A Self-Trained Chess Engine for Crazyhouse Chess

Bachelor's thesis in Computer Science and Engineering

Martin Bratt
Axel Carlberg
Kevin Long
William Thompson
Hugo Westberg
Johan Zander

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY AND GOTHENBURG UNIVERSITY
Gothenburg, Sweden 2026
www.chalmers.se

Bachelor's Thesis 2026

A Self-Trained Chess Engine for Crazyhouse Chess

Martin Bratt
Axel Carlberg
Kevin Long
William Thompson
Hugo Westberg
Johan Zander

GÖTEBORGS UNIVERSITET

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg University
Gothenburg, Sweden 2026

A Self-Trained Chess Engine for Crazyhouse Chess
Martin Bratt Axel Carlberg Kevin Long William Thompson Hugo Westberg Johan Zander

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Co-Examiner: Muhammad Mustafa Hassan, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering

Bachelor's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gotenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Abstract

This thesis investigates the development of a self-trained chess engine for the Crazyhouse chess variant, inspired by the AlphaZero framework. Crazyhouse extends classical chess with a drop mechanic, where captured pieces can be collected and reintroduced onto the board, fundamentally altering strategic evaluation and significantly increasing the branching factor compared to classical chess. The absence of established opening theory makes Crazyhouse a particularly suitable domain for self-learning approaches, as chess engines cannot rely on accumulated human knowledge and must instead discover effective strategies entirely through self-play.

The engine was implemented using a C-based game engine for efficient move generation and rule enforcement, combined with a Python-based Monte Carlo Tree Search (MCTS) algorithm guided by a neural network. Four distinct convolutional neural network architectures were designed and compared, a shallow non-residual baseline, a shallow residual network, a deep residual network and a wide shallow residual network. All architecture were trained through an iterative self-play process and evaluated using a round-robin tournament.

The results showed that residual architectures consistently outperformed the non-residual baseline, confirming that skip connections provide significant benefits for gradient flow and learning stability in the context of Crazyhouse position evaluation. Among the residual architectures, the shallow residual network, using 128 channels and 6 residual blocks, demonstrated the strongest learning trajectory, having a more upwards going regression trend compared to the other neural network models. The deep residual architecture underperformed relative to its theoretical capacity, which suggests that the project's computational and time constraints resulted in insufficient training data. While all trained architectures demonstrated clear strategic intent beyond random play, none were able to win consistently against Stockfish at any tested skill level, indicating that the scale of training required to reach competitive strength exceeds what was achievable within a single academic semester.

Keywords: Crazyhouse Chess, Machine Learning, Monte Carlo Tree Search (MCTS), Reinforcement Learning, Convolutional Neural Networks, Residual Networks, AlphaZero, Chess Engine, Neural Network Architecture.

Sammandrag

Denna rapport undersöker utvecklingen av en självtränad schackmotor för schackvarianten Crazyhouse, inspirerad av AlphaZero-ramverket. Crazyhouse utökar klassisk schack med en placeringsmekanik där tillfångatagna pjäser kan samlas och sedan återplaceras på brädet vilket fundamentalt förändrar den strategiska utvärderingen av pjäser och ökar förgreningsfaktorn avsevärt jämfört med klassisk schack. Bristen av etablerad öppningsteori gör Crazyhouse till ett särskilt lämpligt domän för självlärande arbeten, eftersom schack motorer inte kan förlita sig på ackumulerad mänsklig kunskap utan istället måste upptäcka effektiva strategier helt genom självspel.

Motorn implementerades med hjälp av en C-baserad spelmotor för effektiv dragsgenerering och regelhantering, kombinerat med ett Python-baserat Monte Carlo Tree Search (MCTS) algoritm styrt av ett neuralt nätverk. Fyra distinkta konvolutionella neurala nätverksarkitekturer designades och jämfördes, en grund icke-residual baslinje, ett grunt residualt nätverk, ett djupt residualt nätverk och ett brett grunt residualt nätverk. Alla arkitekturer tränades genom en iterativ självspelsprocess och utvärderades med hjälp av round robin-turneringar.

Resultaten som framkom visar att residuala arkitekturer konsekvent presterade bättre än den icke-residuala baslinjen, vilket bekräftar att genvägsförbindelser ger betydelsefulla fördelar för gradientflöde och träningsstabilitet i kontexten av Crazyhouse-positionsutvärdering. Bland de residuala arkitekturerna visade det sig att det grunda residuala nätverket, med 128 kanaler och 6 residuala block, fick den starkaste inlärningsutvecklingen och visade på större utvecklingspotential genom att ha en mer uppåtgående regressionstrend jämfört med övriga arkitekturerna i de slutliga utvärderingsrundorna. Den djupa residuala arkitekturen presterade sämre i förhållande till sin teoretiska kapacitet, vilket hävdar att projektets beräknings- och tidsmässiga begränsningar resulterade i otillräcklig träningsdata. Även om alla tränade arkitekturer visade tydlig strategisk avsikt bortom slumpmässigt spelande, lyckades ingen vinna mot Stockfish på någon testad styrkenivå, vilket indikerar att den träningsskala som krävs för att nå konkurrensmässig styrka överstiger vad som uppnåeligt inom en enda akademisk termin.

Nyckelord: Crazyhouse-schack, Maskininlärning, Monte Carlo Tree Search (MCTS), Förstärkningsinlärning, Konvolutionella Neurala Nätverk, Residuala Nätverk, AlphaZero, Schackmotor, Neuralt Nätverk Arkitektur.

Acknowledgements

We would like to extend our gratitude and thanks to our supervisor Andreas Abel, for his guidance and professional insights in the work that has been done. Without Andreas tips and feedback of our training and evaluation results, in particular, we would not have had the chance to correct our mistakes which would have been detrimental for the success of our research.

Martin Bratt, Axel Carlberg, Kevin Long, William Thompson, Hugo Westberg, Johan Zander. Gothenburg, May 2026.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Chess has served as a fundamental playground for groundbreaking research in artificial intelligence since the beginning of the 20th century. Early chess engines relied on brute-force search algorithms combined with hand-crafted evaluation functions designed by the very best players and experts in chess [20]. A major milestone for these algorithms was reached in 1997, when IBM's Deep Blue defeated the reigning world champion, Garry Kasparov. This marked the first time a computer system surpassed human capability at the highest level of chess [14]. However, Deep Blue still relied heavily on human expertise. Chess professionals provided the evaluation heuristics, and the system's strategic understanding was largely based on this human knowledge.

For decades, the dominant approach in chess engine development remained largely unchanged. Historically, engines such as Stockfish reached superhuman strength through highly sophisticated, hand-tuned evaluation functions. They also relied on extensive opening-move databases. Together, these components represent accumulated human chess knowledge [20]. While Stockfish and similar engines achieved ratings above 3600 ELO, far exceeding the human peak of around 2900 ELO, they primarily functioned as amplifiers of human strategic understanding rather than as independent learners of the game.

This blueprint for the development of chess engines shifted dramatically in 2017, when DeepMind introduced the chess engine AlphaZero, the first engine to learn exclusively through self-play without any human-provided domain knowledge beyond the basic rules of classical chess [22, 21]. After only 9 hours of self-training using 5,000 first-generation tensor processing units, AlphaZero defeated Stockfish, demonstrating that machine learning approaches could surpass decades of accumulated human expertise [8]. The system, developed by DeepMind, employed a neural network trained through self-play using the search algorithm Monte Carlo Tree Search (MCTS), and improved its evaluation purely through its own experience [11]. AlphaZero's games exhibited a distinctive playing style that sometimes violated established principles, suggesting that it had discovered new strategic con-

cepts independent of human theory [21].

The significance of this breakthrough from AlphaZero extended beyond impressive playing strength. It demonstrated that self-learning can discover effective strategies without requiring any human expertise, that neural network evaluation can surpass hand-crafted heuristics, and that the same approach could master multiple games without game-specific modifications [22]. These findings suggest that self-learning approaches may be particularly valuable for strategic games where human expertise is limited.

The shift toward neural network evaluation was further solidified in the broader engine community following AlphaZero's breakthrough. In August 2020, Stockfish introduced NNUE (Efficiently Updatable Neural Networks), a specialized architecture that allowed the engine to benefit from neural network evaluation while maintaining high-speed search on standard CPUs [23]. While initially a hybrid system, the transition to machine learning was completed in July 2023 with the release of Stockfish 16, which fully removed the remaining hand-crafted evaluation code [25]. This evolution signifies that the best modern engines have moved away from human heuristics in favor of purely learned evaluation functions.

This creates a great opportunity in game variants of classical chess, with modified rules that alter fundamental strategic considerations. While classical chess benefits from centuries of theoretical development, chess variants largely lack this depth of human expertise. The chess variant Crazyhouse, inspired by Japanese Shogi, presents a particularly interesting case study. In this variant, captured pieces change color and can be put back onto the board as a move by the capturing player. This rule completely transforms the strategic landscape, with material evaluation becoming ambiguous since captured pieces strengthen the opponent, and traditional piece values and positional principles require reconsideration. The lack of an established theory means rule-based engines must rely on educated guesses rather than proven principles.

The academic relevance of this project lies in investigating whether self-learning approaches can effectively develop strategic understanding in settings where human expertise provides no guidance. From a technical standpoint, this involves challenges in neural network architecture design, efficient MCTS implementation for games with large branching factors, and the development of methods for extracting learned strategic knowledge. Successful outcomes could provide indications that self-learning artificial intelligence systems may be capable of developing expertise in new strategic problems without extensive human input, potentially extending to domains where expert knowledge is scarce or expensive to acquire.

1.2 Purpose

The primary purpose of this thesis is to design, implement, and evaluate a self-trained chess engine capable of playing Crazyhouse chess-variant at a high and

competent level. The project aims to demonstrate that self-learning approaches inspired by AlphaZero can successfully develop strategic understanding in a chess variant that differs fundamentally from classical chess.

The project will result in a functional Crazyhouse chess engine that learns through self-play, accompanied by an analysis of the strategic principles it discovers. Additionally, the project will develop an interface for human interaction, allowing users to play against the engine and enabling demonstration of its decision-making and capabilities.

The broader purpose is to contribute to the understanding of self-learning in strategic games, particularly in the domains where traditional evaluation heuristics from related games do not directly transfer. This work provides insights into the efficiency and effectiveness of neural network-based learning for a specific game variant and explore whether tested strategic principles can emerge from self-play alone.

The main research question investigated in this project is how different neural network architectures compare in terms of win-rate performance when training a self-play Crazyhouse engine. The architectures studied in this project are shallow non-residual, shallow residual, deep residual, and wide shallow residual. Additionally, the project examines whether the best-performing architecture demonstrates learning of Crazyhouse-specific strategic principles in a manner consistent with human understanding. This evaluation focuses on principles such as king-safety prioritization and whether neural network will play differently when playing the white pieces compared to the black pieces.

1.3 Thesis Overview

This thesis is organized into five chapters. Following this introduction, the structure of the report is as follows:

Chapter 2: Theory. Establishes the theoretical foundation by examining Crazyhouse mechanics, classical engine design, and the AlphaZero-inspired framework utilizing MCTS and neural networks.

Chapter 3: Implementation. Details the technical implementation, focusing on a high-performance C-based game engine integrated with a Python-driven MCTS framework and various evaluation architectures.

Chapter 4: Results. Evaluates the performance of the developed models through comparative tournaments, self-play training metrics, and head-to-head matches to determine the most effective architecture.

Chapter 5: Discussion. Synthesizes the project's findings by analysing architectural trade-offs, training successes, and the operational limitations encountered while developing a self-learning engine for complex chess variants.

