Search: a review of recent modifications and applications". In: *Artificial Intelligence Review* 56.3 (2023), pp. 2497–2562. DOI: 10.1007/s10462-022-10228-y.

[49] Alan Turing. *AMT/K/1/77*. 1948. URL: https://turingarchive.kings.cam.ac.uk/material-given-kings-college-cambridge-1960-amtk/amt-k-1-77 (visited on 06/16/2023).

[50] J. W. H. M. Uiterwijk and H. J. van den Herik. "The advantage of the initiative". In: *Information Sciences* 122.1 (2000), pp. 43–58. DOI: 10.1016/s0020-0255(99)00095-x.

[51] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (2019), pp. 350–354. DOI: 10.1038/s41586-019-1724-z.

[52] Bing Xu et al. *Empirical Evaluation of Rectified Activations in Convolutional Network*. 2015. DOI: 10.48550/arXiv.1505.00853. arXiv: 1505.00853 [cs.LG].

[53] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-63519-4.

[54] *Yucata - Regeln für das Spiel 'Just 4 Fun'*. URL: https://www.yucata.de/de/Rules/Just4Fun (visited on 02/03/2024).

[55] *Yucata - Rules for the game 'Just 4 Fun'*. URL: https://www.yucata.de/en/Rules/Just4Fun (visited on 02/03/2024).

[56] Martin Zinkevich et al. "Regret Minimization in Games with Incomplete Information". In: *Proceedings of the 20th International Conference on Neural Information Processing Systems (NIPS)*. Vol. 20. Vancouver, Canada: Curran Associates, Inc., 2007, pp. 1729–1736. URL: https://papers.nips.cc/paper_files/paper/2007/hash/08d98638c6fcd194a4b1e6992063e944-Abstract.html.