Applying AlphaZero Algorithm to an Imperfect Information Card Game

Computer Science
Master's Degree Programme in Computer Science
Department of Computer Science, Faculty of Technology
Master of Science Thesis

Author:
Sami Krappe

July 2026
Turku

Master's thesis

Subject: Computer Science
Author: Sami Krappe
Title: Applying AlphaZero Algorithm to an Imperfect Information Card Game
Supervisors: Professor Tapio Pahikkala, Associate Professor Antti Airola
Number of pages: 56 pages
Date: July 2026

Reinforcement learning is a machine learning paradigm where the learning system does not require labelled training data, it learns by experimentation and environment feedback. Google DeepMind's AlphaGo demonstrated that reinforcement learning can beat a human champion in Go. It used recorded human expert games for supervised learning. DeepMind's AlphaZero was trained tabula rasa, without human knowledge. The method used self-play with Monte Carlo Tree Search to improve the suggestions from network. AlphaZero's differentiating property compared to AlphaGo is bootstrapping from games played without domain understanding. This thesis studies the possibility to apply AlphaZero-inspired approach to learn to play games of imperfect information. An important part of the approach is to evaluate model goodness without human intelligence. In this work I created an evaluation system using classic minimax, fixed strategy baselines and games against older versions of the model. Results were promising and surprisingly clear. Firstly, the system was shown to learn and to beat trivial strategies and to beat older versions of the model. Secondly, it was shown and measured how much the imperfect information affected the learning potential. It was shown that 73% learning from the absolute maximum was reached and only 7% of the learning potential was left undone. The rest 20% is considered to be behind so called "information barrier", caused by the imperfect information. Learning that 20% would require more advanced methods and those are discussed in the thesis. Thirdly, a methodological approach for decomposition of learning progress was created. Fourth, the created learning and evaluation system is adaptable to other games or to measuring the effect of different amount of hidden information. The covered theoretical background includes reinforcement learning and AlphaZero, probability-related search methods, Markov Decision Process and exploration-exploitation trade-off. Further study, applications and approach limitations were considered, showing interesting directions for future work.

Key words: AlphaZero, Reinforcement Learning, Monte Carlo Tree Search, Imperfect Information, Self-Play

Pro gradu -tutkielma

Pääaine: Tietojenkäsittelytieteet
Tekijä: Sami Krappe
Otsikko: AlphaZero -algoritmin soveltaminen epätäydellisen tiedon korttipeliin
Ohjaajat: Professori Tapio Pahikkala, apulaisprofessori Antti Airola
Sivumäärä: 56 sivua
Päiväys: Heinäkuu 2026

Vahvistusoppiminen on koneoppimisen paradigma, jossa oppiva järjestelmä ei tarvitse leimattua opetusdataa, vaan se oppii kokeilemalla ja ympäristöltä saatavan palautteen avulla. Google DeepMindin AlphaGo osoitti, että vahvistusoppivalla järjestelmällä voidaan voittaa mestaritason ihmispelaaja Go-pelissä. AlphaGo käytti tallennettuja ihmisasiantuntijoiden pelejä ohjattuun oppimiseen. DeepMindin AlphaZero koulutettiin tabula rasa -periaatteella, ilman ihmisen tuottamaa tietoa. Menetelmä käytti pelejä itseään vastaan yhdessä Monte Carlo -puuhaun kanssa parantaakseen neuroverkon ennusteita. AlphaZeron ero verrattuna AlphaGo:hon on ”bootstrappaus” eli oppiminen peleistä, joissa ei vielä ole minkäänlaista aihepiirin ymmärrystä. Tämä opinnäytetyö tutkii mahdollisuutta soveltaa AlphaZero -henkistä lähestymistapaa pelien oppimiseen tilanteissa, joissa pelitila ei ole täysin tiedossa, eli epätäydellisen informaation peleissä. Tärkeä osa kokeilun onnistumisessa on mallin hyvyyden arviointi ilman ihmisälyä. Tässä työssä loin arviointijärjestelmän, joka hyödyntää klassista minimax-hakua, kiinteitä vertailustrategioita sekä pelejä mallin vanhempia versioita vastaan. Tulokset olivat lupaavia ja yllättävän selkeitä. Ensinnäkin järjestelmän osoitettiin oppivan ja voittavan triviaalit strategiat sekä selvästi voittavan mallin vanhemmat versiot. Toiseksi mitattiin, kuinka paljon epätäydellinen informaatio vaikutti oppimispotentiaaliin. Osoitettiin, että 73 % teoreettisesta maksimioppimisesta saavutettiin ja vain 7 % oppimispotentiaalista jäi käyttämättä. Loput 20 % katsotaan johtuvan niin kutsutusta informaatiomuurista, jonka epätäydellinen informaatio aiheuttaa. Tämän 20 %:n oppiminen vaatisi kehittyneempiä menetelmiä ja niitä käsitellään työn keskusteluosuudessa. Kolmanneksi luotiin havainnollinen oppimisen komponenttien seuranta. Toteutettu kokeellinen järjestelmä ja arviointimenetelmä on sovellettavissa muihin peleihin tai kun piilotetun informaation määrän vaikutusta halutaan tarkastella. Työn teoreettinen tausta kattaa vahvistusoppimisen ja AlphaZeron, todennäköisyyspohjaiset hakumenetelmät, Markovin päätösprosessin sekä ”exploration-exploitation” -tasapainon. Jatkotutkimusta, sovelluksia ja valitun lähestymistavan rajoituksia pohdittiin ja näiden perusteella löytyi useita kiinnostavia suuntia jatkotyölle.

Asiasanat: AlphaZero, Vahvistusoppiminen, Monte Carlo -puuhaku, epätäydellinen informaatio, itseään vastaan pelaaminen

Table of contents

1 Introduction

1.1 Motivation and introduction

In this thesis work, I wanted to examine and study the possibility of utilizing machine learning in games, where the player does not know everything about the game state. These kinds of games are called ”games of imperfect information” [1, p. 609].

My interest in this topic relates also to the fact that this kind of experimental study setup has similarities to real-world decision-making situations: There is no prior knowledge, the environment parameters or dependencies are not fully understood, but one should still decide what to do. In real-world decision making there is also an uncertainty factor: sometimes the same thing works, sometimes it doesn’t, even if the environment seems to be the same. In real world one cannot necessarily do a large number of tests, what happens by selecting A or B a thousand times but some kind of simulations can be used. Anyhow, the decision maker uses some kind of internal model to evaluate the state, and this could be at least in some cases be encoded in an algorithm.

Especially, I wanted to find out, how can machine learning work, when there is no “human understanding” involved, and the learning system itself has no domain specific heuristics of the game. Heuristics here means informed reasoning and understanding of how to proceed in any given game state. Heuristic can be defined as “any approach to problem solving that employs a practical method that is not fully optimized, perfected, or rationalized, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation” [2]. Rules of the game are known but they are not used to infer playing strategy.

The machine learning paradigm in this kind of setup where there is no training data available, is called Reinforcement Learning (RL). This paradigm in my own opinion is the most interesting area in machine learning. It can, to some extent, be compared with human and animal ways of learning. Playing games in computer science is very often seen as a search problem, searching for the possible actions and outcomes, then selecting somehow the preferred one. Plain “Trial and error” -method from reinforcement learning may be very slow and ineffective in a game search problem, so I wanted to apply the ideas from Google’s DeepMind group and their AlphaZero [3] approach, where a reinforcement learning search algorithm is used together with a supervised deep learning model. This deep learning model is trained with a large number of games, where the RL-system has played against itself, with no

initial understanding of the game. The experimental self-play game data is given as input to a supervised deep learning model. All the rules and game outcomes come from external referee, saying for example that “this move cannot be done”, or “player A wins now”.

1.2 Games in computer science

Solving games computationally has traditionally been framed as a search problem. In a given a game state, computer player constructs a search tree. In the tree nodes represent states and edges represent moves. A move is what is taking the game from a state to another. A heuristic evaluator scores leaf nodes, and the best move is selected by propagating scores back up the tree. In a two-player zero-sum game, where one player's gain is exactly the other's loss, this is formalized as the Minimax algorithm, which is proven to find the optimal move [1, p. 194]. In the algorithm one player maximizes the score, the other minimizes it. Alpha-beta pruning reduces the number of nodes that need to be evaluated without changing the result. This approach has proven effective in games such as chess, checkers, and Othello.

However, the Minimax approach has fundamental limits. In the game of Go, the branching factor, i.e. how many children each node can have, is so large that exhaustive tree search becomes computationally intractable, even with pruning. Pruning is a technique to narrow search, to find out which branches in the tree do not need to be searched at all. A deeper problem arises in games with hidden information: when a player cannot observe the full game state, the search tree cannot be constructed in the usual way, because the true state of the world is unknown. This thesis concerns precisely this second challenge — how to learn to play strongly in a card game where each player's hand is hidden from the other, without relying on hand-crafted heuristics, based on domain knowledge.

1.3 Randomness and Imperfect Information

If a game contains randomness or hidden information, it is not possible to know with certainty which state will be reached by making a given move; probability therefore inevitably enters the search. Sometimes the probability distribution is known (e.g. a coin flip), and sometimes it is not known in advance, even though a distribution implicitly exists (e.g. a slot machine). In reinforcement learning, a large part of the learning task can be precisely the task of learning that probability distribution: what is the probability one transitions from one state to another, or what is the probability the reward obtained from a new state being of a certain kind, for example win, loss, threat.

Card games often involve hidden information, because a player does not know the opponent's cards. There may also be randomness if, for example, not all cards are dealt into the game, or additional cards are drawn from a shuffled deck. In such cases, building a search tree is not as straightforward as in a fully observable and deterministic game. Methods have been developed for this purpose in which the search space is determinized by sampling some subset of the hidden information and proceeding as though this were known fact [4, p. 2]. For example, in a card game where ten cards have been dealt to each player from a standard 52-card deck, one can — knowing one's own cards — sample ten cards for the opponent from the remaining 42 and conduct the search under that assumption. The sampling becomes more precise as the game progresses, provided the rules of the game are known. It may be known, for instance, that a player does not hold a particular suit if they were unable to follow suit when it was led. In addition, cards seen during play reduce the amount of hidden information. This sampling approach gives rise to a couple of theoretical problems, which are discussed in the Section 2.4 .

If, for one reason or another, building the entire search tree is not practical, one can use models based on probabilities and heuristics to guide the construction of the search tree, weighing more promising directions more heavily. In this work so called domain expertise, experimental knowledge of how the game is played, was deliberately excluded, so heuristics were not an option. The search tree is built using so-called Monte Carlo method, which is described further in Chapter 2.

1.4 AlphaGo, AlphaGo Zero and AlphaZero

AlphaGo is a series or a family of machine learning systems. They are developed by Google's DeepMind group. The group's goal was to defeat human professionals in a game that had always been thought to be too difficult for computers. Originally the AlphaGo Lee won in the game of Go a professional, master-level player Lee Sedol, in March 2016. That generation of AlphaGo was trained using human knowledge and old games played by professional level players. After the initial learning from human knowledge, the game training continued by self-play against itself and then training the model with that data [5]. AlphaGo Zero (October 2017) was trained with no human knowledge at all [6]. It won the original "Lee" 100-0, showing the power of this approach. The next generation was AlphaZero (December 2017) that generalized the approach to other fully observable games like chess. This proved that the method is not limited to Go and it can in those other games also achieve, tabula rasa,

superhuman performance [3]. This last one, AlphaZero, originally inspired me to consider trying the method also to a game of hidden information in this thesis. My concern was to find a game small enough to be self-played and trained on a regular home computer.

1.5 From Perfect Information to Imperfect Information Games

The AlphaZero approach demonstrated that tabula rasa self-play reinforcement learning can produce expert-level performance in perfect information games — without any domain knowledge, heuristics, or human game data [3]. Whether the same approach can transfer to games with hidden information is an open question. Previous work on imperfect information card games typically relies on hand-crafted heuristics derived from domain knowledge and from expert players. This thesis investigates whether AlphaZero-style self-play, combined with a search method that handles hidden information through determinization, can learn competitive play in a trick-taking card game without any such heuristics.

1.6 Research Question, Scope and Contributions

I started the thesis work with one research question in my mind. It was not written down but discussed with my thesis supervisor. During the experimentation phase, test runs and model evaluation, I found that there is another question, quite closely related to the original one that I am also answering. The primary question can be stated like this: “Can AlphaZero-style self-play reinforcement learning produce competitive play in a two-player imperfect-information card game without domain-specific heuristics, and how does imperfect information limit achievable performance?” The secondary question started to be even more interesting as it was something I found through experimentation: "Can performance be decomposed into interpretable layers that separate learned model quality from the structural cost of hidden information?"

This thesis makes three contributions. First, it applies AlphaZero-style self-play reinforcement learning to a two-player trick-taking card game with imperfect information, learning solely from self-play without any domain-specific heuristics. Second, it empirically identifies and measures a stable information barrier caused by hidden cards, quantifying its cost as approximately five percentage points in win rate across all opponents and training stages. Third, it presents a four-layer evaluation framework that separates random baseline, learned model quality, information barrier cost, and minimax ceiling. These enable precise measurement of what is learned versus what is structurally limited by hidden information.

1.7 Structure of this thesis

Chapter 2 presents the theoretical background covering Monte Carlo Tree Search, Perfect Information Monte Carlo, reinforcement learning, and deep learning concepts that form the foundation of this work.

Chapter 3 describes the experimental setup: the two player trick-taking card game used as the test domain, the neural network architecture and state encoding, the MCTS and PIMC implementation, the self-play training loop, and the evaluation methodology.

Chapter 4 presents the results, including learning progression over 618 training iterations, a four-layer performance decomposition separating model quality from the cost of hidden information, head-to-head model validation, and computational resource analysis.

Chapter 5 discusses the findings, interprets the information barrier result, reflects on the suitability of the PIMC approach, and suggests directions for future work.

Chapter 6 concludes the thesis.

1.8 Use of AI tools

Artificial intelligence tools were used in two ways during this work. In the implementation phase, Claude Code (Anthropic) was used as a coding assistant in a pair-programming style via an agentic command-line interface, where the author directed the design and architecture while the AI assisted with implementation. The tool also assisted in code debugging, and in the technical setup of the required software components and libraries to author’s computer. In the writing phase, Claude was used to help draft and structure parts of the thesis text based on the author's own research notes, results, and code documentation. Claude was used to keep track of the ongoing work in my multiple drafts per chapter. All research decisions, experimental design, theoretical framing, and the interpretation of results are the author's own work.

2 Methods and theoretical background

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is defined as "a class of tree search algorithms that make use of simulated games to evaluate search tree nodes that are not terminal states of the game. The simulated games make random choices until a terminal game state is reached, and the value (reward) is obtained by averaging the results of several simulations" [7, p. 123].