2

Theory

This chapter presents the theoretical background for a self-learning Crazyhouse chess engine. It introduces Crazyhouse and its key mechanics, followed by an overview of classical chess engines and the AlphaZero approach based on self-play. The chapter then outlines key components such as Monte Carlo Tree Search (MCTS) and the Predictor + Upper Confidence Bounds for Trees (PUCT) formula, and concludes with a discussion of neural network architectures and related work.

2.1 Crazyhouse Chess Variant

Crazyhouse is a chess variant where captured pieces can be reused in the game by the capturing player [17]. This fundamental rule change, inspired by the Japanese game Shogi, transforms the strategic landscape compared to classical chess. When a piece is captured, it changes colour and enters the capturing player's pocket [4]. Instead of making a regular move, a player may choose to place one of these held pieces onto any legal empty square on the board, an action known as a drop [17].

The drop mechanic introduces several important rule changes compared to classical chess. Pawns cannot be dropped on the first or eighth ranks, preventing immediate promotion [4]. Drops that result in immediate checkmate are permitted, including pawn drops, which differ from similar variants like Shogi [17]. When a promoted pawn is captured, it is placed into the capturing player's pocket but as a regular pawn rather than maintaining its promoted status [4]. Additionally, dropped pawns placed on the second rank for White or seventh rank for Black are allowed to make a two-square advance as their first move, similar to what they can do from their starting position in classical chess [17].

The strategic implications of the drop rule alter piece valuations compared to classical chess. Pawns and knights increase their value significantly in relative importance, while rooks, queens, and bishops decrease in value [4]. This change occurs because when a king is threatened by long-range pieces like rooks, queens, or bishops from multiple squares away, dropping a pawn close by to the king provides an effective defensive resource [17]. Knights, on the other hand, cannot be blocked by dropped pieces and maintain strong offensive potential, allowing them to keep control over

key regions of the board [4].

Another example that differentiates Crazyhouse from classical chess is the value of the king's safety. Though the methods of ensuring the king's safety differ between the variants, in Crazyhouse, it becomes even more critical to ensure the safety of the king [16]. While castling remains an option in Crazyhouse, players can now also defend their king by dropping pieces around it to create protective shields against incoming attacks [16]. Position weaknesses, particularly holes where opponent pieces can be dropped and subsequently supported by additional drops, require constant attention [16]. The ability to drop pieces anywhere on the board means that players must remain alert to strategies that would be impossible in classical chess.

The opportunities of attacking in Crazyhouse mean that draws rarely happen compared to classical chess, as pieces continually return to the board rather than being permanently removed [16]. With nearly every move serving either an attacking or defensive purpose [16]. This requires careful considerations for early queen exchanges, as reintroducing the queen prematurely can leave it vulnerable to pressure from dropped minor pieces [17]. Strategic preparation is necessary to maximize the effectiveness of piece drops, particularly for high-valued pieces [17].

Despite these significant differences between Crazyhouse and classical chess, the fundamental tactics from classical chess remain applicable in Crazyhouse chess-variant [16]. Players must still avoid hanging pieces, prevent their own pieces from being blocked in, trap opponent pieces, and utilize discovered checks, forks, and pins [16]. However, the existence of the drop mechanic amplifies the importance of these tactical strategies, as pieces can materialize on any empty square to exploit weaknesses [16].

Even though Crazyhouse has developed a rather active competitive community on platforms like Lichess and Chess.com, with organized tournaments including annual world championships [26], there is a lack of established opening theory for Crazyhouse. This limit, compared to classical chess, makes it a domain where traditional evaluation heuristics do not directly transfer over to the Crazyhouse chess variant [18]. This lack of extensive theoretical development makes Crazyhouse a particularly suitable testbed for self-learning approaches, as engines cannot rely on centuries of accumulated human knowledge and must instead discover effective strategies through their own experience [5].

However, the unique characteristics of Crazyhouse present significant evaluation challenges for chess engines. This type of chess variant exhibits a considerably higher branching factor compared to classical chess due to the introduction of piece drops, as each empty square on the board represents a potential drop location for any captured piece in the player's pocket [7]. This exponential increase in possible moves at each position creates computational challenges for both traditional search algorithms and neural network-based approaches. Furthermore, formulating handcrafted evaluation functions for Crazyhouse proves particularly difficult because

game positions frequently require substantial search depth to reach relatively stable states [7].

There exist alpha-beta search engines adapted for Crazyhouse, such as the variant-enabled version of Stockfish, which requires an extensive manual parameter tuning to account for variant-specific strategies [7]. These engines determine piece values both on the board and in the pocket, along with specialized evaluation terms for concepts like king threats, mobility bonuses, and attack weights. These differ heavily from their classical chess counterparts. The manual tuning process typically employs optimization techniques but still relies on human intuition about what strategic factors matter in Crazyhouse chess [7]. In contrast, neural network-based evaluation sidesteps this challenge by learning position evaluation directly from game outcomes, though this approach demands substantial training data and computational resources to achieve competitive strength.

2.2 Classical Chess Engines

The development of chess engines began in the early 1900s with *El Ajedrecista*, regarded as the first chess-playing machine. It was an electromechanical device designed specifically to solve the King and Rook versus King endgame [15]. The first computer-based chess algorithm, *Turochamp*, was later developed in 1951 by Alan Turing. During the 1960s and 1970s, chess engines saw significant improvements as computing technology advanced. A major milestone occurred in 1997, when IBM's Deep Blue became the first computer system to defeat the reigning world chess champion in a match [10].

In November 2008, Stockfish 1.0 was released and rapidly established itself as the strongest open-source chess engine. Prior to 2017, Stockfish was widely considered the best chess engine in the world. Therefore, the chess community was taken by surprise when it lost a match to AlphaZero in December 2017. However, subsequent developments allowed Stockfish to surpass AlphaZero, and it has since remained among the strongest chess engines available.

AlphaZero introduced a fundamentally different approach by learning entirely through self-play. Instead of relying on human knowledge or pre-existing game databases, it improved by repeatedly playing games against itself, gradually discovering effective strategies. This breakthrough inspired the development of several similar engines, including Leela Chess Zero (LCZero), which has consistently ranked among the top three engines in the Top Chess Engine Championship over the past 16 seasons [24].

Classical chess engines such as Stockfish are based on the minimax algorithm. This approach involves exhaustively exploring the game tree up to a certain depth, where the engine aims to maximize its own advantage while assuming that the opponent plays optimally to minimize it. An oracle evaluates the visited positions, and the algorithm backtracks through the tree to determine the optimal move [3].

2.3 AlphaZero & Self-Learning

AlphaZero introduced a learning-based approach to game-playing that differs from the classical chess engine paradigm. Traditional chess engines rely on game databases, handcrafted evaluation functions, and human-designed heuristics. In contrast, AlphaZero learns to play solely through reinforcement learning via self-play, using only the rules of the game as its initial knowledge. This shift enables the system to discover effective strategies autonomously, without relying on human bias or predefined domain knowledge [22].

At the core of AlphaZero is a deep neural network that takes a board position as input and outputs both a policy and a value. The policy is a probability distribution over all legal moves, indicating which moves are most promising, while the value represents the estimated probability of winning from the given position. Together, these outputs replace the handcrafted evaluation functions used in classical chess engines. Instead of relying on manually designed heuristics, the network evaluates positions based on patterns learned by the neural network through its own experience [22].

AlphaZero generates games by playing against its current version, using the neural network to guide move selection. Each completed game produces training data consisting of positions, selected moves, and final outcomes. After accumulating a set of games, the network is updated by adjusting its weights to better match the observed results. The policy output is trained to align with the move distributions computed by the Monte Carlo Tree Search (MCTS), while the value output improves its accuracy in predicting the expected game result from a given position. Over time, the network learns to capture both short-term tactics and long-term strategic patterns [22].

This process is repeated iteratively, with each new version of the neural network demonstrating improved playing strength over the previous one.

2.4 Monte Carlo Tree Search

As introduced previously, MCTS is a search algorithm used to make good decisions in large, discrete search spaces such as those encountered in board games. Rather than exhaustively evaluating all possible future states, MCTS incrementally builds a search tree by sampling trajectories through the game tree and using statistical estimates to focus exploration on the most promising regions [22]. It is the core decision-making algorithm in AlphaZero and forms the foundation of the search component.

Each iteration of MCTS is organized into four repeating phases: *selection*, *expansion*, *evaluation*, and *backpropagation*. During the selection phase, the algorithm traverses the existing tree from the root node, choosing child nodes according to a

selection policy until it reaches a new child node that has not yet been expanded. A new child node is then added during the expansion phase, representing a previously unvisited game state. This newly expanded node is evaluated by a random rollout. In AlphaZero however, the newly expanded node is evaluated not by a random rollout but by a neural network that simultaneously estimates a value for the position and a policy distribution over legal moves [22]. Finally, the last phase is the backpropagation where the resulting value estimate is used to update the mean action value Q(s, a), prior probability from neural network policy P(s, a), total visits to parent node N(s), visits to this child node N(s, a), and the exploration constant at each node along the traversed path. This process yields reliable estimates over many iterations which actions lead to favorable outcomes, and at the end of the search a move is selected based on the visit counts accumulated at the root node.

2.5 PUCT Selection Formula

The variant of MCTS used in AlphaZero is based on the Predictor + Upper Confidence Bounds for Trees (PUCT) formula. This formula incorporates the neural network's policy output as a prior, guiding the search toward promising actions from the very first simulation [22]. At each state s_t, an action a_t is selected according to the following equation:

$$a_t = \arg\max_a \Big(Q(s_t,\, a) + U(s_t,\, a) \Big), \tag{2.1}$$

where the exploration bonus U is defined as:

$$U(s_t,\, a) \;=\; c_{\text{puct}} \cdot P(s_t,\, a) \cdot \frac{\sqrt{\sum_b N(s_t,\, b)}}{1 + N(s_t,\, a)}. \tag{2.2}$$

Here, $Q(s_t, a)$ is the mean value estimate, representing the perceived quality of the move, accumulated for taking action a in state s_t, $P(s_t, a)$ is the prior probability assigned to that action by the neural network policy head, $N(s_t, a)$ is the number of times action a has been visited, and c_{puct} is a constant that controls the trade-off between exploiting known nodes of action and exploring less visited ones [6]. By design, the exploration bonus U will be very large for nodes that the policy considers promising to explore but that have so far been visited infrequently. As a node accumulates visits, U shrinks and the algorithm gradually shifts toward exploiting the empirically strongest node candidate. This balance between exploration and exploitation is what makes the PUCT formula for MCTS so effective in large game trees.

When the search reaches a previously unexplored leaf state s^*, the neural network f_θ is queried and returns a policy vector $P(s^*, \cdot)$ and a scalar value estimate v^*. If a terminal state is encountered instead (game over), a deterministic outcome of -1 (loss), 0 (draw), or $+1$ (win) is used directly [22]. Since Crazyhouse is a two-player zero-sum game, the value is negated at each layer during backpropagation to reflect the alternating perspective of the players. The Q-values along the traversed path

are then updated using a simple moving average:

$$Q'(s_t,\, a) \;=\; Q(s_t,\, a) \;+\; \frac{1}{N(s_t,\, a)}\Big[\, v^* - Q(s_t,\, a)\,\Big]. \tag{2.3}$$

2.6 Neural Network Architectures

A neural network is a computational model inspired by the structure and function of biological neurons in the biological brain [19]. At its core, a neural network consists of interconnected layers of artificial neurons that process information through weighted connections [19]. Each neuron receives input values, applies weights to these inputs, adds a bias term, and passes the result through an activation function to produce an output [19]. The architecture of a neural network describes how its neurons are arranged in layers and how these layers are connected [12].

Neural networks typically consist of three fundamental components:

1. An input layer that receives raw data [19].

2. One or more hidden layers that perform intermediate computations [19].

3. An output layer that produces the final result in the form of a single value [19].

The learning process involves adjusting the weights and biases throughout the network to minimize the difference between predicted and actual outputs. This is typically done using gradient descent, an optimization algorithm that iteratively updates the network parameters in the direction that most reduces the error. This combined with backpropagation efficiently computes how each parameter contributes to decrease the error. [19].

The architecture of a neural network is described by the number of layers, the number of neurons per layer, the types of connections between layers, and the activation functions employed [12]. These architectural choices influence both the network's capacity to learn complex patterns and its computational efficiency [13]. Shallow networks, containing only one or two hidden layers, can solve simpler problems but struggle with highly complex pattern recognition tasks [13]. Deep networks, on the other hand, contain many hidden layers and therefore possess greater representational capacity, allowing them to learn important features. Earlier layers capture simple patterns, while deeper layers combine these into more abstract representations [13].

Different neural network architectures have emerged over the years to address specific types of problems. Feedforward networks, where information flows from input to output in one direction, represent the simple architecture and are suitable for basic classification tasks [12]. Convolutional neural networks (CNNs) are a type of neural

network designed to process structured data such as images. They use specialized layers that slide a window over the input to detect important patterns, such as edges, shapes, and textures. By combining these simple patterns, the network can learn to recognize more complex features, which makes CNNs particularly effective for computer vision tasks [12]. Recurrent neural networks which incorporate feedback connections that allow information to persist across time steps, enabling them to process sequential data like text or time series [12]. Each of these architecture types offers distinct advantages depending on the structure and nature of the data being processed.

For the development of a Crazyhouse chess engine, the choice of a neural network architecture carries significant implications for both training efficiency and final playing strength. The network must process board states as input and produce both a policy output that estimates move probabilities and a value output that estimates the winning probability from that position. This dual-headed architecture, successfully demonstrated in AlphaZero from Google DeepMind but for classical chess, requires careful design to balance expressiveness with computational tractability given limited training resources[13].

Residual connections, introduced in the context of neural networks for image recognition, address the degradation problem where adding more layers to a network can oddly decrease performance compared to one might think [13]. By creating shortcut paths that skip one or more layers, residual architectures enable the training of substantially deeper neural networks without suffering from vanishing gradients [13]. In a residual block, the output combines both the processed information from the layer's transformations and the original input, allowing the network to learn residual functions relative to the layer input rather than unreferenced functions [13]. This architectural innovation proved crucial to AlphaZero's success in classical chess, as it enabled the neural network to learn complex position evaluations through many layers of abstraction.

A deep residual network inspired by AlphaZero's design, incorporates a larger number of residual blocks to maximize representational capacity. Residual networks of this kind have demonstrated strong performance in domains requiring deep hierarchical feature extraction, and the conjecture motivating this design is that Crazyhouse's strategic complexity, with its expanded move space and distinct evaluation challenges, may similarly benefit from the additional layers of abstraction that greater depth provides[13].

However, increasing network depth introduces well-documented training challenges. As gradients are backpropagated through many layers, they can diminish exponentially, a phenomenon known as the vanishing gradient problem, or conversely grow uncontrollably, referred to as exploding gradients [13]. Both pathologies make it difficult for the network to learn effectively. While residual connections partially mitigate this by providing shortcut paths for gradient flow, they do not eliminate the problem entirely, particularly in very deep configurations [13].

A dedicated weight initialization scheme known as Kaiming (He) initialization was developed specifically to address this issue in deep rectified networks [9]. By scaling initial weights according to the number of input connections to each layer, the method preserves the variance of activations and gradients throughout the forward and backward passes, enabling stable training even at significant depth. This initialization strategy is therefore applied to the convolutional layers of the deep residual architecture. [9].

Comparing multiple neural network architectures can provide insights into how different design choices influence learning performance in complex domains such as Crazyhouse chess. While AlphaZero's deep residual architecture succeeded for classical chess, Go and Shogi, the distinct characteristics of Crazyhouse, including its higher branching factor and different strategic principles, may favor an alternative architectural design. Simpler architectures might prove sufficient if Crazyhouse's strategic space, while different from classical chess, does not require the same depth of abstraction. On the other hand, the introduction of piece drops might demand even greater representational capacity than classical chess. The empirical comparison across these architectural variants aims to provide evidence about these tradeoffs, specifically in the context of Crazyhouse.

2.7 Previous Bachelor Theses at Chalmers

Previous bachelor theses indicate that AlphaZero-style self-play can be successfully applied to chess variants, but that performance is highly dependent on computational efficiency and training design. In the Antichess thesis [2], a self-trained engine using MCTS guided by a neural network outperformed an MCTS-only baseline, demonstrating the benefit of learned evaluation. However, the added neural network significantly increased computational cost, resulting in slow performance on consumer hardware and limiting competitive strength.

Similarly, the Atomic Chess thesis [1] shows that self-training is feasible even with limited resources. The resulting engine achieved a playing strength above that of the average human player while remaining within a modest hardware budget. The authors emphasize that performance is largely determined by training throughput, that is, how many self-play samples can be generated and utilized effectively. To address this, they employ parallel self-play, shared data storage, and batch evaluation to improve GPU utilization.

Together, these results highlight key trade-offs between model complexity, computational efficiency, and training throughput. These factors are particularly relevant when designing self-learning systems for complex variants such as Crazyhouse, where large search spaces and expensive model evaluations can significantly impact overall performance.

3

Implementation

This chapter describes the implementation of the Crazyhouse chess engine, including a C-based game engine for efficient move generation, rule enforcement and general game logic, a Python-based MCTS framework with neural network guidance, and multiple neural network architectures for position evaluation. The implementation follows an iterative development approach, beginning with a working baseline and progressively adding complexity and optimizations.

3.1 Implementation Overview

The Crazyhouse engine architecture is divided into modules, where each module handles a specific aspect of game play, learning, or position evaluation. At the core of the system is a C-based game engine responsible for the authoritative game state, legal move generation, and state transition logic. This low-level implementation ensures the efficiency required for fast execution, as a large number of position evaluations must be performed during MCTS.

The C-based engine communicates with Python through a shared library interface, combining efficient rule execution with high-level learning algorithms. The search process is carried out using MCTS, where legal moves are generated by the game engine and evaluated using policy and value predictions from a neural network.

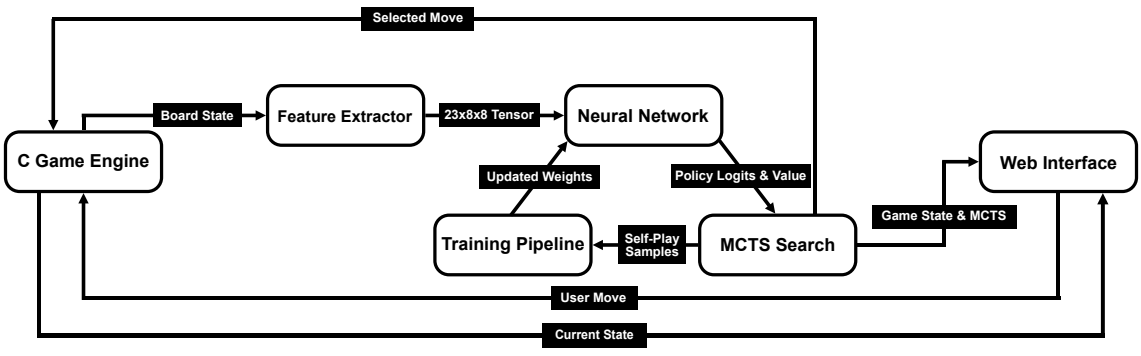

Figure 3.1: Wiring diagram of the Systems Architecture

A single implementation of the Crazyhouse rules is reused everywhere. The same C functions are called from self-play, evaluation, and the web interface, which means that a move considered legal by the game engine is also legal during search and during human interaction. That choice is important for reproducibility because it avoids separate rule implementations that could drift apart.

3.2 Game Rules and Move Generation

The game state is implemented and represented as a Crazyhouse structure, where it stores the board, promoted piece markers, pocket pieces, castling rights, player turn, en passant file, cached king coordinates, move history, state history, and the halfmove clock. The board itself is an 8×8 array of piece values, where each piece only stores a type and a colour.

Because the piece structure does not store whether a piece originated from a pawn promotion, the engine maintains a separate promoted matrix with the same dimensions as the board. Each entry in this matrix indicates whether the piece currently occupying that square is a promoted pawn. This distinction is necessary because a promoted queen is otherwise indistinguishable from a normal queen in the board representation, since both are stored simply as pieces of type queen.

The promoted matrix is required for correct Crazyhouse capture handling. According to the Crazyhouse rules, when a promoted piece is captured, it does not return to the capturer's pocket as its promoted type. Instead, it reverts back into a pawn before being added to the pocket. For example, if a pawn promotes to a queen and that queen is later captured, the capturing player receives a pawn rather than a queen. The engine therefore checks the promoted matrix during captures to determine whether the captured piece should be converted back into a pawn.

Move generation is implemented through the generator that scans all squares, selects the pieces that belong to the active player, and tests every reachable target square. Candidate moves are first checked against the piece-specific movement rules and then filtered by king safety. Legal moves are stored in a fixed-size array of at most 500 entries, which is large enough to cover all realistic Crazyhouse positions while still keeping the memory layout simple.

The regular move rules is implemented and handled by pawns support single and double pushes, diagonal captures, promotions, and en passant. Sliding pieces check that their path is clear. Kings are limited to one square moves, while castling is handled separately because it depends on both the current board layout and the attack status of the transit squares.

The Crazyhouse specific part is the drop generation. For every empty square, the engine checks whether the current player has any captured pieces of the type that may be dropped. Pawns cannot be dropped on the first or eighth rank. The implementation also avoids repeated drop entries by only generating one move per

droppable piece type and destination square, even if the pocket contains multiple copies of that piece type.

When a move is applied, the engine updates the board, the promoted matrix, the pocket arrays, castling rights, the en passant file, the move history, and the draw history to keep the game state consistent after every move.

3.2.1 Board Representation

The board representation is intentionally designed for both search and neural network inference. The engine state stores the board as a matrix of pieces, but the network does not consume that matrix directly. Instead, the module `feature extractor` converts the state into a tensor shape of size $(23, 8, 8)$, which is the channel first layout expected by PyTorch.

The tensor is constructed plane by plane and the first 12 channels encode piece occupancy with one hot plane for white and black pawns, knights, bishops, rooks, queens and kings. The channel index is computed directly from the piece type and colour, so the mapping is deterministic and easy to reproduce. In other words, a white bishop on a given square activates the white bishop plane at that square, while all other piece planes at that square remain zero.

Channels 12 to 16 encode the current player's pocket, one plane for each droppable piece type possible: pawn, knight, bishop, rook and queen. These are not sparse occupancy maps. Instead, the value of each plane is the normalized count of that piece type in the pocket, broadcast across the full board. This means that every possible drop location receives the same global pocket signal. The reason for this design is that pocket pieces are not tied to a specific square, so a spatial encoding would not add any useful meaning or structure. Broadcasting makes the availability of drops visible to the convolutional trunk at every location at all times.

Channel 17 stores the side to make the next move. The entire plane is filled with zero when white is to move and one when black is to move. Channels 18 to 21 store castling rights in the same broadcast format. Channel 22 stores the en passant target square, with a single one at the target location and zero everywhere else.

The resulting input tensor is therefore not just a board image. It is a compact state description that combines local information, such as piece placement with global information, such as pocket counts and castling rights. That design is what makes the representation suitable for Crazyhouse rather than ordinary chess.