The AlphaZero algorithm modifies the simulation idea slightly, so that from a leaf node the game is not simulated to the end, but instead the machine-learned model is asked for the leaf node's "value", the probability of winning from that position. At the same time, the model returns the probabilities for the next moves, i.e. their a priori strengths.

MCTS has some key properties. It is non-heuristic - it does not use or need any heuristic, anytime - the algorithm can be interrupted at any time and still return an answer. Additionally, it is asymmetric, the MCTS search can focus on promising search directions [8, p. 25].

Idea is to run random game simulations from the root. Nodes represent states and edges are decisions or actions that lead to next state from previous one. After several random games, a tree has been built. For state evaluation purposes (goodness of a state), the count of visits to a certain node is stored with sums of the values of the games played through that node. Value is typically found from the leaf nodes, and it is then passed on to the parent nodes. This is called backup in the algorithm.

Algorithm in the simplest form consists of only two steps:

1) Start building the search tree with only one node (root).

2) When a random game visits already visited node, create a new next move node, if it does not exist already.

During iterations the randomness is altered, and the more visited nodes are preferred. This leads to “best-first” growth of the search tree.

In a bit more detail, the MCTS works in 4 steps shown in Figure 1, from [9]: Selection, Expansion, Simulation and Backpropagations.

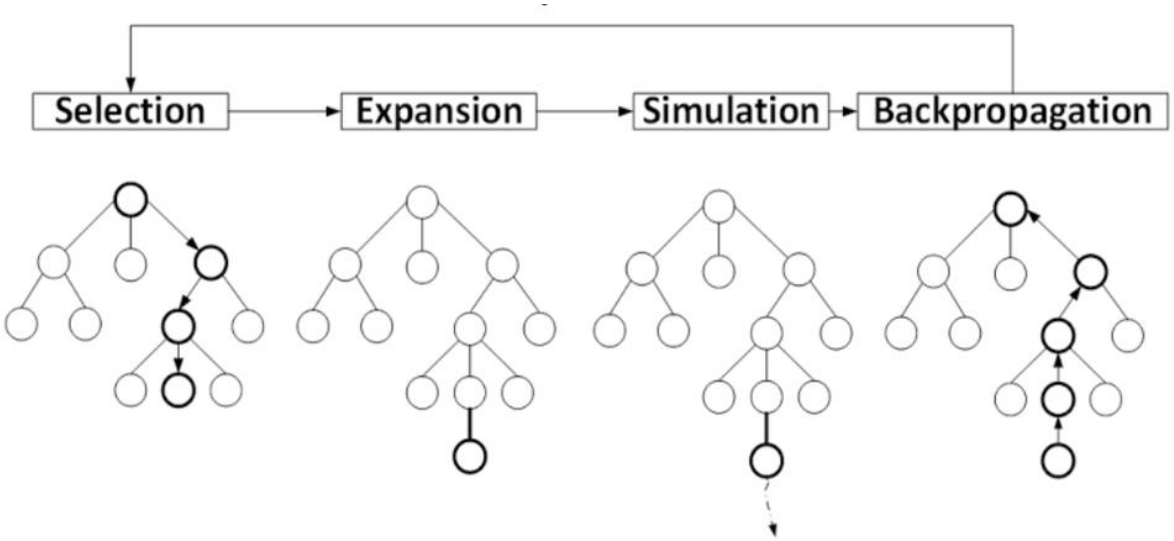

Figure 1: MCTS Algorithm Key Elements

- Selection: Starting from the root, the current game state. Selecting the child nodes with the most promising value, continuing to children's children and eventually arriving at a leaf.

- Expansion: If the leaf is not a terminal state, i.e. the game ends there: create children by applying possible actions.

- Simulation: Play (simulate) the game from that node onwards using some method. In AlphaZero algorithm, the chosen approach is that no simulation is performed — instead, the value of that node (a priori) is obtained from the machine-learned neural network. In this way of thinking, the neural network acts "like a simulation" to obtain the value of the node and the information about the probabilities with which the next moves from this new state will lead to its children. This is the whole idea — the neural network is trained to produce values that match as closely as possible what MCTS computes at its root, because it is believed that MCTS, by exploring different directions in a guided way, has found slightly more accurate estimates of the situation than a static analysis of the state alone.

- Backpropagation: from the leaf node after the simulation, push game values and game outcome upwards to the parents, so that upper nodes also get the information, what is to be found in this direction.

An important feature of the MCTS is selectivity: In this tree search, as the tree grows the move probabilities are adjusted [10, p. 3]. Searching bad moves does not matter if a good move is still selected. Another feature is backup with uncertainty: how much child nodes value affects the value of its parent.

2.2 Hidden or imperfect information in search

Building a search tree is straightforward when it is known with certainty what can follow from each move, and when the opponent's possible moves can therefore be listed. When this information is not available, and the search involves hidden information and even randomness, this property must be modelled and somehow accounted for. If the rules of the game are known, that information can be used to advantage: for example, when a suit must be followed, and if the opponent was previously unable to respond to a certain request, the set from which we are guessing can be narrowed down. Together with deep learning, the use of a machine-learned model has been studied — for instance by training a model to estimate the most probable game states based on move history [11].

One algorithm that has been used for handling imperfect information in search problems is Perfect Information Monte Carlo, PIMC [12]. PIMC is a method in which imperfect information is converted into perfect information from the search algorithm's point of view — the unknown search space is determined by sampling a subset and continuing as though that sample were known fact.

In article [13] it is noted that in games of imperfect information there are two separate problems: move selection and inference. Move selection is the problem of choosing a good move even without complete understanding of the situation, and this is generally handled using a search algorithm combined with static state evaluation. Inference is the task of deducing hidden information based on what the opponents do. These problems are separate but influencing each other, because good inference leads to a more informed move, and a good move can cause part of the opponent's hidden information to become better known.

2.3 Applying MCTS to games of imperfect information and randomness

The use of search trees for game search problems has been shown to work effectively, and it has also been proven that in perfect information, non-random games the minimax algorithm finds the optimal move [1]. The MCTS algorithm can be applied to any search problem that

can be modelled as a decision tree and it can also be used without domain expertise, although having such expertise does bring benefits [8, p. 25].

Several article sources note that applying MCTS directly to games of imperfect information does not work in a straightforward way, for example [14, p. 2]. In his doctoral thesis, Whitehouse points out that imperfect information causes the game tree to grow combinatorially, which creates problems for fixed-depth search approaches since only a shallow search can be performed even with a large computational budget [8, p. 7]. On the other hand, the same source notes: ”The asymmetric nature of MCTS provides a mechanism for overcoming this limitation and additionally MCTS has shown to be effective in large branching factor games such as Go”.

In article [15] this problem is approached in two ways: Determinized MCTS, which is the same as the PIMC mentioned above, and Information Set MCTS, ISMCTS, the idea behind the latter being presented in article [7].

A very good overview of MCTS applications over the last ten years can be found in [16]. A fairly similar trick-taking game application for the game Jass has been done in [15], and a trick-taking game Doppelkopf has been studied in a bachelor's thesis [17].

2.4 Perfect Information Monte Carlo – PIMC

When a game is stochastic or involves imperfect information, modelling it as a search tree requires some additional steps. One approach is determinization, which means converting an imperfect search situation into a perfect one. This can be done by sampling some possible subset from the hidden information and assuming that subset is then known. Put most simply: we guess that the opponent has cards a, b, c, d, e in their hand and then build the search tree under that assumption.

One theoretical problem with the method is strategy fusion and non-locality [8, Ch. 5]. The former means that a player makes a different decision from the same situation in different plays of the game, because of the determinization: depending on the sampling one can from same state sometimes make action X and sometimes Y. Non-locality means that depending on the history of game states, the probability of the hidden cards varies — it is not dependent only on the current state. PIMC has been criticized for example by Russell & Norvig [1, p. 217], who describe it as "averaging over clairvoyance" that "no sane person would follow", and also on the grounds that the method cannot recognize or play a bluff. Despite this

criticism the method has successfully learned to play games. Once the method was found to work experimentally, some theoretical reasons for this were eventually found as well — these are presented in the article "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search" [12].

The method can seem quite unreliable, especially when there is a lot of hidden information. The search can be extended by creating multiple determinizations, each with its own search tree, or an ensemble, and combining their search results as an aggregate [18]. In this thesis' experimental part described in Chapter 3, I followed the ensemble method. This kind of thinking could be applied beyond games as well, in a scenario-based style: assume different scenarios, where each scenario is a sample, and find a suitably weighted compromise from their outcomes.

In this study's experimental part, I chose PIMC instead of alternative approach ISMCTS, which is described in the next section. The core reason for this choice is compatibility with the tabula rasa principle. ISMCTS improves PIMC, it identifies game states across different determinizations that are equivalent from an information perspective and sharing computation between them [7]. However, recognizing which states are informationally equivalent requires reasoning about the structure of the game itself and that would require domain knowledge. The goal of this study was to apply AlphaZero -style learning with no game-specific knowledge given to the algorithm, use of ISMCTS would contradict that principle. PIMC makes no such demands: it simply samples a possible world and runs a standard search. However, if I wanted to have as strong player as possible with any means available, I would probably use ISMCTS.

2.5 Information Set Monte Carlo Tree Search

ISMCTS is an attempt to use the information set method as an aid to MCTS. Its principles are presented in article [7] and it is applied in article [4] to a card game with some further development. ISMCTS is different from determinized UCT search. Determinization has some weaknesses — computational capacity must be divided between the sampled sets even when those sets share common nodes, and this has not been exploited in the algorithms. Put briefly, ISMCTS does not run minimax on sampled information but instead treats the tree nodes as information sets rather than game states.

ISMCTS exploits the observation that when searching with determinized samples, the same states will be visited across different samples. That is, if you play one determinized game all the way through its search tree, that tree will at some point contain the same states as the search tree derived from a differently sampled starting point. For example, if a card game sample A has 8 cards in common with sample B (distributed to the same players), and the total number of cards is 10, then if those two different cards are played first in the search tree actions, the search trees are identical from that point onward. PIMC does not exploit this information and therefore ends up using more computational capacity than necessary [7, p. 120].

A similar observation can sometimes be made in other algorithms outside of game playing — computational advantage can be gained by recognising similarity between subproblems. This is a typical step forward in for example dynamic programming, where the problems solved are often based on recurrence relations, and the same subproblem can repeat itself many times through the recursion. Recognising this kind of situation is where general knowledge of algorithms research is useful, because the solution patterns are already known.

Forming the information set appears to exploit the structure of the game itself, and for that reason it does not fit well with the AlphaZero approach, where nothing about the game is assumed to be known. Based on that, I decided to use PIMC instead of ISMCTS in the experimental setup.

2.6 Self-Play

Supervised learning in machine learning requires a lot of data. In “traditional” setup, the machine learning system would use labelled data from human played games: “In a game state X, the player selected Y, and the outcome is Z”. In my thesis experiment setup, the Y and Z would be the labels, i.e. the predictions, from input vector X.

When one wants to train a model to play games, there must be lot of data from games available. In the AlphaZero -approach there is no data when the learning is starting. Self-play, a method of automated or programmed agents playing against each other, is producing training data. In the beginning, the data is strategically poor, as it is not based on any understanding of game mechanics. However, the data is still correct: it is from actual, simulated game play situations (game states) and the outcomes who won, are also correct.

"This kind of actions from certain state, produce that kind of outcomes". This data is then given to a supervised learning system.

The self-play is based on iterations. There is a number X of self-play games per iteration. Then those game states and outcomes are used to train the model. The model, if it got better, is then used in next iteration where self-play creates again X self-play games of data. Assumption, or the hypothesis, is that as next iteration of self-play is using better model than previous iteration, the training data from those newer games is strategically better than earlier. That better data is then used to teach the next model.

In the original AlphaZero algorithm, each iteration uses in self-play the so far best accepted model. Silver et al. played 25000 games per iteration, using 1600 MCTS simulations in each move selection. In their training setup, for the first 30 moves of a game they are using temperature parameter τ=1, which means more exploratory play in the beginning [6]. Temperature parameter is described in more detail in Section 2.9.3. After those 30 moves the temperature parameter gradually goes down to 0. Search gets greedier. Silver et al. add Dirichlet noise to the a priori probabilities in the MCTS node expansion, to ensure all moves may be tried. As their game amount is big and computation is required a lot, they also stop the games that are clearly lost, to save computation time. Verifying the "clearly lost" is not trivial in their case, game of Go, but in my trick game it is rather simple. In these self-play games, each game position, or state, is stored. The self-play search result, i.e. the move probabilities and the eventual game result (outcome) are also stored with the state. That is then one sample for supervised learning.

In AlphaZero, the self-play is de facto the mechanism for reinforcement learning. In a reinforcement learning system, described later in Section 2.9: an agent interacts with its environment to generate a reward signal. Here, the agent is the player itself, opponent and the game simulation is the environment, a game is an episode – a sequence of play from card dealing to the game end, the move (or sequence of moves) is the interaction, and the game result is the reward signal. The agent does not have any external instructions of what a good or bad move may be, it discovers this all from the game outcomes. It should be noted that the term model in Reinforcement Learning concepts is not the same thing as the neural network model in supervised learning. In Reinforcement learning, the model means the learned transitions $P(s'|s,a)$, the probability of moving to state state s', when doing action a from state s.

In this study, the same principle is applied, on a considerably smaller scale. Best accepted models are used in next generation of self-play, each game producing labelled data for training the neural network. The game selection is done keeping computational budget in mind.

2.7 Machine Learning background

Machine learning can be seen as a computer program that learns to make decisions based on the data it receives. In a traditional program, the programmer lists the conditions and rules for every decision, and the program follows them exactly every time. In a machine learning situation, the rules are often unknown in advance or simply too large to enumerate — for example, the rules for recognising a bird in an image would be impossibly complex to write out manually. There is also inherent uncertainty in the decisions a machine-learned model makes: a traditional program can be fixed until no errors occur, but in machine learning you either accept a certain error rate, train longer, use different training data, or change the training approach entirely.

Machine learning needs data. In supervised learning, each data point has a label telling the system what it should predict from that data point, what the correct output should be, given the input. The label is the “supervision”, supervisor is saying what the right answer should be. In unsupervised learning there are no pre-defined labels, and the goal is to find structure in the data. Goal could be for example grouping similar data points together. In reinforcement learning that is covered in more detail in Section 2.9, the data comes from exploring and interacting with the environment.

Both supervised and unsupervised learning systems need a representative amount of training data. Because learning in those paradigms is in the end about finding statistical patterns in data, if the training sample does not represent the full data space well, the conclusions the model makes will not be reliable. This can be compared to a poll: if the opinion is asked only from under 30-year-old men, the result is not likely to describe the opinions of the whole population. There is also a risk when the neural network model is large with many layers, neurons and learned weights, but the training data is small: the model may simply memorise the training data instead of learning a general pattern. This is called overfitting.