The translation is implemented as follows:

$$\text{piece plane} = (\text{piece type} - 1) \cdot 2 + \text{colour}$$

$$\text{feature index} = \text{channel} \cdot 64 + \text{row} \cdot 8 + \text{col}$$

This direct indexing is what allows the C code to fill the buffer with a single scan over the board.

3.2.2 Standard Chess Move Generation

For ordinary board moves, the generator scans every piece on the board belonging to the active player and tests all target squares. A move is then accepted only if the piece specific movement rule allows the move and the temporary application of that move does not expose the king. The attack logic is implemented in a targeted way where pawns, knights, kings, rook and queen horizontal move lines, and bishop and queen diagonal move lines are checked separately instead of generating every possible opponent reply, as shown in the following pseudo code:

```
IsMoveLegal(game, from, to):
  save board state that may be modified temporarily
  apply the move on the board
  if a king moved:
    update the cached king position
  if own king is attacked in the new position:
    mark the move illegal
  restore the original board state
  restore the cached king position
  return whether the move is legal
```

This implementation of the temporary apply and restore pattern where the king's new position is cached, makes it easy to keep the king safety check local to analyse if an illegal move has been made when moving the king before the resulting move is sent to the C engine. This way, the C engine does not need to generate the full reply tree of the opponent before it is decided whether a move is legal or not.

3.2.3 Crazyhouse Extension

Crazyhouse extends ordinary chess with the ability to capture opponent pieces and later drop them back onto the board as your own. The C engine tracks these reserve pieces in separate arrays for white and black. When a piece is captured, it is appended to the pocket of the player who made the capture. When that player later drops a piece, the piece is removed from the pocket and placed directly on the board. The important implementation detail is that the pocket is tracked by piece type rather than by square. That means the pocket behaves like a multiset. If the player has two knights and one rook in the pocket, the C engine can generate all legal knight drops and all legal rook drops independently of where those pieces came from.

Promoted pieces are handled with a separate boolean matrix. If a promoted pawn is captured, the pocket receives a pawn and not the promoted piece type. This preserves the Crazyhouse rule that promotions do not stay promoted after capture.

The rule is applied both in the ordinary move path and in the move application path, so it is not just a display detail.

The move generator also avoids duplicate work by only generating one drop move per piece type and target square. That is sufficient because the pocket count is represented separately from the move itself. The count is handled later by the pocket state, not by repeating the same move entry.

3.2.4 Legal Move Filtering

A move is not considered legal until it has passed the king safety test. The legality check is implemented by temporarily applying the candidate move, updating cached king coordinates if needed and then querying whether the moving side is in check. If the king is attacked after the temporary move, the move is rejected and the original state is restored.

The attack detector is optimized for the board geometry. It checks the relevant pawn attack offsets, the eight knight jumps, the eight king neighbours and sliding directions for rooks, bishops and queens. This is more efficient than simulating all opposing moves and is sufficient because the only question being asked is, if any enemy piece currently is attacking the king square or not locally on the board.

This implementation also handles the en passant edge case explicitly. A pawn that captures en passant temporarily removes the captured pawn from the board before the check test is performed. Without that step, the legality test could incorrectly report that the king is safe or unsafe because the blocking pawn would still be present.

3.3 MCTS Implementation

The search logic for the MCTS is implemented in Python, while legal move generation and feature extraction still comes from the C engine. The implementation during the search starts from a cloned root state, repeatedly selects child nodes using the PUCT formula. It then applies the chosen move to the cloned board state through the C engine and then evaluates the resulting new child node with the neural network or via a terminal state logic.

The MCTS implementation uses a single `MCTSNode` structure that stores the parent link, the children dictionary, the prior probability, the accumulated value, the visit count and a small amount of terminal state bookkeeping. The only non standard field is the stored move object itself. That field is important because the stateful search later reconstructs the chosen move from the actual C engine move object instead of relying only on the policy index.

The implementation keeps the fixed exploration constant at 1.4, which is set in a shared Python file and is reused everywhere the search class is constructed. The

simulation budget is also fixed per run so that neural network architecture models can be compared under the same search budget, and the default number of simulations per search is set to 128. These values are chosen as stable defaults for the training and testing setup.

3.3.1 MCTS Structure

The search is organized as a rooted tree where the root corresponds to the current game position. Each edge from a node corresponds to one legal move from that position. During search, the repeated simulations update the node statistics, and these statistics are later used to produce a move policy at the root.

At the beginning of a search, a new root is initialized and expanded from the current game state. If no legal moves exist, the position is handled as terminal directly. Otherwise, legal children are created and assigned priors from the neural network policy output. The search then runs a sequence of simulations that repeatedly traverse and update the tree.

This structure is intentionally aligned with the workflow described in chapter 2.4, but implemented with direct integration to the projects game state interface and legal move generator.

From these values, the implementation computes the mean value of an action used in selection. This allows each node to represent both how often an action has been explored and how good the action has been in previous simulations.

3.3.2 Selection and Expansion

Selection repeatedly walks from the root to a leaf by choosing the child that maximizes the PUCT score, as given by equation 2.1 in section 2.5. What matters for the implementation is that the score combines three pieces of information which is the prior policy from the network, the mean value of the node and the visit count of the node relative to its parent.

Expansion then occurs when the search reaches a node that has not yet been expanded and the position is not terminal. The engine generates the legal moves from the current game state, converts those moves to policy indices, evaluates the position with the neural network and creates one child node per legal move. The child nodes store the prior probabilities directly from the policy output, which is why the legal move mask must be applied before the softmax calculation.

3.3.3 Evaluation and Backpropagation

The neural network returns two values for every evaluated position, a policy logits over the full move space and a scalar value prediction. Before the policy is used, the implementation masks illegal moves and applies softmax so that probability mass is

assigned only to legal actions. The legal move mask is created from the exact move list returned by the C engine, which keeps search and rule enforcement aligned.

The leaf value is stored at the leaf node from the perspective of the player to move there. It is then negated at each step as it propagates upward, so that every node's Q-value always reflects the value from the perspective of the player to move at that node. The sign flip in the PUCT score calculation then converts a child's Q-value back to the parent's perspective for selection.

3.3.4 Final Move Selection

After the simulation budget is exhausted, the root visit counts are turned into a move distribution. The implementation uses temperature to control how deterministic this distribution is. For temperature zero, the most visited child is chosen directly. For positive temperature, the visit counts are transformed into probabilities with the following rule:

$$\pi_\tau(a) = \frac{N(a)^{1/\tau}}{\sum_b N(b)^{1/\tau}} \tag{3.1}$$

Here $N(a)$ is the root visit count for action a and τ is the temperature. A low temperature makes the distribution sharper and is good when the strongest immediate play is the goal, while a higher temperature keeps more alternatives alive.

In self-play, the visit distribution is also mixed with uniform noise through an epsilon parameter. The implementation therefore samples from:

$$\hat{\pi}(a) = (1 - \epsilon)\pi_\tau(a) + \epsilon \cdot \frac{1}{|A_{legal}|} \tag{3.2}$$

where $|A_{legal}|$ is the number of legal actions at the root. This is a simple and explicit way to increase exploration without changing the search algorithm itself. The current implementation uses a schedule with different temperature and epsilon values for the opening, middle game, and endgame.

The MCTS search module returns the updated root node, and then either samples from the visit distribution or takes the argmax depending on the use case. That means the same search result can be used for training data generation, deterministic evaluation, or human playable interface.

3.4 Neural Network Architectures

All architectures are convolution neural networks designed to evaluate Crazyhouse positions and help guide the implemented MCTS. Each network predicts a policy

distribution for legal moves and a value estimate for the current position.

The implementation for the different neural network architecture models are further compared in Section 3.4.2.

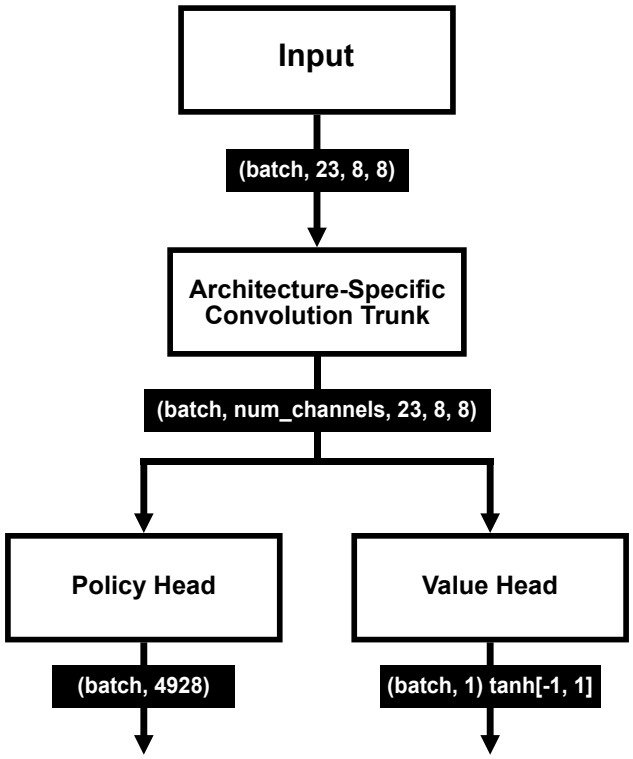

Figure 3.2: Overview of the neural network architecture used in this thesis, illustrating the input representation, architecture-specific convolution trunk, and the separate policy and value heads. The neural network processes an input tensor and produces both policy outputs and a scalar value estimate.

The overall design uses a shared trunk with a dual-head structure, where a convolutional feature extractor creates spatial features that are then sent to separate policy and value prediction heads.

3.4.1 Common Architecture Components

All architectures use the board representation and move encoding scheme introduced in Section 3.2.1. The same input tensor format and policy indexing are reused across all neural network architecture models to ensure a fair comparison between architectures.

The convolutional trunk keeps the structure of the board through the network. Since all convolutional layers have 3×3 kernels with padding, the board representation is still 8×8 after every layer. The input for the network is a tensor with shape $(23, 8, 8)$, which transforms into an internal feature representation. The number of

channels then depends on the architecture. The baseline, shallow residual and deep residual architectures have 128 channels while the wide shallow architecture has 256 channels.

In this context, depth refers to the number of stacked residual blocks, where one residual block includes 2 convolution layers, and width refers to number of feature channels used in each convolutional layer. Increasing depth will allow the neural network to capture more complex strategic patterns across the whole board, and increasing the width allows the network to learn a larger number of feature representations at every board position.

The policy head first reduces the trunk output to 32 channels with a 1×1 convolution. This acts as a learned dimensionality reduction before the final linear layer. The purpose is to shrink the feature map before flattening it into the final linear layer. This keeps the number of parameters in the final classifier manageable while preserving sufficient information to distinguish the 4928 move indices.

The value head reduces the trunk output to a single channel with a 1×1 convolution. This is a stronger compression because the value head only needs one scalar output. After flattening, the value head uses one hidden fully connected layer and a final `tanh` activation to keep the prediction inside $[-1, 1]$.

All architectures use batch normalization and ReLU activation in the trunk after every convolution layer. The residual variants places 2 convolutional layers inside each residual block so that the input of each block is added back to its output, hereby creating the skip connection in a residual neural network architecture. That makes the training more stable for deeper neural networks because gradients can flow through the skip connection, keeping the signal strong.

The policy and value heads are shared across all architectures and are summarized in Table 3.1. The policy head reduces the number of channels to 32 before the final classifier, while the value head reduces the representation to a single channel before scalar regression. This difference reflects the higher representational demands of policy prediction compared to scalar value estimation.

Head	**Layer**	**Description**
Policy	Conv 1×1	Reduce trunk output to 32 channels
	BatchNorm + ReLU	Non-linear projection
	Fully connected	Outputs move logits
Value	Conv 1×1	Reduce trunk output to 1 channel
	BatchNorm + ReLU	Non-linear projection
	Fully connected	Hidden layer
	Fully connected	Outputs scalar value
	Tanh	Output in $[-1, 1]$

Table 3.1: Shared structure of the policy and value heads.

All architectures share the same input representation, output representation, activation functions, loss functions, and optimization procedure. The architectures differ only in structural properties such as the number of convolutional layers (depth), the number of feature channels (width), and the presence or absence of residual connections. This setup enables a controlled investigation of how architectural complexity influences learning performance in Crazyhouse.

The output representation also includes separate policy indices for promotions. A promotion is encoded not only by destination square but also the piece type. So promotions to a queen, rook, bishop, and knight are treated different in the policy space, even when the pawn moves to the same square. This allows the policy head to accurately learn different promotions from self-play instead of promotions counting as the same move.

In the additional information for the promoted pieces, the network still needs to keep current colour and movement behaviour after promotion to know which side the piece belongs to and distinguish it from originally owned pieces of the same type. Without this feature, promoted pieces would be treated identical to standard pieces of the same type, which could lead to inaccurate evaluations.

An alternative to this implementation would have been to treat queen promotions as ordinary pawn moves since it is the most common piece to promote to in practical play. However, they all are represented in separate policy actions in order to have a consistent move encoding. It also uses the idea that a queen promotion is the most common, which from a neutral perspective might not be true.

3.4.2 Architecture Comparison

The four architectures differ in trunk depth, channel width, and the use of residual connections. Table 3.2 summarizes the structural differences between the implemented neural network architectures.

Architecture	**Hidden Channels**	**Residual Blocks**	**Trunk Style**
Shallow Non Residual Baseline	128	0	Sequential
Shallow Residual Baseline	128	6	Residual
Deep Residual Architecture	128	20	Residual
Wide Shallow Residual	256	6	Residual

Table 3.2: Summary of the implemented neural network architectures.

Shallow Non-Residual Baseline

The Shallow Non-Residual Baseline is the simplest architecture and serves as a reference model. It consists of three stacked convolutional layers with 128 channels and does not include residual connections. This sequential design prioritizes computational efficiency but limits representational capacity.

Shallow Residual Architecture

The Shallow Residual Baseline extends the non-residual architecture by introducing residual connections while maintaining the same input layer of three stacked convolutional layers with similar depth and channel width throughout. The architecture includes six residual blocks, with 2 convolutional layers per block, allowing improved gradient flow and more stable training without significantly increasing architecture size.

Deep Residual Architecture

The Deep Residual architecture is further expanded upon from the Shallow Residual architecture and increases the depth by using twenty residual blocks, resulting in a significantly deeper architecture. This design aims to capture more complex positional features at the cost of increased computational requirements.

Wide Shallow Residual Architecture

The Wide Shallow Residual architecture takes the opposite approach compared to the Deep Residual architecture and increases channel width from 128 to 256, while still maintaining a shallow depth. This allows the model to learn a larger set of feature representations without increasing depth.

3.5 Training Setup

The training procedure for the Crazyhouse chess engine was designed as an iterative self-learning pipeline, inspired by AlphaZero. The overall setup consists of three components: self-play generation, storing of generated games and optimization of the neural networks based on the stored games. In each iteration, the neural network architecture was used to guide self-play games through the MCTS, thereby producing training samples from the explored positions. These samples were then collected and reused during training, enabling the neural networks to learn from a broad set of positions rather than only the most recently generated games. After optimizations, the updated checkpoints were used for the next round of self-play, which results in a closed feedback loop in which the networks gets incrementally improved through more generated data. All training and evaluation were performed on Minerva, a compute cluster hosted by Chalmers University of Technology and the University of Gothenburg. Batch scripts were used to automate self-play, training, and evaluation. Initial development and testing were conducted on local machines to verify correctness before running experiments on the cluster.

3.5.1 Self-Play Generation

Self-play was used as the primary mechanism for generating training data. In this stage, the Crazyhouse chess engine played games against itself, with move selection guided by MCTS in combination with the neural network's policy and value predictions.

For each position, the Crazyhouse chess engine generates legal moves with the C game engine, encodes the position as features, and runs MCTS to obtain an improved move distribution. The chosen move is then applied to the current game, and the resulting position is stored together with the search distribution and the final game outcome.

The exploration behaviour is controlled by a piecewise schedule. Early in the game the code uses a higher temperature and a higher epsilon. Later in the game both values are reduced so that the search becomes more deterministic. The schedule is defined over the opening, middle game, and endgame by dividing the maximum number of plies into three intervals.

The implementation uses two separate quantities during move selection. The root visit distribution is first computed from the search tree, and then an epsilon mixture is applied before sampling. That is why the self-play solution stores both the visit based policy target and the actually sampled move.

```
PlaySelfGame(game):
  while the position is not terminal and the ply limit is not reached:
    legal_moves = generate legal moves
    features = get features from the C engine
    root = run MCTS from the current state
    pi = root visit distribution at temperature 1
    sample move from the visit distribution mixed-
        -with epsilon uniform noise
    record a sample using the current state and pi
    apply the sampled move to the game
  assign a final value target to every sample in the game
  save the samples to disk
```

As shown, the stored sample is more specific than a single dense tensor. The training pipeline expects `pi_indices` and `pi_values`, which are sparse representations of the policy target, together with `state_flat` and the final scalar target `z`. That sparse format keeps the saved data compact while still allowing the dense policy vector to be reconstructed during training.

3.5.2 Replay Buffer Management

The generated data from self-play was stored in a generated replay buffer, allowing it to be reused during training. Instead of training on each game immediately after it was generated, samples were accumulated over multiple self-play runs.

Each training sample consists of the board features, the sparse policy target, and the scalar outcome target. The target value is assigned after the game ends. A win, loss, or draw is converted to the configured reward value, and the same result is assigned to every position in the game from the perspective of the player to move

at that position.

3.5.3 Loss Calculation and Optimization

A neural network specific architecture was trained using a joint objective consisting of a policy loss and a value loss. The policy loss compares the predicted move probabilities with target distributions generated from self-play and is implemented using cross-entropy loss over the policy logits. The value loss measures the difference between the predicted position evaluation and the final game outcome using mean squared error. The total loss is defined as the sum of these two components, encouraging the neural network to improve both move selection and position evaluation simultaneously.

```
TrainBatch(batch):
  policy_logits, value = model(batch.features)
  policy_loss = cross entropy between logits and target policy distribution
  value_loss = mean squared error between value and target outcome
  loss = policy_loss + value_loss
  backpropagate loss
  update parameters
```

Additionally, L2-regularization was used in the optimizers weight decay parameter. This is used to penalize large weights during training, which helps avoid overfitting and encourages the model to learn more generalized features.

3.5.4 Training Loop Structure

The overall training loop alternated between phases of self-play and neural network optimization. First, a set of self-play games was generated using the current model, and the resulting samples were added to the accumulated dataset. Next, these samples were loaded into a PyTorch dataset and processed in mini-batches using a data loader. For each batch, the model performed a forward pass to produce both policy logits and a value estimate, after which the combined loss was computed and backpropagated through the network.

3.6 Evaluation Framework

The evaluation framework is designed to compare the performance of different neural network architectures trained through self-play. The primary objective is to assess relative playing strength under controlled and consistent conditions, ensuring that observed differences arise from architectural design rather than external factors.

3.6.1 Round-Robin Tournament Setup

Evaluation was done as a round robin tournament. Each model played every other model the same number of games, and the colours were swapped so that neither

side benefited from always starting first. The implementation also seeded the games with a small set of opening lines so that the comparison was not dominated by a single opening family.

The number of games per pairing was fixed, resulting in a balanced dataset where all models were evaluated under the same number of interactions. This structure allowed for direct comparison of win rates and overall performance across all architectures.

Move selection during evaluation is deterministic. The temperature was set to zero, which means that the most visited root child is always selected. That removes sampling noise and makes the match outcomes easier to compare across models.

All evaluation games used a fixed simulation budget per move. That kept the computational effort constant even when models differed in inference speed. The evaluation script then recorded win, loss, and draw counts, as well as draw causes such as stalemate, repetition, the fifty move rule, and maximum ply termination.

No time constraints were imposed per move. Instead, each move was computed using a fixed simulation budget, ensuring equal computational effort for all models regardless of hardware variability.

3.6.2 Metrics Collection

The evaluation script stores the final results in a structured form so that the outcomes can be summarized later. For each pairing, it tracks wins, losses, draws, and the reasons for draws. The derived metrics are win rate, draw rate, and loss rate.

The draw breakdown is useful because two models can have the same total draw rate while still reaching draws for very different reasons. A large number of stalemates suggests positional pressure, while a large number of maximum ply draws suggests that the game often remains unresolved for a long time.

$$\text{draws} = d_{max} + d_{50m} + d_{rep} + d_{stale}$$

This decomposition is retained in the report because it gives more insight than a single draw number. It also mirrors the data that the evaluation script writes to the results files. The draw results are categorized based on their underlying cause:

- **Maximum move limit (d_{max}):** Games that terminate after reaching a predefined maximum number of moves without a decisive result.
- **50-move rule (d_{50m}):** Games drawn due to the absence of captures or pawn moves within the last 50 moves per player.

- **Threefold repetition (d_{rep}):** Games where the same board position occurs three times.

- **Stalemate (d_{stale}):** Games where the current player has no legal moves but is not in check.

3.6.3 Strategic Understanding Tests

While quantitative metrics such as win rate provide a useful measure of overall performance, they do not fully capture the extent to which a model has developed a deeper understanding of the game. The current implementation primarily relies on round robin tournaments, but the code base also supports more qualitative inspection through fixed openings and direct board analysis. Those tests are useful when a model plays well numerically but still chooses moves that look strategically weak to a observer.

In practice, this means that the evaluation pipeline can be extended with fixed board positions or opening sequences without changing the underlying search code. The same engine, the same feature extractor, and the same move index mapping can be reused for that kind of analysis.

3.7 User Interface

The project includes a Flask based web interface for interactive testing. The backend exposes routes for reading the current state, listing checkpoints, configuring the bot side, submitting moves, triggering engine moves, resetting the game and starting or stopping Stockfish based comparison games. The interface uses the same C engine and the same MCTS implementation as the training pipeline, so the behaviour seen in the browser is consistent with the behaviour used during self-play.

The backend keeps the game state in memory and converts it to a board matrix and pocket counts for the frontend. The frontend is responsible only for rendering and user input. Legal move checking, terminal state detection and move application still happen in the engine layer, which keeps the interface thin and avoids duplicating game logic in JavaScript.

3.7.1 Graphical Board Display

The graphical board is rendered in the browser as a fixed 8×8 grid with separate pocket displays for white and black. The board is updated after every move, and the frontend highlights the selected square, the legal target squares, the king square when the side to move is in check, and the current pocket counts. This makes the Crazyhouse specific mechanics visible without changing the underlying engine code.