The fundamental power of machine learning is that a neural network can find patterns and features in data that a human would not notice. A good, trained model has built an internal

representation of the data that allows it to give useful answers even for inputs it has never seen before. That is called generalisation. In this study, that input that has never been seen before is a game state and the output is an evaluation of that state and a probability distribution of possible moves. The probability then is interpreted as the goodness of the move, “a good player would in this state do this, with this probability”. A key method in ensuring the model generalisation ability is to split the data into training and testing set. Test data is not ever shown to the model during training, and the goodness of the model is evaluated by measuring how well the model predicts the test data. Properly done splitting of the data also mitigates the risk of overfitting. In this study, generalisation is evaluated through game outcomes against fixed opponents and earlier model versions rather than through a test dataset, since the objective is playing strength rather than prediction accuracy.

2.8 Deep Learning

Deep learning is a family of machine learning methods where the model is structured as a layered computational network. The word deep refers to the number of layers, sequence of computation from inputs through many layers to produce the output [1]. This layered structure allows the network to build abstract representations of high-dimensional inputs. This is why deep learning has become one of the most common approaches in areas such as image recognition, speech processing and game playing.

The most common network types are feedforward, recurrent, convolutional and residual networks. In a feedforward network, information moves only in one direction: each node, a simple computational unit, in a layer passes its output forward to the next layer. A unit computes a weighted sum of all its inputs, essentially a linear combination, and passes the result through a nonlinear activation function. It is important that the activation function is nonlinear, otherwise the composition would still be a linear function of the inputs. Universal approximation theorem says that a feedforward network with only one hidden layer and sufficiently many hidden units, can approximate any continuous function to any desired degree of accuracy [19]. This is important because the network usually needs to learn complex patterns. Without this ability, it would be linear regression, except when combined with some mathematical transformations to adapt to non-linear relationships. Because the key operation is matrix multiplication, efficient linear algebra libraries are at the core of practical neural network implementations. In a recurrent network, some outputs are fed back as inputs to earlier layers, which gives the network something analogous to memory or internal state.

When training a supervised learning model, the network's output is compared to the known correct output (label) using a loss function. It can be for example the Euclidean distance between two vectors. The difference, or error, is then propagated backwards through the layers in a procedure called backpropagation. At each layer, the unit weights are adjusted slightly in the direction that reduces the error. This is gradient descent: the principle is the same as Newton's method for minimising a single-variable function: differentiate, then step in the direction of decreasing value. In a network with many parameters, the derivative becomes a gradient, and adjusting the weights in the direction of the negative gradient moves the loss toward a lower value. A simple visualisation of the idea with a two-variable function, like a hilly landscape, where the gradient is then the steepness of the hill. Going to negative gradient takes one lower, and this is where we want to go when minimising the error function. All optimisation through differentiation and gradient descent have a significant risk of finding only local minima, but there are many methods to avoid this too. This may seem a bit complicated, but in practice all this differentiation and weight modifying to minimise error, is handled automatically in modern neural network frameworks. With those, basic application of the software libraries does not require a deep understanding of mathematics.

A practical problem with very deep networks is that the gradient signal weakens as it travels backward, from the output layer towards the input layer, through many layers, making the earlier layers difficult to train. The error signal can be considered as lost before it reaches the first layers. In some cases, the gradient can explode, which is just as bad as the vanishing gradient. Residual networks address this by adding shortcut connections that skip one or more layers and pass the input directly to a later point in the network. These shortcuts give the gradient a direct path backward and allow much deeper networks to be trained reliably. The layer learns the residual: the difference between its input and the desired output. The neural network in the experimental part of this study uses a residual architecture, which is described in detail in Chapter 3. The deep learning and neural network architecture design is a topic of its own with a huge amount of published study. In this thesis work, the emphasis was not in the optimal network design but in applying one that seemed feasible based on the literature.

2.9 Reinforcement Learning

2.9.1 Reinforcement Learning Fundamentals

Reinforcement learning is one of the machine learning paradigms, alongside supervised and unsupervised learning. The key difference is in how the learning happens. In supervised learning, the system learns from labelled examples — for each input, the correct output is known, and the system is told when it makes a mistake. In reinforcement learning, there are no labelled examples. Instead, an agent is placed into an environment. Agent learns by interacting with the environment and getting evaluative feedback from it, a reward signal. Reward signal tells the agent "What happened", but not what it should have done instead. Sutton and Barto describe this as the difference between evaluative and instructive feedback, and selecting actions based on evaluative feedback is the defining characteristic of reinforcement learning [20, Ch. 1]. One explanation of the difference between evaluative and instructive could be "doing an action was followed by a reward of 20 euros" vs "action gave 20 euros, but it should have given 100". There is a psychological analogy to the reinforcement learning: positive feedback strengthens behaviour and negative feedback weakens it.

The core elements of a reinforcement learning system are:

- the agent, the learner and decision-maker
- the environment, everything the agent interacts with
- the state, the current situation the agent is in
- the action, what the agent can do in that state
- the reward, a scalar signal received from the environment after each action;
- the policy, a mapping from states to actions that tells the agent what to do in each situation.

Closely related to the policy is the value function, which estimates the total future reward achievable from a given state when following a given policy.

An essential feature of reinforcement learning that separates it from other machine learning paradigms is delayed reward. The consequences of an action may not be felt until many steps later. In some games the reward is instant: a goal in a football game for example. But often

the arrives only at the end: one player wins and the other loses. In such cases the reward is called sparse. This creates the credit assignment problem: which of the many moves played during the game deserve credit for the outcome? This problem has been addressed already in 1960s, Minsky studied “complex reinforcement learning systems” and he gave an example, where “one million decisions is needed to win a chess game, should each decision element be given a millionth of the credit?” [21]. There is a lot of study of this and related problems, Markov Decision Processes. Those consider for example a discounted return where early reward is considered higher in value than the same reward but later. In the experimental part of this study the simplest possible approach is used: the final game outcome is assigned uniformly to every move in the game. A win assigns +1.0 to all moves played, a loss assigns −1.0. This is a deliberate simplification. It does not distinguish good moves from bad ones within a game but relies on the law of large numbers: over thousands of games, moves that tend to appear in winning games will receive more positive signal on average.

Reinforcement learning in practise is often done in simulation, and for good reason. In real-world environments mistakes can be costly or irreversible — a self-driving car cannot learn to drive in live traffic where an error could be fatal. In simulation, a mistake simply resets the episode. In this study the card game is fully simulated: every game resets cleanly, there are no fatal states, and the agent can generate as many episodes as the compute budget allows.

Reinforcement learning algorithms are typically classified as model-based or model-free. In model-based approaches the agent either knows or learns a transition function — a model that predicts which state results from taking an action in a given state. An example of model-based could be a chess game, where the rules are known, what happens when a move is conducted. In model-free approaches no such transition model is learned; the agent learns the value function or policy directly from experience. The experimental system in this thesis is model-free in the traditional sense: the neural network learns the value function and policy directly from self-play outcomes without building any explicit transition model. However, it is worth noting that MCTS requires a simulator of the game rules to run its tree search. This simulator is not learned, it is the known, deterministic rule engine of the card game. Some literature therefore describes AlphaZero-style systems as sitting between model-free and model-based: no learned transition model, but dependent on a perfect given model.

2.9.2 Markov Decision Process

Reinforcement learning is formally grounded in the theory of Markov Decision Processes (MDP). An MDP provides the mathematical framework for sequential decision-making: an agent in some state takes an action, transitions to a new state according to some probability distribution and receives a reward. The Markov property states that this transition depends only on the current state and action — not on any earlier history, in probability theory it is "memoryless property of a stochastic process" [22].

The mathematics in RL and MDP are very close to each other, as they are about state transition probabilities, reward probability distribution, discounting. One difference in the practical approach, however, is that a MDP can be solved "from outside" when knowing the models attributes but in RL the agent is "inside" the model trying actions and making observations.

The components of an MDP apply directly onto the card game used in this study. The state is the information available to the player at their turn: their own hand, the cards played in previous tricks, and whose turn it is. The action is the card chosen to play. The reward is +1.0 for winning the game and −1.0 for losing, received at the end of the final trick. One episode — a complete sequence of interaction from start to terminal state — consists of five tricks, giving ten moves total (five per player). The transition function is the game rules: given a state and a played card, the next state is determined by the rules of the game.

Strictly speaking the game in this thesis is a Partially Observable MDP (POMDP) rather than a full MDP, because the player does not observe the complete game state — the opponent's hand and the undealt cards remain hidden. This is the fundamental source of imperfect information in the game. The determinization approach, described in Section 2.4, handles this partial observability during search by sampling possible completions of the hidden information.

An important observation about MDPs is that even if the agent knows the environment and its rewards perfectly, it may still face a difficult learning task. The Rubik's cube is a classic example: the mechanics and the outcome of every move are fully known, yet finding the optimal solution requires deep search. Knowing the rules does not mean knowing how to solve it.

2.9.3 Exploration and exploitation

One of the central challenges in reinforcement learning is deciding when to use what you already know and when to try something new. If the agent always chooses the action it currently believes is best, it may never discover better alternatives it has not tried. If it always explores, it never builds on what it has learned. This trade-off is called exploration versus exploitation. Managing it is essential for learning a good policy, e.g. what to do in a given state.

A classic example to understand this is the multi-armed bandit problem — named after slot machines, sometimes called one-armed bandits. Imagine a row of slot machines, each with an unknown payout probability. You have a limited number of pulls. Do you keep playing the machine that has given you the best results so far — exploiting your current knowledge — or do you try a machine you have played less, which might be better? To find out which machine is best you must spend pulls on machines that might not pay off. This trade-off between exploiting known good options and exploring uncertain ones is one of the fundamental problems in reinforcement learning and has been studied extensively. Under certain conditions the problem is equivalent to a Markov Decision Process.

In this study the exploration-exploitation trade-off is handled by the MCTS search itself through two mechanisms. The PUCT formula used during tree search includes an exploration bonus that scales with how rarely a node has been visited. It is naturally directing search toward promising but less-explored actions. Secondly, at the move selection, a temperature parameter τ controls how exploratory the final choice is: during the first six moves of each self-play game $\tau=1.0$, meaning selection is proportional to visit counts and a range of moves is explored. After that $\tau=0.0$ and the most-visited action is always selected. During evaluation phase temperature is always zero and play is fully greedy.

2.9.4 AlphaZero as a reinforcement learning system

The algorithm used in this study is an instance of reinforcement learning. The policy improvement operator is MCTS: Neural network gives its prediction and MCTS makes it better. Each search produces a visit distribution over actions that is a better estimate of the optimal policy than the neural network's output alone. The value function is approximated by the neural network, trained on outcomes of self-play games. This combination, MCTS-guided

policy improvement and neural network value estimation, trained entirely from self-play, is the essential structure of the AlphaZero algorithm.

The self-play loop described in Section 2.6 is the reinforcement learning loop in concrete form. The agent interacts with an environment, the card game, and receives a reward at the end of each episode. Training loop, gating, evaluation methodology and other implementation details are described in the Chapter 3.

3 Experimental Setup

3.1 Case study: card game ‘Trick’

To be able to respond to my research question, I decided to apply AlphaZero method into a game that is simple enough to model as a programming exercise but still having the imperfect information and randomness involved. I decided to create a trick-taking card game, and to play a version of it where there are 2 players, not all the cards are dealt, and where not only the last trick wins but the count of won tricks during the game.

I reasoned that the search space could not, for computational complexity reasons, be too large for home computer setup. I also wanted for practical reasons to see quite quickly, within hours instead of days or weeks, does the approach work or do I need to fine-tune any parameters. I thought that a reduced deck of cards, where several cards are removed, would still model the research question and search space but with just smaller amount of hidden information. And the amount of hidden information would correlate with the amount of training games required. After some experimentation I decided to have a deck of 15 cards, with 3 suits and 5 cards per suit, and both players being dealt 5 cards. This way there still is some hidden information as 5 cards are not dealt.

One concrete thing was that I needed to be able to find the absolute optimum, as I did not have any superior master level human players available to do the evaluation of the goodness of the system learning results. Comparing my model to that optimum was a significant requirement. It is worth noticing that even optimal player does not win all games, as some deals are just such that one cannot win with them. With a smaller than normal deck the optimal baseline calculation became more feasible. Without such a ceiling value, it would not have been possible to distinguish between the model goodness from the factor the imperfect information brings. In the further chapters, this gap will be defined as information barrier. From these evaluation points of view, the selection of a smaller deck is a methodological choice, not merely a scale-down version.

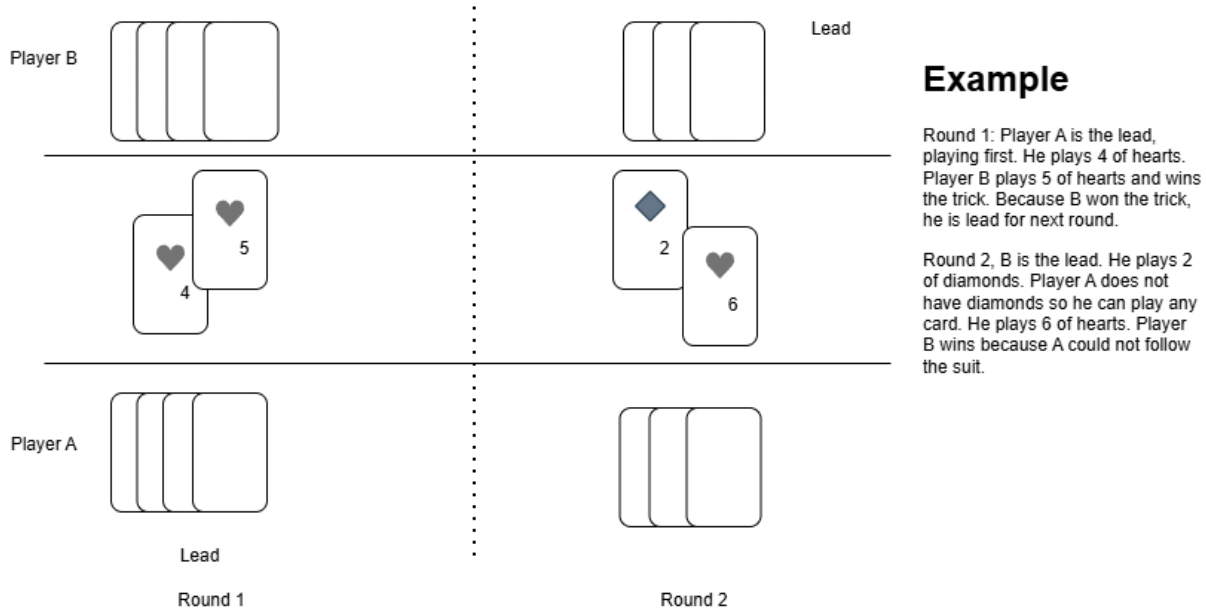

Figure 2. Trick Game Example

Figure 2 shows an example of the game mechanism. The game itself is played with hidden hands, both players have random 5 cards in the beginning, and first player – called the 'lead' - puts one card from his hand to the table. Another player responds, he needs to 'follow suit', meaning he must play a card of the same suit than the player who started the round. The player who plays highest value card of the leading suit, wins that round - a 'trick'. Next round (with 4 cards now in hand) is started by the player who won previous trick. Figure 1 shows two example rounds, where player B wins both rounds, 2 tricks. Basic variations of the game are "who gets most tricks" and "who wins the last trick". In this study I decided to program the version where "who gets most tricks" wins. The system does not have any domain-specific heuristics or evaluation. Only the selection of legal moves is programmed to the search, as well as the follow-suit rule. Strategic state evaluation is done by the neural network at every leaf node, and that network is trained through self-play.

3.2 Test system

3.2.1 Overall architecture

The software of experimentation consists of modules Coach, Evaluator, Network, MCTS and Storage. The functions implementing the game rules described above are in game engine that is used, directly and indirectly, by these modules. Network and Storage do not use game engine, for the Network this is also a question of principle, as it should only be using encoded

game state data without access to domain knowledge, and Storage is just managing data movements and persistence. Figure 3 shows these modules and their relations.

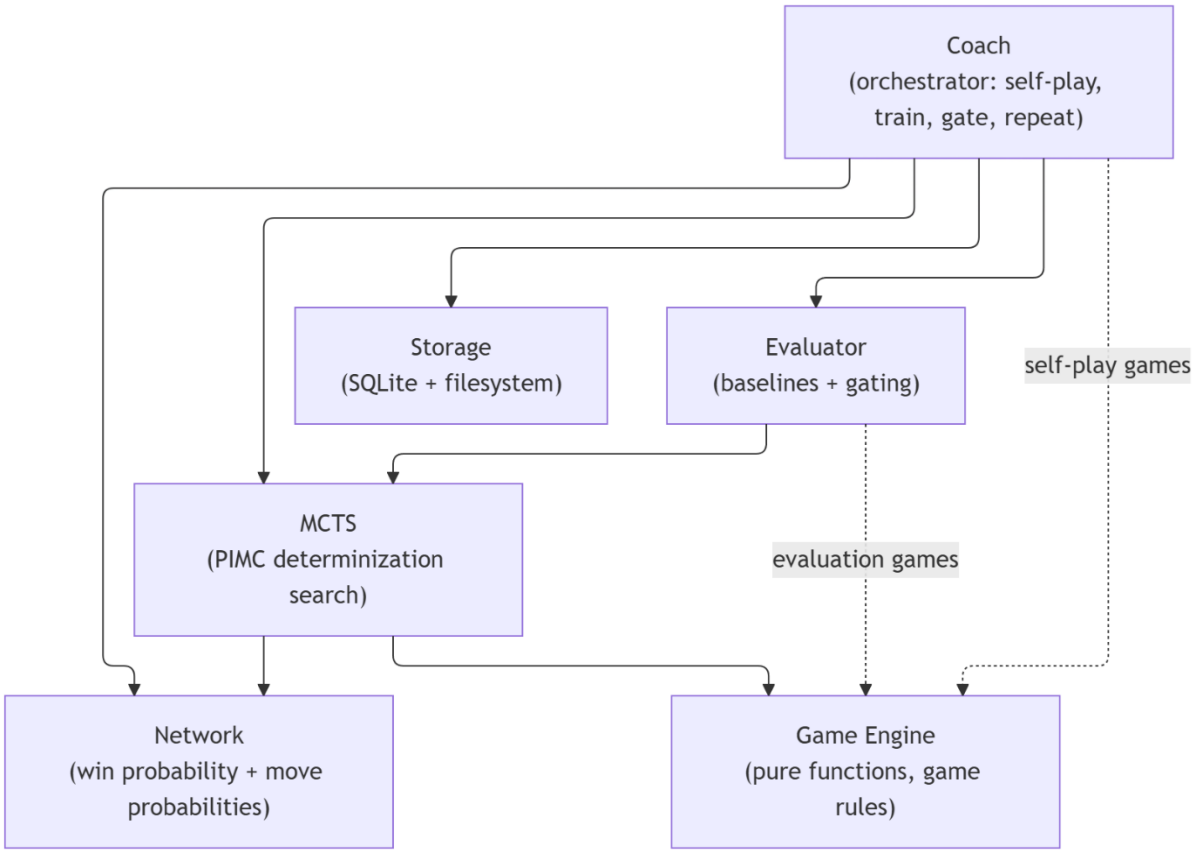

Figure 3. Experiment System Architecture

“Coach” is the orchestrator or the “main” program that creates self-play games, stores the results and needed information, trains next version of ML model, does gating, i.e. compares results to earlier best model and repeats all this until stopping conditions are met. Those conditions are either time in minutes or training plateau – meaning that no new models have been better than the current best model during X generations where X is a parameter.

“Evaluator” is used to play a model against fixed strategies players, those are: Random who plays random but legal move, Highest plays highest numeric value card that is allowed to play, Lowest plays lowest allowed card. Some baselines were run first to test game mechanics, playing those fixed strategies against each other in all combinations. This is also the mechanism “Coach” uses internally for gating when comparing candidate model against the current best.

“Network” is the neural network that is trying to learn two things: probability of winning from given state and from that state the probability of playing each legal card.

“MCTS” is Monte Carlo Tree Search that is used to find best possible move from given state. It uses determinizations in Perfect Information Monte Carlo (PIMC) style – sampling opponent cards from the set of unknown cards and running 10 simulations from each sample.

“Storage” contains classes that store models and games to a simple database and to file system. Those classes also help to load training data from the database.

A more detailed, class-level view of component connections follows in Section 3.8.

3.2.2 Orchestration

The AlphaZero principle is straightforward, system puts players to play against each other, they do not know the rules of the game, but the system or “referee” just prohibits playing illegal moves and after the game it tells who won. When players are “thinking forward” i.e. conducting MCTS, all the information about the situation “goodness” is coming from the machine learned neural network model. MCTS search results (card play probabilities) and game results are both stored to train next version of the model. New model is put into play against old best model and if it wins enough, it is selected as new best model. This procedure is called “gating”. Then the loop starts from the beginning.

3.2.3 Baselines and evaluation

I wanted to have some baselines for comparison as the whole point of this study is to remove a “human player master” from the setup. I put the fixed strategies to play against each other, then I implemented a “cheat mode”, where machine learning player has perfect information about opponent hand, this was to evaluate model and MCTS goodness without the “information barrier”. Then I implemented a minimax-player with perfect information and put that to play 4000 games against fixed strategies. I think this approach is a novel idea and it was feasible to implement with these small decks. With minimax I could see how good the ML model with MCTS and perfect information could be, “absolute optimum”. With quick intuition one might think that it would win 100% of the games but that’s not the case as can be seen from the baseline games. From this limit I could see that there’s not necessarily point in training more, when getting closer to that optimum.

My hypothesis is that there is an “information barrier” – the difference, percentage points in measurements, between minimax optimum and ML + MCTS model: no matter how good the model and MCTS is, that barrier is there because of imperfect information. The amount of “information barrier” can be estimated by using the “cheat mode”. The decomposition is shown later in Chapter 4.

Information barrier could be made smaller by introducing some heuristics and learning from what the opponent has played in some situations. Basically, one cannot know for sure when having bigger deck sizes, but the size of unknown would be smaller. That would potentially make better results from determinizations. In principle, the information barrier should be diminishing over time as the model could learn from the past moved. But in that case, I would need to input also past moves to the network, that changing the network and input architecture quite a lot.

3.2.4 Modelling the neural network for a card game

One of the first things was to model the state representation. That means the search tree state that is given as input for the neural network. Card encoding is just an index 0-14, I have 3 suits (A, B and C) and 5 cards (0,1,2,3,4) per suit: card index = suit x 5 + rank. Things to be encoded into input are (6 vectors):

- The cards currently in “my” hand (bits 0-14 in the encoded input vector)
- The cards already played in the game, e.g. “seen” (15-29)
- Cards I have played (30-44)
- Cards opponent has played (45-59)
- The leading card in current trick, if any (60-74)
- Legal moves for me (75-89)

Each of those planes is a 15-dimensional binary vector, indicating presence of each card, the numbers in parentheses mark the positions within the 90-dimensional vector. This coding is in machine learning usually called one-hot encoding.

On top of those 6 vectors there are 8 scalars:

- My won tricks, normalized by dividing with 5 to [0,1]
- Opponent won tricks, normalized to [0,1]
- Current trick number 1-5, normalized to [0,1]
- 1 bit: 0/1 – is it “me” leading the trick (e.g. playing the first card)
- 3 bits: leading suit A, B, C
- 1 bit: is there a leading suit

I tried also with smaller amounts of bits, as some of these are dependent on each other, for example opponent won count could be counted from my count and trick number. Also, the history, what me or opponent has played, could be seen irrelevant. But in principle that could be thought of a seed to contextual learning or heuristic: If we can infer the information about the suits the opponent does not have it can be helpful. That would be one of the first simple playing logics in the search tree, if this approach would allow programmed heuristics.

3.2.5 Network Architecture

The network design was not the focus of this work. The architecture was selected based on common practice for small board game domains combined with the CPU-only constraint, which required fast inference during MCTS search. The results later confirmed the choice was appropriate.

Input vector is 98-dimensional. It is projected to a 128-dimensional hidden representation, fully connected layer and batch normalization. The needed result is policy and value head, meaning probabilities per card in a vector of 15 values and a scalar value in range of [-1, +1]; +1 meaning certain win from this position (state) and -1 certain loss. Considered was that I had only a laptop and CPU, not a GPU. This network has about 130000 parameters and while testing and experimenting, it seemed a feasible task for a CPU. These decisions were also made with the idea in mind that if this approach works and network really learns, it could then be made even better with more training, larger network and a GPU. Main target was to see, if this approach works at all for hidden information games.

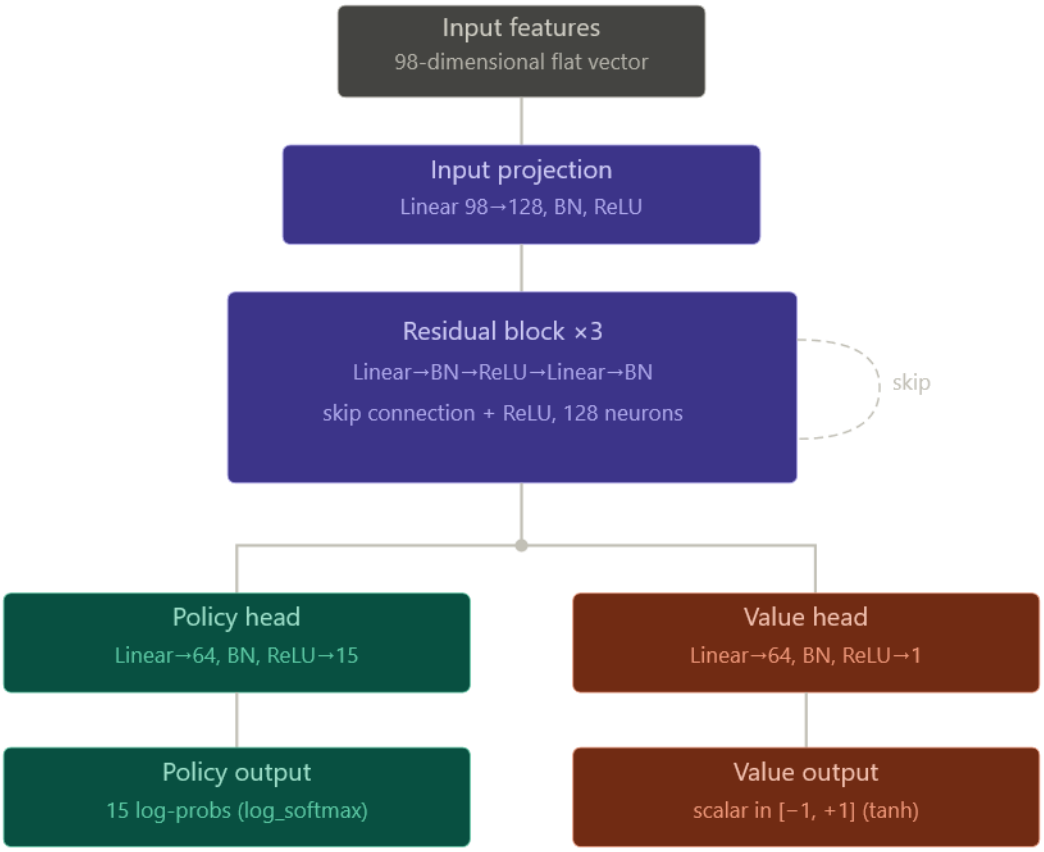

Figure 4. Neural Network Architecture

Loss function needs to have two terms: Policy loss L_π and Value loss L_v. Policy loss is here defined as cross entropy between the network given (predicted) card probabilities and the actual visit count percentages from MCTS search root. Value loss is very simple comparison between how the self-play game ended from the MCTS search root state z, (always +1 or -1) and predicted value v that is a range [-1, +1]. I used mean squared error (MSE) as loss function between those, $L_v = (z - v)^2$. The total loss is sum of those two losses, $L = L_\pi + L_v$. I decided not to give them different weights.

3.3 MCTS and PIMC implementation

3.3.1 The Imperfect Information Problem in Tree Search

Monte Carlo Tree Search (MCTS) is based on the requirement that game state is observable [7]. In a card game like Trick, this is not true: player cannot see what the opponent has in his hand, as all the deck cards are not dealt. Both opponent and undealt cards are unknown.

In MCTS, game is simulated. How to do that when it is not known how and with what cards the opponent may respond and play? This is handled in this experimental work by using Perfect Information Monte Carlo (PIMC). The search algorithm is not modified but the initial state is basically guessed. This guessing has another name, sampling, taking amount of opponent cards by sampling from the possible set of cards. Possible set is all the deck cards minus hands in player own hand and the ones that are already seen during the game. Then with this sample, MCTS uses it as it would be known and true, and so MCTS is able to search with perfect information. This is called determinization [12]. This is repeated over many times, so conducting the search by starting from different sampled states it is assumed that some uncertainty is removed. If the target if the experiment would be to get as good player that is possible in general, the sampling should be improved. This could be achieved by some heuristics for example, remembering what suits the opponent does not have – players must follow suit - and so on. But in AlphaZero, this kind of heuristic improvement is not allowed, at least if the heuristic would be given by the programmer.

One of the known and studied approaches would have been to utilize so called Information Set MCTS (ISMCTS). It improves the PIMC by evaluating game states in determinizations and deducing what states are equivalent from information point of view. But that is so called domain knowledge. AlphaZero approach requires that all information is gathered via self-play. MCTS itself has no game-specific information. In game state, it just gets the information from the “Coach” that which one wins the current trick and who plays next. And all the values for search nodes are coming from ML model, that was trained with self-play search and game result data.