The board representation used by the frontend is derived directly from the C game state. Pieces are converted to single character symbols and then mapped to Unicode

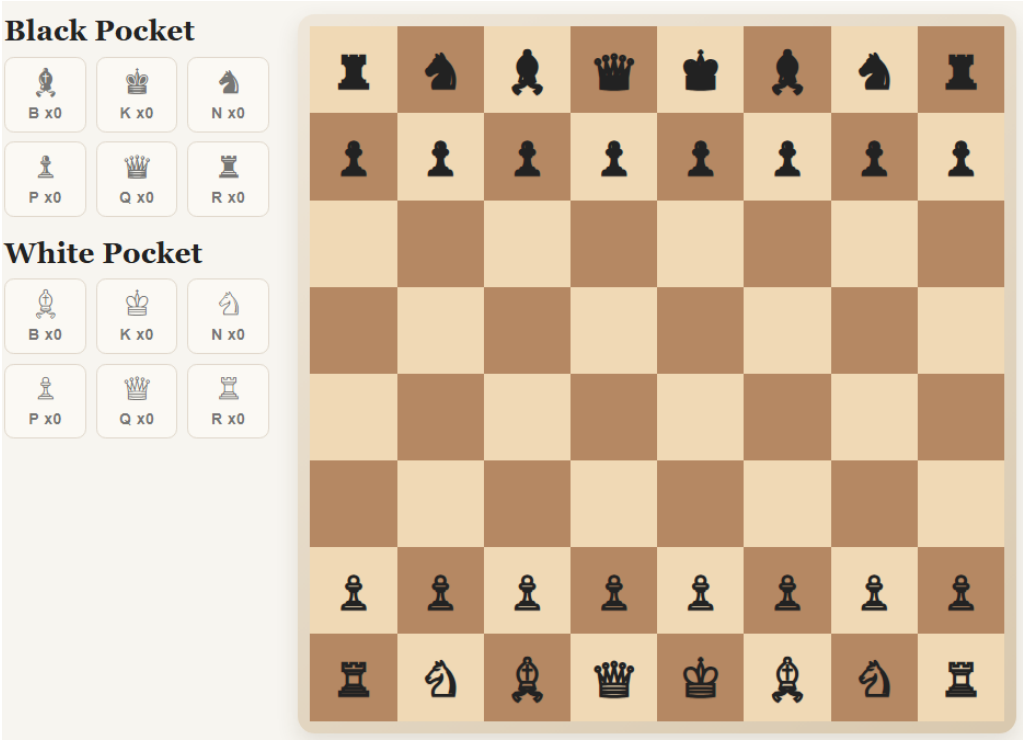

Figure 3.3: User interface of the Crazyhouse chess engine.

chess characters for display. The display therefore reflects the engine state rather than a separate visual state.

3.7.2 Interactive Gameplay and Match Configuration

The interface supports human-versus-engine play through a dedicated match configuration panel. Before a game starts, the user can configure which side is controlled by a human or an engine, select one of the implemented neural-network architectures, provide a checkpoint path, and set the number of MCTS simulations used per engine move. When the match is started, the backend resets the game, loads the selected checkpoint, reconstructs the corresponding network in evaluation mode, and assigns the engine to the appropriate side. This makes the interface a practical testing environment for examining trained models outside the self-play pipeline.

In addition to human-versus-engine play, the interface also supports engine-versus-engine matches. In this mode, both sides are controlled by neural networks, each configured with a selected architecture, checkpoint, and number of MCTS simulations. This enables direct comparison between different models under identical conditions within the same interactive environment.

During play, human moves are entered directly through the graphical board. A normal move is created by selecting a source square and a destination square, while a Crazyhouse drop move is created by first selecting a piece from the pocket and then selecting a target square on the board. The frontend only enables interaction on

the human player's turn, but all submitted moves are still validated by the C engine before being applied. When it is the engine's turn, the backend runs MCTS guided by the loaded neural network, using the shared engine implementation for feature extraction, legal move generation, move application, and terminal-state checks. After the chosen number of simulations, the move associated with the most visited child node is selected and applied to the current game. The board, pockets, and status indicators are then updated immediately, and the game ends automatically on checkmate, stalemate, or draw by threefold repetition. This ensures that the interactive mode relies on the same rule implementation and search procedure as the rest of the project.

As an addition to these evaluations of the models, a interface for playing versus the Stockfish engine was developed to be able to estimate the potential chess-ELO of the neural networks.

4

Results

This chapter presents the evaluation and performance results of the developed neural network architectures, including architecture model comparisons with a model that plays random moves, a tournament between all the architectures, and a head-to-head match-up. The results begin with architecture match-ups with a random engine and a round-robin tournament with all architectures, and gradually isolate the most promising architectures for extended training and final strength assessment.

4.1 Model Comparison

To compare the different neural network architectures, we ran an evaluation program in which each model played games against every other architecture. We had four different architectures: baseline non-residual, shallow residual, deep residual, and wide shallow residual. We also included a model playing completely random moves in our evaluation to see how well our architectures were performing. Figure 4.1 shows how our models performed when playing against solely the random model over time.

Figure 4.1 shows that our engine improves with training, and that all of our architectures are making moves with strategic intent rather than random selection.

Figure 4.2 tested all four neural network architectures at multiple checkpoints throughout their training progress, measuring their win rates when competing against each other. The architectures played 60 games against each other. The graph shows win rate percentage on the y-axis and checkpoint index on the x-axis.

The architectures demonstrated competitive performance throughout the tournament, with win rates between 13-32% at a majority of the checkpoints. However, the wide shallow residual architecture achieved a performance spike at the final checkpoint (40), reaching 51.7% win rate, a big improvement from 15.0% at checkpoint 30. The other architectures remained stable, with baseline and shallow residual around 30-32% at the last checkpoint.

Figure 4.3 presents a direct head-to-head evaluation, isolating the match-ups to only

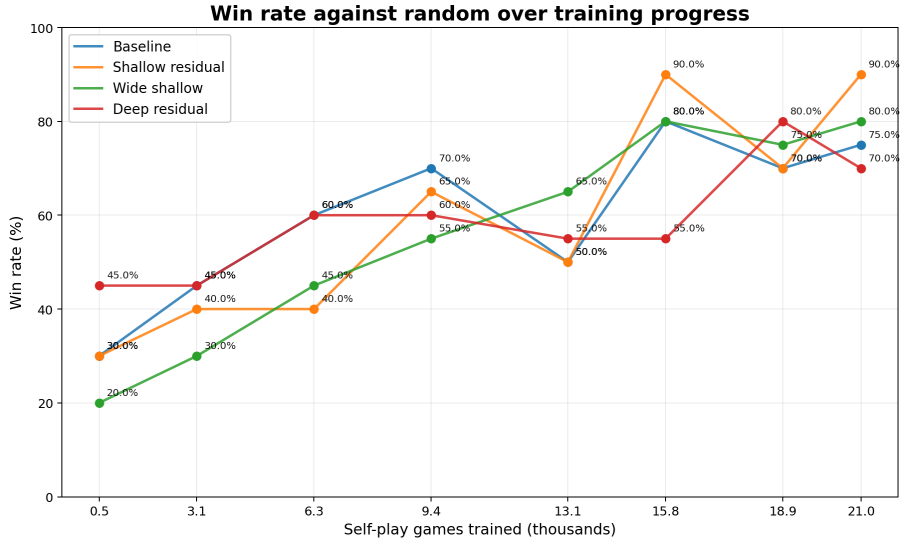

Figure 4.1: Win rate progression of the four neural network architectures against the random model during self-play training.

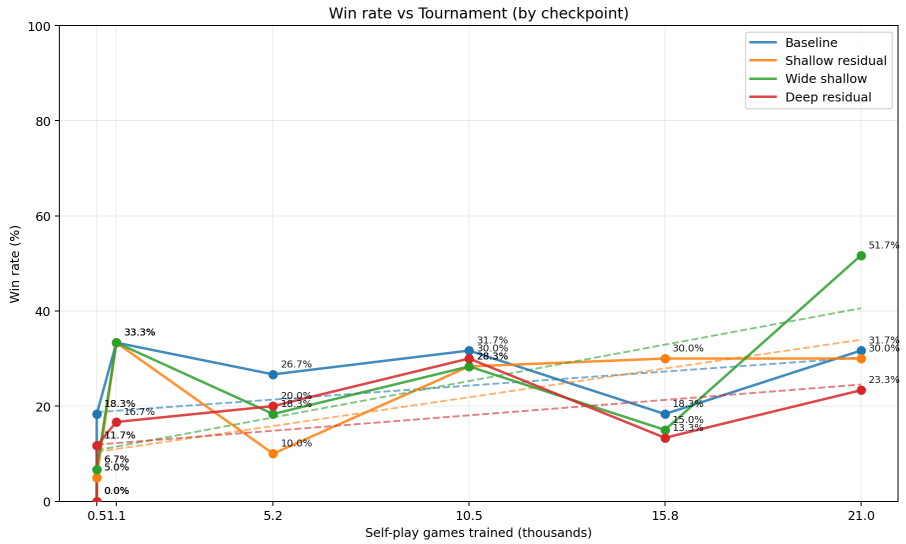

Figure 4.2: Results of all the neural networks in head-to-head evaluation during self-play training.

the shallow residual and wide shallow residual models. This final internal comparison identifies the strongest model for the final assessment against Stockfish.

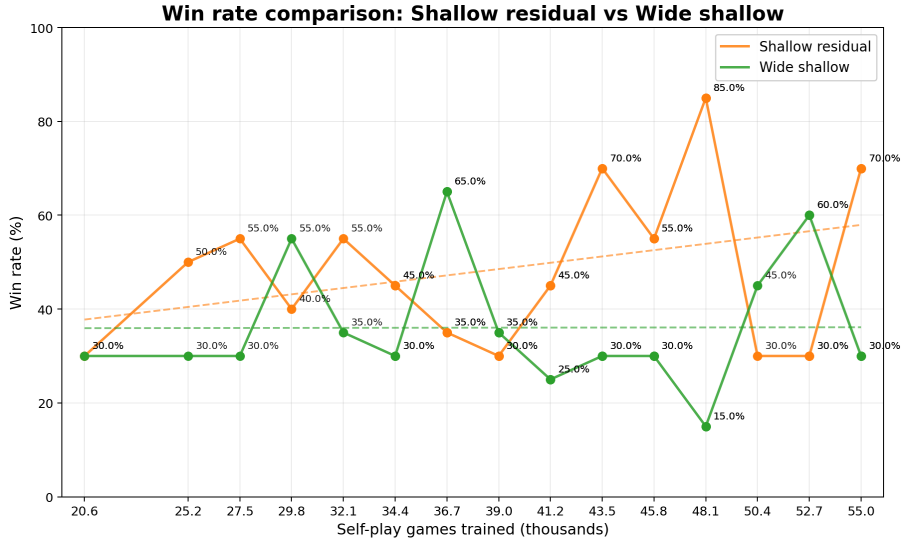

Figure 4.3: Head-to-head evaluation between the shallow residual and wide shallow residual architectures during the final stage of training.

Figure 4.3 illustrates a highly competitive performance between the shallow residual and wide shallow residual architectures. However, the shallow residual model demonstrates a marginal superiority, securing a higher win-rate in nine of the evaluated checkpoints compared to wide shallow residual's five. This edge is further supported by the regression lines for each architecture, which position shallow residual as the top-performing architecture. Based on these findings, the subsequent evaluation focuses exclusively on the shallow residual architecture to assess the current capabilities of the chess engine.

4.2 Final Engine Strength

To determine the final strength of our engine, we decided to match it up against different levels of Stockfish. The following table demonstrates our final evaluation, where shallow residual played 100 games against level one through five Stockfish. The ELO of Stockfish level one is estimated to be 800, and level five is estimated to be 1600 ELO. We decided not to play against any higher ELO since the result did not change between the levels we played.

Estimated Elo	Games won by NN	Games Drawn	Games lost by NN	Win %
~800	0	1	99	0
~1100	0	1	99	0
~1350	1	2	97	1
~1500	0	3	97	0
~1600	0	0	100	0

Figure 4.4: Results from matches between the shallow residual network and Stockfish at different levels of strength.