3.3.2 Determinization

At each search root state, when a move should be selected, I decided to make 20 samples or determinizations from the possible unseen cards. Sampling is just taking random set, size of

set being the amount of opponent cards in hand. That number is known information. In the beginning the unknown amount is larger than when playing the last cards. The deck size 15 means that in the first sample there are 10 unknowns. If player A knows his own 5 cards, then from A's point of view, opponent can have C(10,5) = 252 different hands. 20 samples out of that are ~8% - that means 8% odds for having true opponent hand in the sample set. When opponent has only 2 cards in the end, the sample coverage becomes ~95%: at this point the unknow pool consists of 2 opponent cards and the 5 undealt ones, possible sets count being then C(7,2) = 21. 20 samples out of those being 20⁄21≈95%. However, in the end, it is less likely to affect the game result so much, as the scoring or winning in my setup is to calculate all the won tricks, not just the last one.

3.3.3 The Ensemble Search

Ensemble search is a method for combining search results from independent searches. In this experiment, from a given search state I was sampling 20 different determinizations, and for each determinization, I gave a search budget of 10 simulations. This selection of parameters was iterated over some different ones; the biggest factor was the compute time used for each search simulation. 20 searches (determinizations) are run side by side, and the search results are then voted as summed aggregates from all the results. Search result itself does not refer to opponent hand that is unknown. Result is only the probability vector over the cards in hand. Figure 5 shows how Ensemble MCTS works with determinizations and voting.

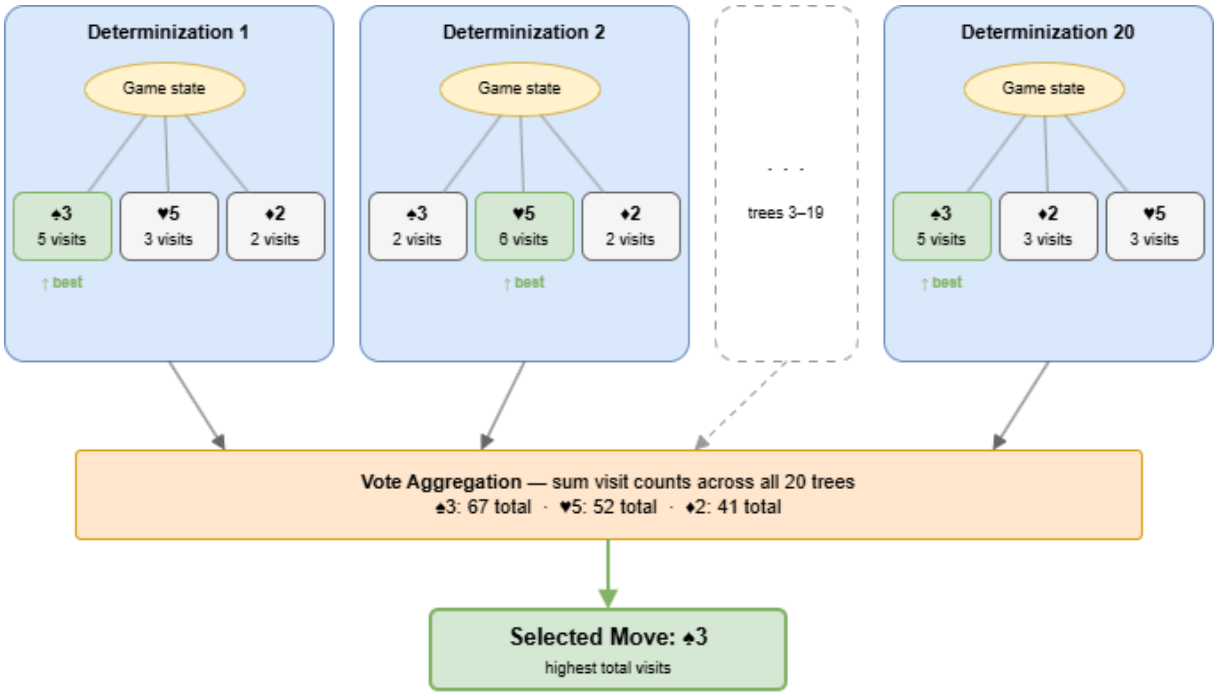

Figure 5. Ensemble MCTS and Voting

The overall search process is:

1. SAMPLE N=20 determinizations of opponent hands. Sample from same observable InfoState.
2. CREATE N MCTS Trees from sampled N GameStates
3. EXPAND all root nodes in one batch ML call
4. FOREACH simulation (total 200, 10 per tree)
    a. SELECT -> Walk from root to Leaf using PUCT
    b. EXPAND -> collect non-terminal leaves from all N trees, call ML inference with all at the same time
    c. BACKPROPAGATE -> each tree independently
5. AGGREGATE visit counts from all N trees to one. Only legal cards have visit counts, others are zero.
6. CREATE probability vector -> produce action probabilities over legal cards

7. SELECT move, if $\tau > 0$, sample one card proportionally to probability vector, if $\tau = 0$ select the one with highest probability.

Practical implementation detail in Ensemble search is the stepwise batching implementation. This follows from the notion that model inference takes basically the same time, independent of is it called with one input vector or 100 inputs. I implemented batched call mechanism to ML model – all determinization searches are running parallel and when one search would be instantiating a model call, the input vector is put into a batch "on hold". Only when all the searches are waiting for the model inference, the model predict is called with all the input vectors. This was actually a big improvement in the overall training and evaluation efficiency. In the self-play games, measured time for playing 100 games improved ~5x, from around 130 seconds to 25 seconds. AlphaZero uses decoupled, asynchronous training and evaluation pipeline. I did not start that kind of setup, as I only had one physical hardware. AlphaZero has parallel hardware for 3 purposes, NN weights for new iteration from recent self-play, evaluation of AlphaZero players from different iterations and self-play using the best model so far [6]. This is something different than mine, but the idea of separating the self-play and inference batches came from the AlphaZero article.

3.3.4 Single-Tree MCTS and the UCB Formula

In MCTS, a formula for balancing between exploitation and exploration was suggested by Kocsis and Szepesvári [23]. They called it UCT, as tree application of UCB standing from Upper Confidence Bound. The exploration-exploitation trade-off and the UCT formula are discussed in Section 2.9.3. From UCT there was later a further developed version PUCT, where P stands for Predictor, that tries to identify better arms in the multi-armed bandit problem [24].

The PUCT score for child selection at time t, or exactly selection of action a_t, a variant of PUCT suggested in [6], $a_t = \ argmax_a(Q(s_t, a) + U(s_t, a))$, where

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

and

- $U(s,a)$ = exploration "bonus"

- $Q(s_t, a)$ = mean value from visits through this node (sum value / visit count)
- $\sum_b N(s, b)$ = total visit count to the state s, counted recursively from its children visits $N(s, b)$ over each action b. Sum of all visits to one's children trivially equal to visits to itself.
- $P(s, a)$ = prior probability from NN models policy head
- $N(s, a)$ = visit count of the child of s when doing action a
- c_{puct} = 1.5 – an exploration parameter chosen empirically. Theoretical UCT constant being $\sqrt{2}$, but as PUCT formula is a bit modified, I tried something quite close to that and the exploration worked. As my simulation count was very small, I believe fine-tuning this would not have had a significant impact.

Key differences between PUCT from [24] and AlphaZero PUCT [6] are: Dropping of logarithm from inside the square root, in theory then exploration decays faster as the value of visited nodes get more weight. On the other hand, $P(s, a)$ scales the exploration bonus with the neural network model a priori probability – meaning that moving to network proposed directions is encouraged more than others.

3.3.5 Value Estimation: Neural Network vs. Random Rollout

The neural network returns a value between [-1; +1] from the current player's perspective, -1 being certain loss and +1 certain win. It is good to note that it is not a probability but a score. In the search, the value is converted to root players perspective in the backpropagation phase. In practice, this means that if the current node is other player than the root, the value is negated.

In my experiment setup there are 200 simulations per InfoState, and those are equally distributed to 20 determinizations, leaving 10 simulations per search. The game is 10 moves long: both players playing 5 cards. From the beginning in the first move, there are 14400 branches in the search tree. This follows simply from multiplication, both players having 5 choices, then 4, 3, 2 and 1: 5! x 5! = 14400. This means that the tree rarely gets to end state in the search, especially in the beginning of the game. Later, when players have fewer cards, the tree gets significantly smaller. I measured the average MCTS search depth after completing the actual training and evaluation. Using untrained network and 1000 synthetic deals, in the

beginning the search depth averages to 4.6, ranging from 3 to 8, in the mid-game 4.7 on average, ranging from 3 to 6, and only in the end the depth always reached the terminal state. For verification purposes I also did the same measurement for the best model and it showed the searches went deeper – essentially meaning that the trained model's policy head guides the search to promising direction.

Untrained network in the beginning is only noise, therefore MCTS does not really have any signal for the a priori probabilities in the node expansion. This is what I call here a "cold start" problem. For this I used a weight parameter for the self-play, where the value head was not used at all during the self-play, with value estimated from the random rollouts instead. It is important to distinguish between the depth of the search tree itself, which is limited by the simulation budget, from the depth of the rollout used to reach a leaf with a value. Whenever a leaf is expanded there is also a random rollout to the end, but that is fast and does not affect search depth. Command line parameter states the weight between the network value and the random rollout value. Policy prior from the network is used in the self-play from the beginning to guide the MCTS. In the evaluation phase I used value and policy heads, not rollouts. Both AlphaGo Zero [6] and AlphaZero [3] address this problem through very high simulation counts.

3.3.6 Temperature and Move Selection

During the self-play I used temperature variable $\tau = 1.0$ for first 6 moves and the last ones using temperature $\tau = 0.0$. Temperature 1 leads to exploration and 0 to exploitation. Value 1 samples an action with probability that is proportional to visit counts, hence preferring good looking ones but still giving possibility for others as well. 0 picks always the most visited action, that can be seen as deterministic and best possible known direction. This can also be seen as "Greedy" search with $\tau = 0.0$

In the Gating and Evaluation phases the temperature τ is always 0.0, meaning that there is no exploration, only a greedy search.

In the move selection, ML model gives probability vector over all 15 cards but only the legal cards in the end have non-zero probabilities. MCTS implements the removal of illegal ones and does not use those to mess up the search probabilities.

After my implementation I noticed a trivial optimization that I decided not to implement anymore once I had the results already. The search is rather stupid when player has only 1

card in his hand – he must play that but still the system plays those 200 simulations and does model inference. This means 20% of the self-play time was wasted. I did not change the implementation after the results because I thought the change would change the results a bit. Intuitively the results would be slightly better, because now in the training data there are 20% of the input vectors so that there is actually only one card in hand but model still gives probabilities over 15 cards (the deck size) as always. The loss function in ML training still works consistently, it pushes down the 14 values and rises that one. Basically these 20% of the rows where only 1 card in hand, dilute the training data with trivial rows. It does not teach the model anything wrong but certainly not anything of strategic value. One another trivial optimization is the situation where one of the players already has more than half of the possible tricks. He wins no matter what. This does not corrupt the training data, because the count of won tricks is in the data, it is easy for NN to learn that "if my won tricks >= 3, my value is 1.0 (win the game)". This dilution of training data with trivialities appears also in this case.

3.4 Training loop and gating

3.4.1 Phase 1 – Self-play

Self-play is used to create training data. In the self-play, the newest - not necessarily approved in gating - model plays against another similar one. Both players use MCTS to select moves, and in MCTS the model is used when expanding a new leaf to the search tree.

Self-play produces the following data: from each state the player encounters, what are the MCTS results e.g. possibilities for each move (card selection) and what was the outcome from the self-play game from that particular state. Outcome here means who won. The move probabilities are obtained by summing and averaging visit counts from determinization trees, as the ensemble MCTS approach states and combining the results.

3.4.2 Phase 2 – Training

In training phase I used a concept of "training window" that means the set of training data to be used in the model training. Generally, the idea is to use only the data from the self-play games that originate from the best model so far. When I started the tests, I noticed that generation of self-play data took most of the time instead of model training. Then I realized that I could use the same data as was used in the rejected model training if the data was

created with the best available model. My reasoning for the older data (but still after the last accepted model) not producing better results than the current best model is only because of the used decks and the random sampling in the search did not just create better games.

Self-play data is always generated by the current best gate-accepted model; rejection means the trained weights are discarded and the next iteration starts fresh from the same best model. Training uses the games that are played after previously set current best model with a window parameter, I used 10.

Example: if current best model is v100 and we're now on iteration 115, in the v115 training I am using training data from rounds 106-115. But if we're on iteration 105, I am using only the training data from rounds 101-105 because the data from round 100 was generated using the model before the v100 was found.

3.4.3 Phase 3 – checkpoint

After training the model, it's weights are stored to disk. Initially the stored model is marked as "not accepted". After the evaluation and gating in next phase, it can be marked as accepted.

The trained model, even if it is not accepted in the gating process, is kept for later evaluation purposes.

3.4.4 Phase 4 – Gating

I used gating method to evaluate the model goodness. Does candidate win the current best model? Winning rate is a gate parameter, I used 52% - if the candidate model wins more than that number of games against current best, it is then selected as current best. Evaluation games are always played with a new set of random decks than what were used in training, to avoid learning only the training data.

3.4.5 Phase 5 - Repeat

The previous phases are iterated until one of the stopping conditions is met. Either

- total time used is over the limit
- maximum number of iterations is reached
- learning results are plateauing too many rounds in a row.

Each of these conditions is given as a parameter. Training can be restarted from any given model generation over again with new parameters if that is needed. This happened when I found out that the initial training session took quite a lot of time – time condition was met - and I understood there would still be learning to be done.

The whole training loop is presented in Figure 6.

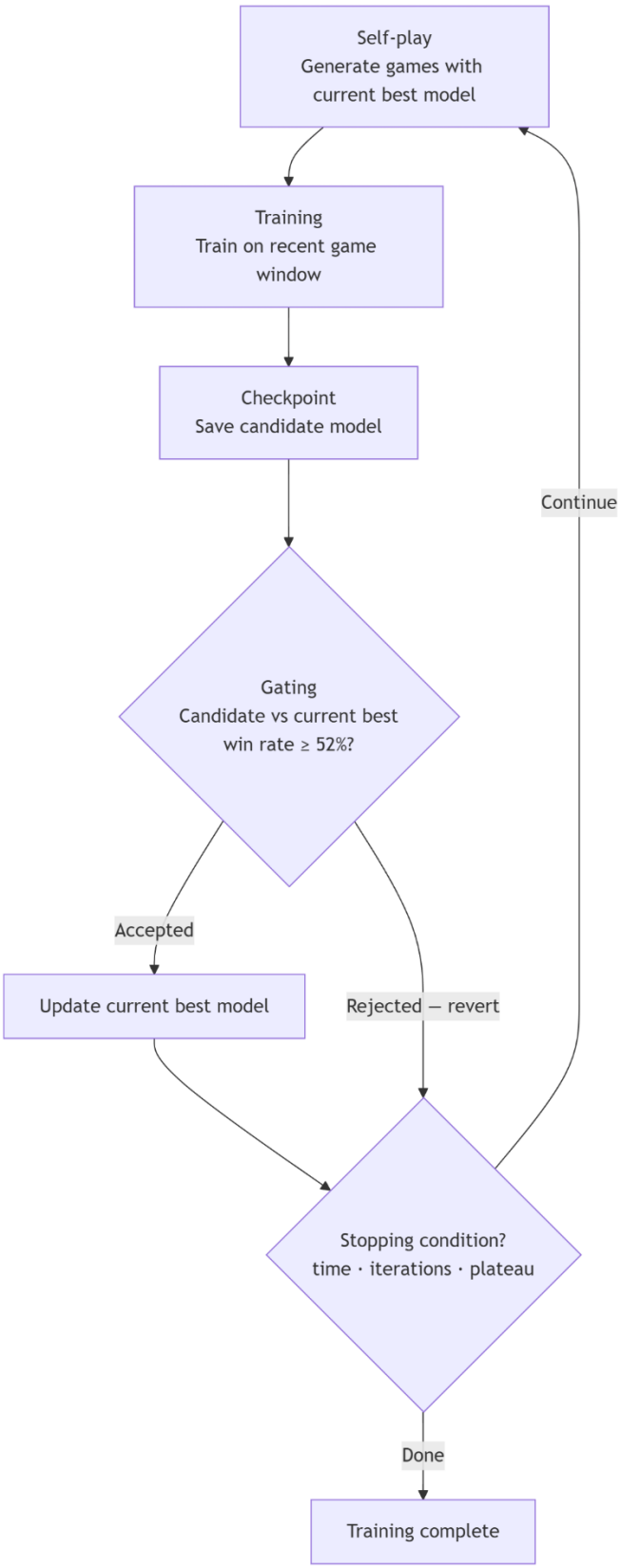

Figure 6. Training Loop Diagram

3.5 Cold Start problem and bootstrapping

When training starts in AlphaZero-style, the beginning is just noise, as described in Section 3.3.5. ML model does not have any capability to predict possible cards to play nor the outcome of the game (win possibility). With the quite small simulation count in MCTS the tree does not reach the terminal state – in practice the search depth reaches 3-8 levels in the early game, and the values come from noisy network. In the endgame with only 3 moves left and when there are less cards and search space is smaller, search narrows down to just 3 levels and the values also come from actual terminal states whose value is exact and does not require network. So instead of using the noisy value head in the self-play I had the self-play play randomly until the terminal leaves. But in all the evaluation, gating and head-to-head games I used only the values from the network. This is strictly not the same as in AlphaZero that just has more simulations, it uses 800 simulations per move over millions of games. The self-play games, however, are always played until the end, and in all the node expansions the network is called for the move prior probabilities. The policy prior is used from the start in every expansion even though it starts uninformative. Visit count distribution, which is the training target, is driven mainly by the outcomes of the random rollouts. This means that the policy head receives genuine training signal from the beginning, not just noise. The value head is bypassed and replaced by the real outcome from the simulation.

This is the bootstrapping problem or cold start problem – how can I utilize anything from the model in the self-play? The answer in this work was, as mentioned above, to use the policy head from the start but not the value head. This way I could with the computational budget I had, give to the model training at least some signal to work with. From the results, in Chapter 4, we can see that this approach worked.

3.6 Evaluation methodology

All the evaluation games in this study use a four-game-per-deal design to eliminate two sources of noise. One is luck in the card-dealing and the other is first-move advantage. First means that if one gets bad or good cards, it may be impossible to win or lose with those. The player who starts is considered to have an advantage in trick-taking games. To eliminate this source of noise, I created a mechanism that I am calling a four-game-per-deal mechanism. Simply put, one deal of cards is played four times. Players have no memory of the cards from

previous games in the same deal. Both players will play both hands from the deal, and with both positions, starting or following. Table 1 shows the combinations.

Table 1. Evaluation Game Combinations to Remove Luck

	Evaluation Player	Fixed Opponent	Player to start (lead)
Deal 1, Game 1	Hand A	Hand B	Player
Deal 1, Game 2	Hand A	Hand B	Opponent
Deal 1, Game 3	Hand B	Hand A	Player
Deal 1, Game 4	Hand B	Hand A	Opponent

I came up with this idea from the scoring format used in Bridge tournaments. They are organized in a way that many (or all) players play with all the pre-ordered decks against others. Pre-ordering of the cards is naturally hidden from players, but it is for the organisers to ensure that others can be dealt the same game setup. This way, even with bad hand one can get tournament points: if you are playing better than any other team with that bad hand, you are considered better players. This is called barometer scoring in Bridge. In my setup, each deal is played four times, and total amount of won games for a player is counted over all those. This design allows meaningful comparison between players even with small sample sizes – small sample size without this mechanism would be skewed by the random deals that in small numbers would not necessarily be varying enough.

This evaluation methodology is separate from the gating process described in Section 3.4, which compares candidate models only against the previous best model to decide acceptance. Evaluation against fixed baselines and reference players serves a different purpose: measuring absolute playing strength at selected checkpoints throughout training.

Five evaluation baselines were used throughout. Three of them were so-called fixed strategies. Random strategy player plays any legal card at random. Highest and Lowest strategy player plays always highest or lowest legal card, respectively. These were simple to understand and such that I wanted to beat any of these with the trained model so that I could claim it has learned to play.

Fourth, a Cheat mode, was implemented so that it plays with Neural Network and MCTS, but instead of sampling from hidden information, it is given full knowledge of opponent hand.

This way from each generation of evaluations, I could see the effect of hidden information, i.e. information barrier.

Fifth, a minimax solver with perfect information provides the absolute performance ceiling, the best achievable result. It has full information of opponent's hand, and it uses minimax-algorithm that is proven to find optimal game play in games of perfect information and full observability.

With these evaluation methods I could draw the goodness score of a model. Fixed strategies are clear comparisons, the percentage of won games against those is a basic measure that should get bigger over the generations. Then, using ML model with cheat mode against the fixed strategies, I could see the percentage of games won more because of having perfect information. And last, minimax against fixed strategies, the difference between cheat and minimax result can be seen as the theoretical possibility of learning more. Together these five reference points form the four-layer decomposition framework used to interpret all results in Chapter 4.

3.7 Key implementation lessons or MCTS bugs

Two perspective-tracking bugs caused significant issues during development and are worth documenting here as implementation lessons.

Node's player perspective is of big importance. In the game of Go and Chess in AlphaZero, players are always changing turns: player B plays always after A, then player A plays again. But in this trick-taking game the winner of previous trick gets to play next. $Q(s, a)$ is always from the point of view of that state s – this sounds trivial, but I implemented originally the tree in a way that every second node alternated the sign. So, if the player in the child node is different, the Q value is negated when summing the values up from the leaves towards the root. This was a key debugging experience. Basically, the wrong sign means that a child node rightfully being very good choice, becomes a very bad choice from the node's point of view. Originally the whole system did not seem to learn anything, and this changed the results. That is clear and self-evident but during the debugging it was not. This programming mistake practically broke the whole idea of MCTS being a policy improvement operator, because the search results were practically random or even finding the worst possible move. Only exception being those where the same player in the search simulation always continues to be leading the suit, and those are rare cases. The fix itself was trivial once the root cause was

identified, but unit tests alone would never have caught it. First, I was running the training for hours with no progress, then trying change of network architecture, input vector formatting, even smaller decks. This kind of bug passed through all the structural and automated unit tests, as it just did not come to my mind. The algorithm, from execution point of view, works just fine and that is why it was not caught in the tests. I had the idea that "more simulations should help", so I added a lot of simulations and games, with no effect at all or even going worse. The system did not beat even the fixed strategy random player. Eventually, this root cause crossed my mind and after that many things started to fall into place.

The bug persisted in the code a long time while I was concentrating on creating the technical training-evaluation-gating pipeline to be as automated as possible. For that automation I created a lot of unit tests, also for the game mechanics, I developed the batching of model inference in the search, as "poor man's parallelism". The problem showed itself only in the behavioural diagnostic – "does it learn?". And that was possible to see only after developing the automation so much that it was even possible to run longer trainings with many iterations and changing of the parameters. From software development aspect, this shows that optimisation and correctness are not aligned but orthogonal. A nicely running but incorrect system is still broken.

Perspective was also present in another bug, independent but related to the previous, that was found at the same time fixing the mentioned child sign-problem, it was in the expand and backpropagation: expanding the node returned the value from the expanded node's player perspective but backpropagation treated it always as the root player's point of view. Values were then credited to the wrong player. Interesting note from this is that the first bug affected both self-play and evaluation, while the latter one only affected evaluation, because self-play was not using value head in MCTS, as described earlier in the Section 3.5.

3.8 Component Architecture

Figure 7 shows the key components of the system and their relationships. The source code is organised into six packages — "game/", "model/", "mcts/", "training/", "evaluation/", and "storage/" - those being the directory names for Python packages. The most important components for understanding the system are the classes that cut across them. Five of these correspond directly to the conceptual modules introduced in Section 3.2.1: Evaluator is implemented as "Arena", Network as "NNetWrapper", MCTS as "EnsembleMCTS" and "MCTSearch", and Storage as "GameRepository". The Coach class in the 'training' package

acts as the top-level orchestrator, driving the iterative loop described in Section 3.4, and coordinating all other components. Coach is also responsible for gating: after each training iteration, it evaluates the candidate model head-to-head against the current best model, accepting the candidate if it passes the gating criteria, showing statistically meaningful improvement as discussed in Section 3.4.

The game engine provides the foundation for all other modules. It is implemented as a collection of pure functions: “get_legal_moves()”, “apply_move()”, “is_terminal()”, and “get_result()”. It is operating on immutable frozen dataclasses. No state is held between calls. This design is essential for MCTS, which must branch from the same game state many times without side effects; it also makes the engine straightforward to be tested.

The “NNetWrapper” component accepts an “InfoState” encoding and produces the two outputs described in Section 3.2: a policy distribution over the 15 possible cards and a scalar value estimate. It exposes both a single sample “predict()” and a batched “batch_predict()” interface; the batched interface is used by the ensemble search.

The MCTS layer contains two components. “EnsembleMCTS” implements the PIMC determinization approach described in Section 3.3: it samples twenty plausible opponent hands, constructs twenty independent search trees, and drives them in lockstep so that all leaf evaluations can be submitted to “NNetWrapper” in a single batch call. “MCTSearch” implements standard AlphaZero MCTS on a single fully observable game state. The lockstep design reduces approximately 200 individual network calls per move to roughly ten batched calls, yielding the approximately five-fold speedup reported in Section 3.3.3.

Within the training pipeline, “SelfPlayWorker” generates games by invoking “EnsembleMCTS” for every move and recording the resulting (state, policy, outcome) triples. The “Trainer” loads these samples from storage and updates the network weights.

The “Arena” component implements the fair four-game-per-deal evaluation described in Section 3.6, together with the three heuristic baseline players, cheat and the minimax solver. Arena selects moves through a common “pick_move()” interface implemented by each player type; for an MCTS-driven player this interface is provided by “MCTSPlayer”, which wraps “EnsembleMCTS” so the search can be evaluated identically to the heuristic baselines. This is also the mechanism Coach uses internally for the gating step described above. The “GameRepository” persists all self-play games, training samples, and checkpoint metadata to

a SQLite database, allowing training to be paused and resumed at any checkpoint without data loss.

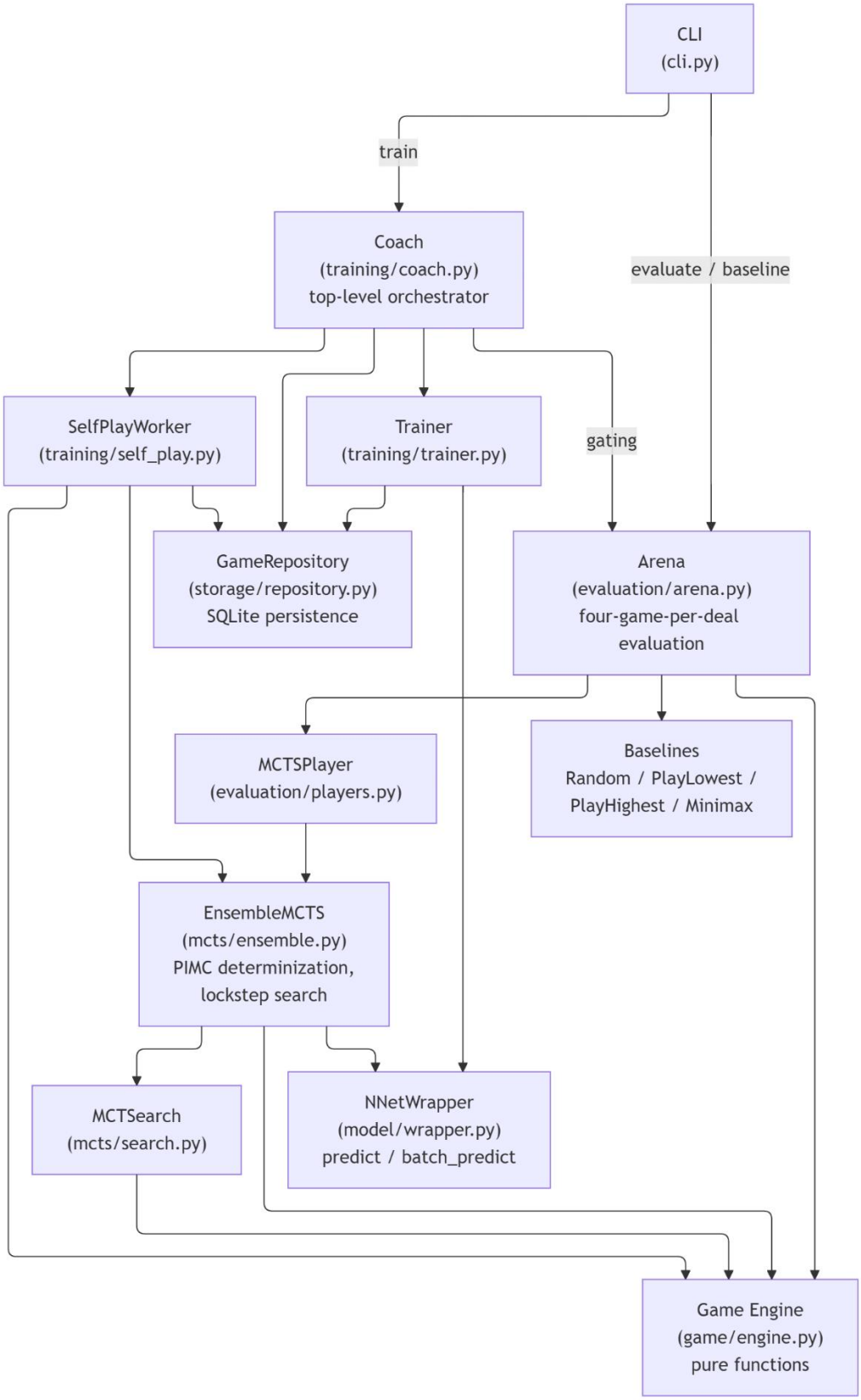

Figure 7. Software Component Architecture

4 Results

4.1 Overview of the results

The system learns but it suffers from what I call an “information barrier”. For evaluation purposes I have created a minimax perfect information version of the search. That version is giving player perfect information about opponent hand and doing minimax search. It is theoretical upper limit of how good a player could be as it finds the optimum. That limit is a good comparison point instead of comparing to 100% wins. Another evaluation strategy has been to create a cheat player, who knows opponent hand. Cheater does not do minimax but plays with MCTS and ML model. Difference between cheat and minimax is measuring the network quality: With perfect information, how good the network is compared to minimax. These are the key elements in my evaluation.