5

Discussion

This chapter presents a critical analysis and detailed interpretation of the results presented from core architectural choices, analysis of the self-play training and the operational constraints of the implemented engine. By reflecting on the choices made throughout the project, the challenges encountered, both successes and limitations, key insights can be derived regarding the necessary trade-offs between model complexity, computational efficiency, and strategic depth in developing self-learning engine for complex chess variants.

5.1 Motivation for Choosing Crazyhouse

The choice of Crazyhouse as the target chess variant was motivated by two complementary considerations; its deep strategic depth that exists and its possibilities for different approaches in self-learning.

From a strategic standpoint, Crazyhouse is substantially more complex than classical chess. The drop mechanic presents huge changes in the fundamentals of chess, there is an altered material evaluation, piece evaluation and a king safety considerations in ways that have no direct parallel to classical chess and its theories. This makes it a domain where hand-crafted strategies are inadequate and where self-learning approaches that builds its own evaluation entirely from experience has a strong theoretical justification. The almost complete absence of established opening theory further strengthens this case and a self-trained engine in Crazyhouse cannot draw on centuries of accumulated human knowledge and must discover effective strategies entirely through self-play instead.

From a feasibility standpoint, Crazyhouse was also chosen based on observations from the two preceding bachelor theses at Chalmers on a related topic. The Antichess project from 2024 demonstrated that a neural network guided MCTS engine can be meaningfully trained within the time frame of a semester for this type of project, though inference costs on consumer hardware were a limiting factor [2]. The Atomic Chess project from 2025 showed that the same approach is feasible under a tight budget, producing an engine above average human level on Lichess and identifying training throughput as the key bottleneck [1]. These results suggested that

a similar effort applied to Crazyhouse was plausible within the project's constraints while still making it an achievable challenge, provided that training throughput was treated as a primary design concern from the start.

5.2 Crazyhouse as a Self-Learning Domain

As touched upon a little bit before, Crazyhouse present challenges as a self-play learning domain that go beyond those of both classical chess and the previously studied variants. The most significant is the infinite return of material because captured pieces can re-enter play as drops from the pocket rather than being permanently removed. This means that games can extend for far longer compared to classical chess and decisive imbalances are harder to sustain in a match. This creates two concrete consequences for self-play training.

Firstly, the training signal is noisier and harder to learn from. In classical chess, a material advantage is relatively durable and the position evaluation it implies is stable across many subsequent moves. In Crazyhouse, material advantages are temporary, meaning a captured piece by the opponent immediately strengthens their drop resource. The value head must therefore learn a more context dependent evaluation function that accounts not only for board material but also for pocket contents and the positional factors that determine how effectively drops can be executed. This is a considerably more demanding learning problem than standard chess evaluation.

Secondly, the risk of uninformative games during training is higher. Games that are not resolved by a decisive positional advantage tend to continue until the ply limit is reached. Therefore, inflating the proportion of draws in the training data and reducing the quality of the value signal. The ply limit draw penalty ($z- = 0.1$) and the three phase exploration schedule implemented in the self-play pipeline were designed to partially mitigate this, but their effect was moderate. This is a fundamental consequence of Crazyhouse's game structure rather than a correctable implementation issue. And, addressing it fully would likely require more sophisticated training techniques such as adaptive value target shaping during training or curriculum-based training, where we gradually let the neural network learn more and more complex strategies with time.

These consequences highlights challenges that were not fully anticipated. Crazyhouse is arguably the most strategically complex of the three variants examined across the past three years of thesis work, which likely contributes to the difficulty observed here. Although both the wide and shallow residual models demonstrate clear evidence of learning through self-play, neither configuration was able to consistently defeat Stockfish at any tested skill level ranging from approximately 800 to 1600 ELO, confirming that the trained neural networks remain substantially weaker than even the lowest Stockfish setting. These results suggest that, while policy improvement is occurring, the learned representations are still insufficient to compete with even modest handcrafted search-based engines for the chess variant Crazyhouse.

5.3 Analysis of Engine Performance

5.3.1 Interpretation of Round-Robin Results

The round-robin results showed a consistent and largely reproducible ordering across all evaluation phases, with the pattern becoming more pronounced as training progressed. The choice to use a collection of chess openings in the evaluation was justified with the motivation that the broader understanding of the chess variant was of large interest. By setting up evaluations with openings we ensured equal opening advantages and disadvantages for both neural networks, as well as being able to analyse if the engines have the ability to adapt to positions it has not encountered enough times prior.

Throughout the first training phase, all four architectures fluctuated considerably across checkpoints 1 through 40, where shallow residual and wide shallow residuals regressions both trended upward significantly, whereas the baseline and deep residual showed flatter trajectories. This divergence in regression trend, rather than any single checkpoint result, motivated the decision to carry only shallow residual and wide shallow residual architectures forward into the second training phase.

In the second training phase, after baseline and deep residual were frozen at their current checkpoints, the two active architectures continued to improve substantially while the frozen opponents' relative win rates declined, confirming that the gains were driven by continued learning rather than evaluation variance. Towards the end of the second training phase, the latest evaluation round showed that while shallow achieved a win rate of 70.0% compared to wide shallow residual's 30.0% at the last checkpoint, the regression analysis also clearly favored the shallow residual architecture. Despite high variance in individual checkpoints, shallow residual's regression line demonstrates a stronger upward trajectory over time, indicating a more consistent long-term improvement trend which is why we proceeded with shallow residual as our architecture of choice going forward into the last phase of head-to-head evaluation against Stockfish.

These results provide a much more appropriate answer to the main research question regarding win-rate comparison across architectures. Residual connections provide a clear and consistent advantage over sequential, non-residual computation. The non-residual baseline was consistently outperformed by both residual architectures of comparable depth, suggesting that skip connections significantly improve gradient flow and learning stability in the context of Crazyhouse position evaluation. This finding is consistent with residual network theory [13] and with AlphaZero's own reliance on a deep residual trunk.

The deep residual architecture's consistently perform poorly, starting at 5.00% win rate at checkpoint 1, roughly around 1.000 training games, and increasing to approximately 23% in the final checkpoint round warrants for a careful interpretation. This result should not be taken as evidence that architectural depth is inherently

disadvantageous for Crazyhouse. However, it reflects the mismatch between the deep neural network's capacity requirements and the training data volume that was achieved during a very short and limited time period. AlphaZero trained its deep residual architecture on tens of millions of self-play games using thousands of TPUs [22]. Towards the end of training phase one at checkpoint 40, approximately 20.000 self-play training games, the deep residual architecture is far below the scale of data its design requires to begin demonstrating its potential. A further contributing factor is inference cost, the 41-layer trunk evaluates positions substantially more slowly than the 13-layer shallow architecture models, meaning the deep residual generated fewer total games within the same training time budget, compounding the data deficiency in the first training phase. The resource constraints of this project therefore make it impossible to draw any final conclusions about the long-term potential of the deep residual architecture for Crazyhouse, and the current results only characterize the performance at a scale where the architecture is undeniably undertrained.

The comparison between the shallow residual and wide shallow residual architectures is arguably the most informative result of the project. Because, both architectures operated within a training scale where meaningful learning clearly occurred, making the divergence between them interpretable rather than noise. The fact that shallow residual's regression line trends clearly upward while wide shallow's remains flat around 35% suggest that the two architectures respond differently to additional training data, rather than simply fluctuating around a shared performance ceiling. One might expect the wider network, with 256 channels width compared to shallow residual's 128 channels, to hold a structural advantage given its greater representational capacity. Yet the results suggest the opposite at this training scale, the added width appears to hinder the wide architecture model rather than help, possibly because the wider network requires significantly more data to meaningfully utilize its additional parameters. Shallow residual, being a leaner architecture, may generalize more effectively from the volume of self-play data generated at the end of this phase.

Whether this advantage would hold or disappear at a larger training scale remains an open question. Is it plausible that wide shallow residual would eventually close the gap or overtake shallow residual give sufficient data. But, within the scope of this project, the evidence consistently favors the shallow residual architecture, and the upward regression trend suggests this is a signal worth taking seriously rather than a temporary fluctuation.

5.3.2 Challenges of Crazyhouse

Several challenges emerged during the project that were specific to the variant or to the implementation stage, and which affected both the validity and the interpretation of the results.

Draw Rate Inflation

Initial analysis after approximately 4.500 self-play training games was achieved revealed draw rates of over 61% for the deep residual architecture and 30-42% for

the other architecture models during evaluation. Root cause analysis identified two contributing factors to this. Firstly, randomly generated bit-strings in the Zobrist hash initialization caused threefold repetition detection to fail silently under multi-process evaluation. Meaning, different processes could compute different hashes for the same position on the board if their random state was not synchronized, causing repetitions to go undetected. This was addressed by verifying that the fixed seed was applied deterministically at library load time across all processes. Second, all draw categories had been added together in the evaluation output, making it impossible to distinguish between repetition, stalemate, the 50-move rule and the maximum ply limit. The evaluation framework was updated to track all four draw types separately, enabling more targeted diagnosis and allowing future runs to attribute draws to their correct causes.

Promotion Encoding Gap

Early on in the project, it was discovered that promotion moves were not correctly encoded in the policy output. All four promotion choices (knight, bishop, rook and queen) were mapped to the same policy index, meaning the network's promotion type preferences were not correctly represented and the engine effectively always promoted to queen regardless of what the policy head suggested. To fix this the policy space was expanded from 4.096 indices to 4.928 by introducing explicit promotion-specific indices for all four piece types, correcting this inconsistency and improving the learning signal quality.

MCTS Move Resolution

A more subtle bug was identified in the MCTS implementation. During tree traversal, move application could fail when a selected child move was compared against the current simulation state's legal moves by object identity rather than by move content, which could cause mismatches when the same move was represented differently in different states. This was resolved by converting each move into a UCI string, a standard text format, so moves could be compared fairly. If two moves happened to look identical in that format, we used a secondary check to tell them apart. This made sure that whenever the code applied a move during a simulation, it always picked the right one, every time, without any random or unpredictable behaviour.

Deep Residual Value Head Instability

The deep residual architecture initially exhibited training instability, with the value head's hyperbolic tangent output saturating near ± 1 early in training before the network had learned anything meaningful about the positions evaluation. This was addressed by initializing the final linear layer of the value head with weights drawn from $\mathcal{U}(-0.01, 0.01)$, keeping initial value predictions close to zero and preventing premature saturation early on in the training of the different branches in the tree. Kaiming normal initialization was then later applied to all convolutional layers to maintain healthy gradient magnitudes throughout the deep branch.

5.3.3 Actions based on Performance Results

The project followed an iterative approach in which results from the evaluation directly informed the next decision. Two such decision are particularly worth going over and examining in light of the full evaluation data at hand.

The first decision was towards the end of training phase one when the last evaluation round-robin of checkpoint 40 showed that shallow residual (51.7% win rate) impressively outperformed both the non-residual baseline (31.7% win rate) and deep residual (23.3% win rate). Wide shallow residual (30% win rate) displayed more potential compared to baseline, also clearly indicated by the upward trend of the regression displayed in figure 4.2. Rather than continuing training of all four architectures for completeness, it was decided to halt training of the baseline and deep residual architectures and concentrate the full compute power on the two strongest performers. This was a deliberate resource trade-off. The relative order of the last evaluation round for training phase one was stable across multiple preceding evaluation rounds, giving confidence that the gap reflected genuine learning differences rather than evaluation noise or bugs. Additionally, redirecting compute power to the two more promising architectures was more likely to yield a cleaner answer to the main research question than spreading resources evenly.