Minimax upper limit can also be used as a stopping condition for more training: when network plays almost as good as minimax (both with perfect information) – no further training brings a lot of quality to the network itself. So, when the network is this good, the remaining gap is what I call “information barrier” – caused from the fact that the system is guessing what opponent has. This part could be made better by creating heuristics or some Bayesian belief tracking but that was not how AlphaZero works. Trivial enhancements could be at least inferring information about opponent suits if he for example was not following suit in the previous round. That would make determinizations more accurate, because they would be sampled from smaller number of possibilities.

Training was conducted by playing against previous best model and if the new model won enough games, that was then selected as best model so far. During training and self-play iterations I did not play evaluation games against fixed strategies. Instead, I trained for about 10 hours and after that ran fixed strategy, cheat and minimax -evaluations with 250 deals (1000 games). My method was to play the same deal with all four possible player hands and starting player combinations. That is how I can remove the factor of possible first player advantage or factor of being dealt bad cards.

There were eventually 627 iterations with same training and evaluation parameters. Before that I ran trainings and evaluations with smaller datasets and parameters, to verify the system technically works and brings at least some small improvements. After that looked promising, I ran a couple of 8-hour training sessions, ending to 627 of which version 618 was the last

accepted model in gating. Of those, total of 172 models were accepted in the gate mechanism that was described in the Chapter 3. All the training that led into the end results, had 200 self-play games per iteration. Gating threshold was set to 52% and it was using 25 deals leading into 100 games as described in the Section 3.6. All the evaluation games were played with the four-games-per-deal mechanism described in that same section.

Results have been evaluated by 4 criteria:

- Learning progress over iterations (generations)
- Model goodness against fixed strategies (low, high, random)
- Model goodness with perfect information ("cheat mode") against fixed strategies
- Comparison against minimax, exhaustive search with perfect information, that would be absolute optimal player

From these I could establish the components of model goodness: network quality and information barrier, and how much there would still be possibilities to learn. Figure 8 shows the decomposition of learning and Figure 9 shows how learning evolved over generations.

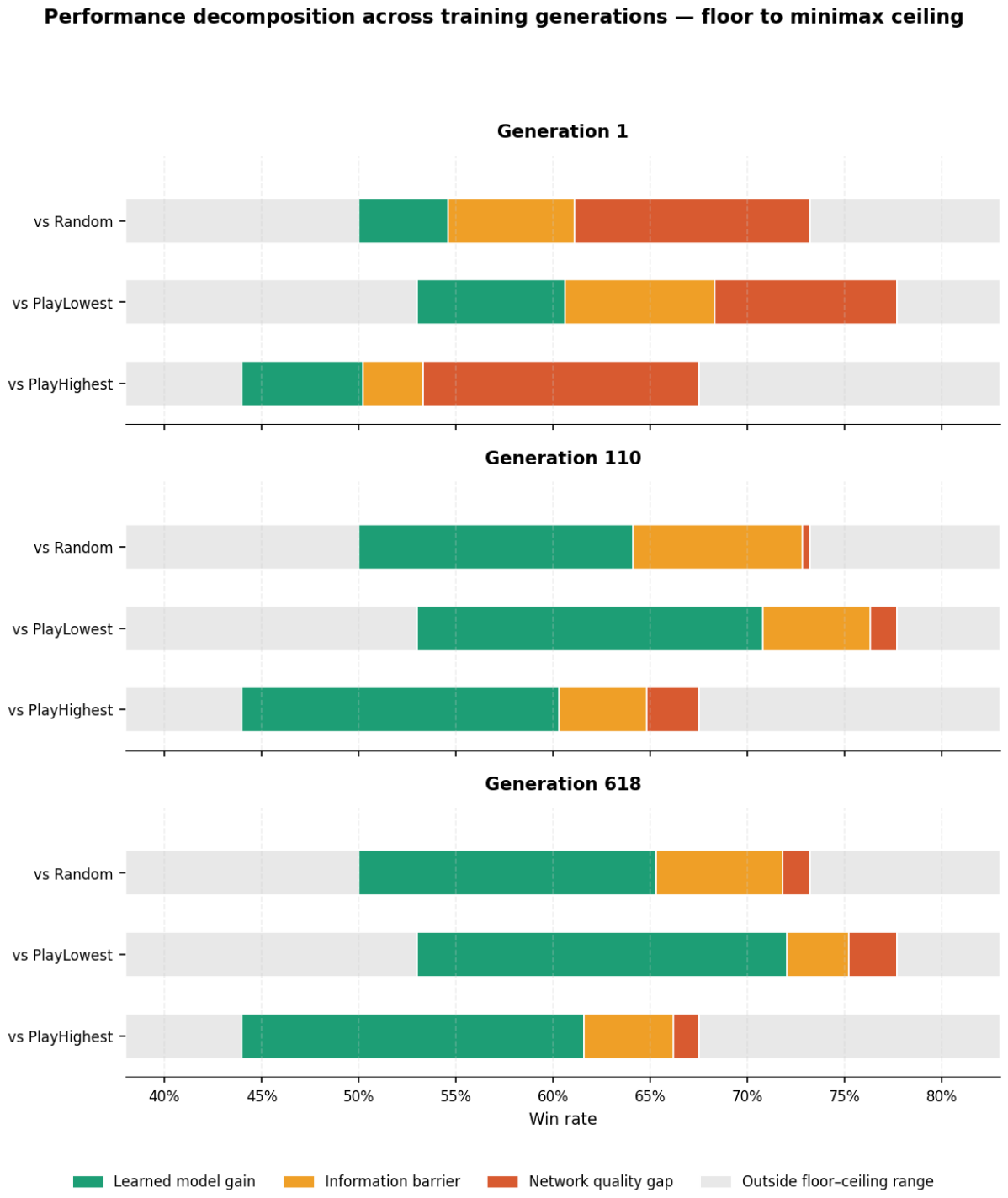

Figure 8. Performance Decomposition Over Generations

Same numbers are shown in a different format in Table 2. Random vs Random was not actually played, it is theoretically 50%, if two random player play enough games against each other, it converges to 50% like coin-flip.

Table 2. Four Layer Decomposition of Learning Results

Layer	vs Random	vs PlayLowest	vs PlayHighest	Average
Floor random baseline	50.0%	53.0%	44.0%	49.0%
Learned model	65.3%	72.0%	61.6%	66.3%
Learned + Cheat	71.8%	75.2%	66.2%	71.1%
Minimax ceiling	73.2%	77.7%	67.5%	72.8%

4.2 Baseline validation

First, I ran the fixed strategies against each other to get some baseline understanding and number values. These are based on 1000 deals, 4 games per deal as described in the fair evaluation method to remove first player advantage or the effect of better hands.

Table 3. Fixed Strategy Baselines

Player setup	Win rate (first mentioned player win %)
Random vs PlayLowest	~53%
PlayHighest vs Random	~56%
PlayHighest vs PlayLowest	~60%

Table 3 shows that always playing highest legal card brings the best outcome of these and always playing the lowest seems to be the worst strategy. Random legal moves playing player is somewhere in the middle. This is quite intuitive. As a minimum goal for my experimentation I wanted that the model and MCTS must win each of these fixed strategies. I also wanted the learned model to win “Random” and “Lowest” -strategies more than the “Highest” did. Eventually the best model won 65% against Random (baseline 56%) and 72% against Lowest (baseline 60%), so that criteria was clearly met.

4.3 Learning progression

Figure 9 shows how the system learns over iterations. Out of 172 gate-accepted models I ran batch evaluation for every 5th accepted model. I decided to do so because batch evaluation was time consuming. I wanted to play so many games that the effect of variance in the decks and in MCTS search would be rather small. Each batch evaluation consists of 6000 games. Model is put playing 250 deals in a fair setup, meaning 1000 games, all in normal and cheat mode against 3 fixed strategies (250 x 4 x 3 x 2 = 6000).

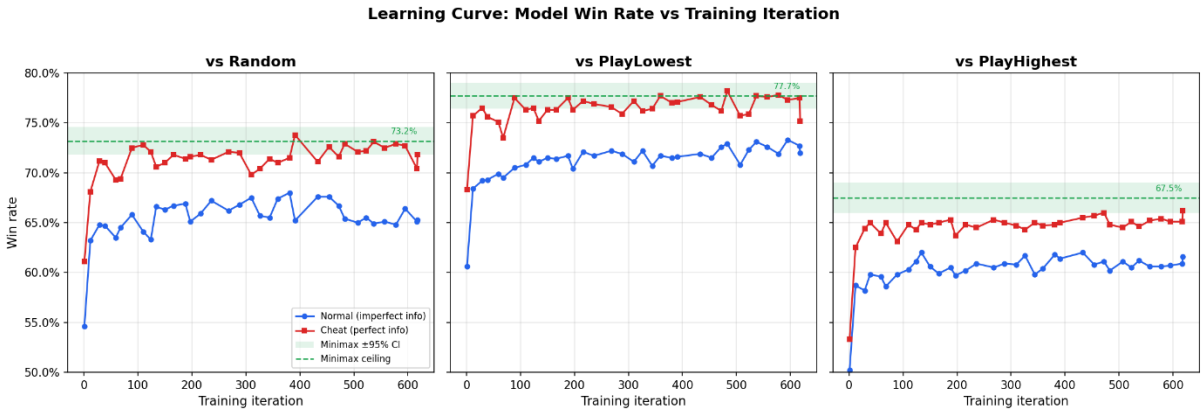

Figure 9. Model Performance Against Fixed Strategies

From each of the graphs can be seen that learning is rapid in the beginning, first 50-60 iterations and plateaus quite soon. Only the performance against “PlayLowest” seems to improve after 100 generations. The gap between red and blue line is the information barrier, and between minimax and red it’s the potential of learning more via self-play and training. The difference between red and blue lines remains quite unchanged, from this I deduce that the cost of hidden information stays the same, when the opponent stays the same too. In the random opponent case, the gap varies more and my assumption is that it is because of the random player just sometimes guessing better.

In a few points the red line goes over the possible optimum of the minimax. This is a statistical artifact because minimax against those strategies was calculated in separate run, only once, with 4000 games. To be exact, the minimax ceiling should have been calculated with the same exact deals than the ones used in the batch evaluation. That is why I have drawn with light green an area of ±1.5 percentage points of uncertainty to the minimax value.

4.4 Head-to-Head comparison

I did also head-to-head comparisons, so that the last accepted model (v618) was played against some of the earlier accepted versions. There I against used the fair-switch-places method and ran a quite large set of games, 500 deals and 2000 games per comparison pair. The results show that v618 wins but the margin is very small.

Key finding here is that v618 is basically not any better than v110. Win rate varies a bit up and down after v110 and I would consider it noise, ±1 percentage points. One thing to note is that plateau against ML models starts earlier than against fixed strategy, my thinking is that as

ML model uses MCTS and the model, it just plays better than some fixed strategy even in earlier generations. Table 4 shows the results from head-to-head comparisons.

Table 4. Head-to-Head Comparison, 2000 games

Opponent model generation	V618 win rate %	Note
V1	60.9	V1 is first model, random
V29	54.1	Rapid learning
V59	52.4	Slower learning
V110	50.5	Plateau starts
V236	50.9	
V391	50.6	
V507	50.3	
V581	51.2	
V618	50	Sanity check

Another thing to notice is that against PlayLowest and Random, the Cheat-version reaches or is very close to minimax ceiling, possibly meaning that against those (bad) strategies the model is already as good as it can get. Against PlayHighest there seems to be potential for learning still.

4.5 Information barrier

The approximately 5 percentage points gap between normal and cheat mode is stable across all opponents and all training stages. This consistency is strong evidence that the gap is structural rather than a learning failure — no amount of additional training can close it, because the cheat mode already plays near-optimally.

There is a strategy fusion problem related to PIMC, or any algorithm sampling possible worlds, like repeated minimax. In their article, Frank and Basin show that “The flaw (strategy fusion) in this approach occurs because of the property of incomplete information games that the exact state of the world at any given point of play may not be known to a player. This imposes a constraint on a player’s strategy that he must behave the same way in all possible worlds at such points; a constraint typically broken when combining strategies designed for individual worlds” [25].

There is a problem of averaging: At each move, the search samples 20 candidate opponent hands and runs independent MCTS trees on each, aggregating the results. Even if the correct

opponent hand is within the 20 samples, the agent cannot identify which one it is. All 20 determinizations are averaged, i.e. equal weight. If the optimal move is different between the true world (lucky sample) and the 19 incorrect samples, the incorrect worlds outvote the correct one when averaging them.

There is the problem of non-locality: In the same article, Frank and Basin state that "Such algorithms (e.g., minimaxing) will not take into account the possibility that under some worlds the play may never actually reach the node they are examining. When selecting moves, they may therefore make mistakes by erroneously considering payoffs in world states that are in fact of no consequence at that position in the tree." A bit differently said, it means that game history may affect the search space. In a perfect information game, the best strategy is independent of the history [8].

The gap, what I'm here calling the information barrier, is the cost of these limitations in this specific game. My assumption is that changing the deck size would affect this gap and the "cheat" mode would eventually still get closer to the optimal minimax. I also assume that other methods would be needed to learn the heuristics of the game, for example from the fact that opponent cannot follow a suit the unknown set of cards gets considerably smaller.

4.6 Computational resources

Training to v618 required approximately 700 iterations at ~80 seconds each, totalling roughly 16 hours on a standard laptop CPU. No GPU was required — the network is small enough (~130K parameters) that CPU inference is fast. Dominant cost is the MCTS tree search rather than neural network computation. Batch evaluation of 36 accepted model checkpoints added 5 hours, bringing the total project compute to 21 hours.

5 Discussion

5.1 Did the approach work?

Short answer is yes. From the aspect of gating – the model kept getting better, and from large amount of evaluation games between the last accepted model (v618) and the older accepted models we can see the improvement. The gating used less evaluation games so there may have been random variance during the gating and training. From the aspect of comparing against the fixed strategies versus the cheat and minimax baselines, it was quite clearly seen that the model goodness became very close to optimal, only the barrier caused by hidden information made the biggest difference. In numbers, model learnt 73% of the possible amount of learning. Of the remaining 27%, 20% was due to information barrier and only 7% was the actual amount of unused learning potential.

5.2 The information barrier and strategy fusion

This became a key finding in this work. With this approach, the model did not learn to deduce the hidden information, but it learnt to play. Approximately 20% of all the possible learning is the cost of hidden information in this game setup. A different deck setup would likely produce a different number. Possible continuation of the study would be to parametrise the deck setup, the number of hidden cards, to see whether the effect is linear or something else.

Strategy fusion is a mechanism behind the barrier, and this work measured its cost. The cost was circa 5 percentage points in winning ratio across all opponents and training stages. PIMC without other improvements always suffers from that. Measuring it was possible because the game search space is small enough for exhaustive minimax. Because the strategy fusion is inherent to PIMC-based approaches, this cost is not a failure but a measured structural property of the used method.

5.3 Network quality: what the model learned

The gap between the cheat mode and minimax averages approximately 1.7 percentage points in the game win ratio. This is the network quality gap, what a perfect-information version of the same system could still gain with further training. As the information barrier looks quite constant, my interpretation is that this same amount is also the possible maximum gain in the hidden information version as well.

This means that a network of 130000 parameters, trained with a laptop CPU and no GPU available, reached near-optimal play within hours, not days. Based on the results, changes to the network architecture would not close the network quality gap significantly. Near-optimality here refers to the amount there was possible to learn, considering the information barrier. This is a positive finding; system was hitting the ceiling that is based on the game characteristic.

One methodological note is that the minimax ceiling values were computed in a separate evaluation run rather than on the identical deal sets used for batch evaluation. This introduces a small amount of statistical uncertainty in Figure 9. However, minimax baseline calculation used the highest number of games, 4000 games. The core findings are not affected by this uncertainty, as the information barrier of approximately five percentage points comfortably exceeds it.

5.4 Limitations and future directions

Underlying assumption in the results evaluation is that strategy strength is transitive: floor, learned, cheat, minimax are in order from weakest to strongest. This is not always true in games, for example in rock-paper-scissors the strength is cyclic, not linear. This property was not tested nor considered. With the fixed baseline strategies, the measured strength between those showed to be linear. Same observation was seen in the head-to-head model comparisons. These tests do not prove, however, that the strength model in this specific game is linear. Whether linearity assumption would hold with wider set of testing is left open.

PIMC samples opponent hands uniformly at random. A smarter approach would weight those samples by what the play history makes plausible. If the opponent did not follow suit in round 2, many hypothetical hands become unlikely. This is Bayesian belief tracking. It would make determinizations more accurate without requiring domain heuristics baked in by the programmer. Improving the system would be easy with heuristics, but that was not the question in this study. One possibility, in the spirit of AlphaZero, would be to try to learn also a heuristic, tabula rasa. That would in my opinion require the game move history to be given to the model as additional input. This would touch the strategy fusion as well, as the history of the game would be a part of the decision, not only the current state.

Counterfactual Regret Minimisation (CFR) is the game-theoretic alternative. DeepStack [26] and Pluribus [27] are significant results. Unlike PIMC, CFR reasons about information sets

directly rather than determinizing. DeepStack and Pluribus also learn about the opponent weaknesses and bluffing. It is a fundamentally different approach, more principled for hidden information games, but also more complex to implement and less compatible with the tabula rasa self-play philosophy of AlphaZero.

A third direction would be to apply ReBeL [28]. It addresses a challenge in games of imperfect information: “the value of an action may depend on the probability it is chosen.” ReBeL specifically accounts for this.

All three directions would be natural extensions of my work.

5.5 On the choice of game size

The small game was a deliberate choice, not a limitation. It enabled exhaustive minimax, which enabled the four-layer decomposition, which enabled me to measure the information barrier. A larger game would have produced a system I could not have evaluated so well.

5.6 When to stop training

Head-to-head results show that v618 is statistically indistinguishable from v110. All plateau models produced margins that, with 2000 games against the best model, fall within statistical insignificance. But still, gating kept accepting models for hundreds more iterations — roughly 75% of compute time produced no meaningful gain.

For the gating to approve models, it could be wise to have stricter criteria. From the Binomial probability formula, it can be calculated that if two compared models in reality are equal, and the odds for winning would be 50 / 50, the probability of winning more than 52% - the gating threshold - of the games, when there are 100 games played, is quite high, 38%. This means that in 38% of the gating tests, an equally good model is accepted. Approving an equal model as the new best model is not too bad, but approving a worse model can be – at least it slows down the overall progress of the whole training pipeline. Using 100 games and 52% threshold has 24% probability of approving a model that has 48%-win probability. Using 500 games and 55% as win threshold would reduce probability of approving a worse model to 0.1%. However, stricter gating would be slowing down the pipeline and if the gate is too strict, it would not approve small but real improvements. This is a case of balancing and in this thesis this criterion was chosen, other thresholds could also be defended. It is worth noting that in

the evaluation phase I used 1000 games against fixed strategies and 2000 games in the head-to-head-comparison, so the results are reliable.

When to stop is itself an exploration-exploitation trade-off. Knowing when to stop is only possible here because I have a known ceiling — the minimax gives the reference point. In larger games without exhaustive search, I would see the curve flattening but could not know how far I am from optimal. The small game size enabled meaningful stopping criteria, not only evaluation material.

5.7 Broader applicability

A survey lists several interesting application possibilities for MCTS beyond games and discusses where the focus of theoretical work in this area should be directed [16, Ch. 7]. These align closely with the general application areas of reinforcement learning. I find particularly interesting among them, automated planning, for which international competitions exist. In these, plain MCTS does not perform well, but Bayesian Reinforcement Learning combined with MCTS, which enables risk-sensitive Bayes-adaptive policies that optimally trade off exploration, exploitation, and robustness [29], appears a promising direction. Planning is an application area with direct industrial relevance. Scheduling, which is closely related to planning, is another potential application. Some combinatorial practical problems may also be suitable — the article mentions automated architectural space planning: "automated design of generating large scale floor plans with adjacency constraints".

A quite different domain, but one that maps well onto a search problem, is Security Games. In those the task is to optimize the patrol routes of security personnel (police, guards, etc.) to prevent terrorism, smuggling, and similar threats. On this point the survey somewhat critiques the underlying assumption of perfectly rational actors, which arguably does not always hold for humans. Structurally similar is vehicle routing, which combines the classical traveling salesman problem with the knapsack problem — the former being minimization of travel time (NP), the latter (also NP) maximizing the value of goods carried subject to a capacity constraint. Both have well-known algorithmic solutions individually, but when the two problems must be solved simultaneously, MCTS has been explored as a tool. It is worth noting that in these problems a good enough solution often suffices. Travelling 1002 kilometres instead of 1000 or achieving €9,995 in value instead of €10,000 is likely to be acceptable. When optimality is not strictly required, the exploratory nature of MCTS can produce results that are good enough in practice.

An interesting application of MCTS can be found in chemistry, where existing knowledge, for instance of organic chemistry reactions, is used to algorithmically discover new reactions. The search space there is apparently so large that laboratory experimentation alone is not a feasible approach within any reasonable timeframe.

6 Conclusions

This thesis studied the AlphaZero method and its theoretical background. The thesis investigated whether AlphaZero-style self-play reinforcement learning can be applied to a two-player trick-taking game with imperfect information. Short answer is ‘yes’. The AlphaZero method was modified to handle hidden information and applied to a game of hidden information. The results separate into four measurable layers: random baseline, learned model, information barrier, minimax ceiling. The information barrier measurement is one of the key findings and a result that I did not expect beforehand to be so clear. Another result is that despite the hidden information, the measured 73% learning progress shows that the system learns meaningfully, especially when considering that the measured information barrier accounts for most of the remaining 27%.

It would require further study with more computational capacity to verify the result in a game with more hidden information and larger search space. Especially interesting it would be to measure the found information barrier as function of the search space size and amount of hidden information – is it linear or something else. To measure this, the experimental system can play same game with changes in parameters, and the infrastructure already exists.

A methodological takeaway is that the four-layer decomposition of learning may transfer to other decision problems beyond games, where imperfect information is often the de facto assumption.

References

[1] S. J. Russell *et al.*, *Artificial intelligence: a modern approach*, Fourth edition, Global edition. in Pearson series in artificial intelligence. Harlow: Pearson, 2022.

[2] "Heuristic," *Wikipedia*. Feb. 28, 2024. Accessed: Mar. 25, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Heuristic&oldid=1210861290

[3] D. Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, pp. 1140–1144, Dec. 2018, doi: 10.1126/science.aar6404.

[4] J. Goodman, "Re-determinizing Information Set Monte Carlo Tree Search in Hanabi," May 14, 2019, *arXiv*: arXiv:1902.06075. Accessed: Mar. 04, 2024. [Online]. Available: http://arxiv.org/abs/1902.06075

[5] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: 10.1038/nature16961.

[6] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017, doi: 10.1038/nature24270.

[7] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information Set Monte Carlo Tree Search," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 2, pp. 120–143, Jun. 2012, doi: 10.1109/TCIAIG.2012.2200894.

[8] D. Whitehouse, "Monte Carlo Tree Search for games with Hidden Information and Uncertainty," phd, University of York, 2014. Accessed: Mar. 04, 2024. [Online]. Available: https://etheses.whiterose.ac.uk/8117/

[9] M. Świechowski and T. Tajmajer, "A Practical Solution to Handling Randomness and Imperfect Information in Monte Carlo Tree Search," presented at the 16th Conference on Computer Science and Intelligence Systems, Sep. 2021, pp. 101–110. doi: 10.15439/2021F3.

[10] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Computers and Games*, vol. 4630, H. J. Van Den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., in Lecture Notes in Computer Science, vol. 4630. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83. doi: 10.1007/978-3-540-75538-8_7.

[11] C. Solinas, D. Rebstock, and M. Buro, "Improving Search with Supervised Learning in Trick-Based Card Games," *AAAI*, vol. 33, no. 01, pp. 1158–1165, Jul. 2019, doi: 10.1609/aaai.v33i01.33011158.

[12] J. Long, N. Sturtevant, M. Buro, and T. Furtak, "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search," *AAAI*, vol. 24, no. 1, pp. 134–140, Jul. 2010, doi: 10.1609/aaai.v24i1.7562.

[13] M. Buro, J. R. Long, T. Furtak, and N. Sturtevant, "Improving State Evaluation, Inference, and Search in Trick-Based Card Games," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, Jan. 2009, pp. 1407–1413.

[14] J. Schumacher and M. Pleines, “Improving Bidding and Playing Strategies in the Trick-Taking game Wizard using Deep Q-Networks,” in *2022 IEEE Conference on Games (CoG)*, Beijing, China: IEEE, Aug. 2022, pp. 307–314. doi: 10.1109/CoG51982.2022.9893614.

[15] J. Niklaus, M. Alberti, R. Ingold, M. Stolze, and T. Koller, “Challenging Human Supremacy: Evaluating Monte Carlo Tree Search and Deep Learning for the Trick Taking Card Game Jass,” in *Artificial Intelligence and Soft Computing*, vol. 12416, L. Rutkowski, R. Scherer, M. Korytkowski, W. Pedrycz, R. Tadeusiewicz, and J. M. Zurada, Eds., in Lecture Notes in Computer Science, vol. 12416. , Cham: Springer International Publishing, 2020, pp. 505–517. doi: 10.1007/978-3-030-61534-5_45.

[16] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, “Monte Carlo Tree Search: A Review of Recent Modifications and Applications,” *Artif Intell Rev*, vol. 56, no. 3, pp. 2497–2562, Mar. 2023, doi: 10.1007/s10462-022-10228-y.

[17] J. Obenaus, “Implementing a Doppelkopf Card Game Playing AI Using Neural Networks,” Bachelor’s thesis, Freie Universität Berlin, Berlin, 2017.

[18] R. Bjarnason, A. Fern, and P. Tadepalli, “Lower Bounding Klondike Solitaire with Monte-Carlo Planning,” *ICAPS*, vol. 19, pp. 26–33, Oct. 2009, doi: 10.1609/icaps.v19i1.13363.

[19] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1989, doi: 10.1016/0893-6080(89)90020-8.

[20] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*. in Adaptive computation and machine learning. Cambridge, Mass: MIT Press, 1998.

[21] M. Minsky, “Steps toward Artificial Intelligence,” *Proc. IRE*, vol. 49, no. 1, pp. 8–30, Jan. 1961, doi: 10.1109/JRPROC.1961.287775.

[22] “Markov property,” *Wikipedia*. Mar. 07, 2026. Accessed: May 14, 2026. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Markov_property&oldid=1342208169

[23] L. Kocsis and C. Szepesvári, “Bandit Based Monte-Carlo Planning,” in *Machine Learning: ECML 2006*, vol. 4212, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., in Lecture Notes in Computer Science, vol. 4212. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. doi: 10.1007/11871842_29.

[24] C. D. Rosin, “Multi-armed bandits with episode context,” *Ann Math Artif Intell*, vol. 61, no. 3, pp. 203–230, Mar. 2011, doi: 10.1007/s10472-011-9258-6.

[25] I. Frank and D. Basin, “Search in games with incomplete information: a case study using Bridge card play,” *Artificial Intelligence*, vol. 100, no. 1, pp. 87–123, Apr. 1998, doi: 10.1016/S0004-3702(97)00082-9.

[26] M. Moravčík *et al.*, “DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker,” *Science*, vol. 356, no. 6337, pp. 508–513, May 2017, doi: 10.1126/science.aam6960.

[27] N. Brown and T. Sandholm, “Superhuman AI for multiplayer poker,” *Science*, vol. 365, no. 6456, pp. 885–890, Aug. 2019, doi: 10.1126/science.aay2400.

[28] N. Brown, A. Bakhtin, A. Lerer, and Q. Gong, “Combining Deep Reinforcement Learning and Search for Imperfect-Information Games,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2020, pp. 17057–17069. Accessed: May 23, 2026. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/hash/c61f571dbd2fb949d3fe5ae1608dd48b-Abstract.html

[29] A. Sharma, J. Harrison, M. Tsao, and M. Pavone, “Robust and Adaptive Planning under Model Uncertainty,” *ICAPS*, vol. 29, no. 1, pp. 410–418, Jul. 2019, doi: 10.1609/icaps.v29i1.3505.