The second decision was reflected in the second training phase evaluation trajectory. As wide shallow residual and shallow residual continued training, a divergence began to emerge, towards the middle of this training phase (where both had roughly 55000 games). Shallow residual began to show a clear regression trend towards 58.0% compared to wide shallow residual's 34.0%. This emerging gap in the dedicated head-to-head evaluation between these two architectures presented in figure 4.3, is the basis for selecting shallow residual architecture for the last evaluation phase, head-to-head evaluation against Stockfish.

The draw rate and implementation correctness issues identified during training were similarly treated as blocking problems requiring immediate resolution. The decisions to invest time in proper draw classification, Zobrist hash verification, promotion encoding correction and MCTS move resolution, rather than accepting noisy or incorrect results and proceeding, reflect a consistent implementation commitment to ensuring the evaluation framework produces valid data before conclusions are drawn from it.

5.3.4 Comparison to Other Variants

Comparing these results with the Antichess (2024) and Atomic Chess (2025) theses provides useful context. The Atomic Chess project achieved a playing strength above the average Lichess player after approximately 450.000 self-play games over 150 hours of training [1]. The Antichess project achieved more modest results, with the neural network guided engine outperforming a flat MCTS baseline but remaining below competitive human-level play [2]. The current project's Crazyhouse chess engine were unable to win consistently against Stockfish at the lowest skill level after

approximately 55.000 training games, indicating a lower absolute playing strength than the Atomic Chess result at comparable scales.

Several factors likely explain this gap. Crazyhouse's larger effective branching factor, due to piece drops adding a new class of moves at every ply, means that each self-play game explores a smaller fraction of the relevant game tree than in Atomic Chess or Antichess, requiring more games to achieve comparable positional coverage. The value learning problem is also harder, as discussed previously. These factors collectively suggest that Crazyhouse requires significantly more training data to reach equivalent strength and that the scale achievable within a single academic semester is a more limiting bottleneck for this variant than for the previously studied ones.

5.4 Limitations

Cluster limitations

As previously stated the neural networks were trained on the Minerva Cluster, this was both a opportunity and limitation. As it is hosted at Chalmers it was a zero-cost solution that made training on more powerful hardware more readily available throughout the training and evaluation of the neural networks. But it also came with some limitations.

1. **Knowledge**

   We had limited prior experience with working on remote clusters as well as building a neural network based chess engine. This meant that although we got the program running remotely quite fast, the code execution was neither fast nor effective. We had many issues with crashes that led to more and more iterations of the different neural network architectural models. Limited knowledge on how to optimize for the specific purposes along with limited information both on the CPU/GPU utilization meant that the only metric we could interpret was time per match and total time to play and train on the self-play matches. This led to a lot of questions regarding if we've hit the limits, if it's too complex or if there is something wrong with the code. After many iterations and hours researching we found ways to monitor CPU and GPU usages, but lacked the knowledge to optimize further.

2. **Queuing**

   Another limitation was the unpredictable nature of a seemingly open cluster. The cluster is widely available to Chalmers and GU students, this led to a highly variable queue time as well as an inconsistent access to the most powerful hardware. Therefore we deemed it to be best to not optimize or use specific cluster nodes as we assumed it would have led to fewer hours on the cluster. Which in turn would have led to fewer games played. Although we chose to not target specific nodes, we still had long queue times during periods and

several weeks. This combined with the fact that we discovered rule breaking behaviours and bugs led to us having less games played on the final networks than we would have liked.

Time Limitation

Despite implementing strategies informed by prior bachelor theses, our project encountered substantial obstacles that hindered the total completion of neural network training. Specifically, cluster resource limitations imposed strict constraints on computational availability and training duration. Additionally, we encountered unforeseen technical challenges that hindered the progress as well as making many hours of training worthless. The combination of restricted access to computing infrastructure, unexpected implementation difficulties, and the inherent time constraints of the project prevented us from achieving full model convergence. Consequently, the final neural network (using shallow residual architecture) remained undertrained and failed to demonstrate performance metrics that would be considered meaningful or competitive within the field. While we made incremental progress and gained valuable insights throughout the development process, the model's final predictive strength remained below the threshold necessary for practical application or meaningful comparative analysis. As we also set out to compare 4 different architectures, this meant that a big part of the project was spent on spreading out the limited resources between the different architectures.

Input And State Representation

During the later stages of the project, three concrete limitations were identified in the representation of input and the current game state. Each very likely to have affected both the quality of the training but also the final engine strength. Each represents a specific design choice that, in hindsight, could be improved in future work.

1. **Difference from AlphaZero: the side-to-move channel**

   AlphaZero tackles the problem of perspective by representing the position from the active player's point of view. Meaning that the board was rotated so that the current moving player always play upward. This means that the neural network never needs to learn that white and black pieces behave identically but from opposite directions. We did not implement it like this, instead we have a fixed board orientation and an explicit channel is added to indicate whose turn it is. This means that the network has to learn that the same piece configuration is advantageous for the white pieces but disadvantageous for the black pieces. All dependent on channel 17, which results in another layer of abstraction and in turn a potential increase in the amount of training required for the network to generalize correctly for both colours.

2. **The limitation of the opponent's pocket not being visible**

   The current implementation uses 23 channels, missing the 5 channels required to represent the opponents pocket. Although 23 is sufficient when developing simple strategies, it lacks the ability to interpret all possible defensive and offensive strategies of the opponent. The input tensor currently only encodes the active player's pocket and not the opponent's which makes it effectively invisible to the network. This creates a blind spot that would be harmless in classical chess but is a significant handicap in Crazyhouse, where drop threats fundamentally shape both attacking and defensive strategies.

5.5 Future Research

The findings of this project reveal several concrete directions for future work on self-learning engines for the chess variant Crazyhouse.

The most impactful improvement for future work would be an increase in training scale. The win rate trajectories for the shallow residual architecture had not converged by the end of the project period and the architecture continued to improve throughout training. Extending training with access to longer Minerva job allocations or additional GPU resources would likely produce substantially stronger engines and enable a more definitive comparison for this architecture and the others, especially wide shallow residual. A natural target would be to scale the amount of training games towards an estimate of 450.000 self-play games like the previous thesis Atomic Chess did, which produced an engine above average human level for that variant.

Another related direction for future work is the optimization of training hyperparameters, particularly the exploration schedule and the ply limit penalty. The current values were chosen based on reasonable defaults rather than observational tuning and it is plausible that different schedules would produce a better balance between position diversity and learning signal quality. Automated hyperparameter search using a subset of available compute budget could be a valuable addition to the pipeline.

The two evaluation metrics, win rate against a random opponent and win rate in round-robin tournaments, revealed different pictures of relative architecture performance. Against random, deep residual reached 70% by checkpoint 40, comparable to wide shallow residual and baseline, suggesting it had learned basic strategic play despite underperforming in competitive head-to-head matches. This discrepancy indicates that future work should use both evaluation types from the start, as tournament performance alone may obscure whether a model has learned fundamental positional understanding versus competitive strength specifically.

A concrete representational improvement that should be prioritized in future work is expanding the input tensor from 23 to 28 channels by adding the opponent's

pocket as explicit input. As discussed in the representation section, the network currently must infer the opponent's available drop resources indirectly from what is missing on the board. Given that Crazyhouse tactics are heavily dependent on pocket contents, this information asymmetry likely introduces unnecessary noise into the value head's learning signal. Measuring whether value loss decreases with a 28-channel representation would directly test this hypothesis.

The evaluation implementation could also be strengthened for future research through the addition of systematic strategic understanding tests. The current evaluation is based entirely on win rates in round-robin tournaments and observational game analysis. A more rigorous assessment of Crazyhouse-specific strategic understanding could be constructed by testing the engine's move selection on a set of handcrafted positions that require variant-specific reasoning and strategies, such as positions where the optimal move is a tactical piece drop that would have no analogue in classical chess. Comparing the engine's selection against expert analysis in these positions would provide a richer and more direct measure of what the self-play training has produced.

From an architecture standpoint, the results indicate that shallow residual architectures with moderate channel width represent the most effective design choice under the computational constraints of this project. After training phase one, data suggests that the shallow residual architecture with 128 channels and six residual blocks has developed the stronger learning trajectory of the two best architectures. Future work on this could explore whether this advantage extends when training further, or if a wider architecture catches up and surpasses the shallow residual. An interesting question would be to investigate whether there is a point of decreasing returns in channel width for Crazyhouse position evaluation. Alternatively, if substantially more compute power becomes available, revisiting the deep residual architecture with sufficient training data to approach convergence would be a valuable experiment and the results of this project would then make it much clearer if depth is not inherently disadvantageous for Crazyhouse but only that it requires a scale of training data that was not achievable here.

Finally, the evaluation framework could be improved by adding systematic tactical tests on handcrafted Crazyhouse positions, where the best move is a piece drop. This would be distinct from strategies derived from classical chess. A set of fixed test positions with known correct moves would allow strategic understanding to be measured independently of overall playing strength, providing a richer picture of what self-play training actually produces in terms of Crazyhouse-specific reasoning.

5.6 Conclusion

This project set out to design and see if a self-trained chess engine could be capable to learn and play the chess variant Crazyhouse. Additionally the project set out to compare different neural network architectures when trained through AlphaZero-inspired self-play. The project also aimed to assess whether the best-performing ar-

chitecture demonstrates Crazyhouse-specific strategic understanding consistent with human knowledge of the variant.

The primary research question was answered, but perhaps not as clearly as we would have hoped for. The residual architectures consistently outperformed the non-residual baseline, thereby confirming that the skip connections provide a meaningful benefit for gradient flow and learning stability in the context of Crazyhouse position evaluation. Among the residual architectures, width proved to be more effective than depth under the time and computational constraints of this project.

The secondary goal of demonstrating Crazyhouse-specific strategic understanding was only partially achieved. Through observation of gameplay using the graphical interface, we observed no colour-dependent strategy nor any clear prioritization of king safety, indicating that while the models learned purposeful play, deeper variant-specific understanding had not yet emerged. All trained architectures clearly learned to play with strategic intent beyond random move selection, as confirmed by their win rates against the random baseline increasing consistently throughout training. However, none of the models showed any higher level play or strategy. Consistently loosing against Stockfish at any tested skill level, ranging from approximately 800 to 1600 ELO, which means that the depth of strategic understanding could not be verified against a meaningful external benchmark.

Several limitations affected the final outcome of the project. The total amount of games played was insufficient for the models to achieve competitive strength, and the training curves had not converged at the end of the project, indicating that more training would likely produce further improvement. The input representation encoded only the active player's pocket and not the opponent's, this created a information asymmetry that likely complicated value learning in a variant where the opponent's available drops are highly relevant to both offensive and defensive strategies. Cluster availability and queue times on Minerva introduced additional unpredictability in training duration.

Despite these limitations, the project produced meaningful and interpretable results. The finding that shallow residual architectures with moderate to wide channel width represent the most effective design choice under the given constraints is a concrete and actionable conclusion. The consistent improvement observed across all residual architectures through self-play confirms that the AlphaZero-inspired approach is applicable to Crazyhouse, even if the scale required to reach competitive strength exceeds what is achievable within a single academic semester.

Bibliography

[1] Hannes Adolfsson et al. "A Self-Trained Chess Engine for Atomic Chess: Leveraging Self-Training for the Creation of Chess Bots in Underexplored Variants". Supervisor: Andreas Abel. Bachelor's thesis. Chalmers University of Technology and University of Gothenburg, 2025.

[2] Gustaf Algeskog et al. "A Self-Trained Engine for a Chess-Variant: Combining a Neural Network and Monte Carlo Tree Search to Create an Antichess Engine". Supervisor: Andreas Abel. Bachelor's thesis. Chalmers University of Technology and University of Gothenburg, 2024.

[3] Antoine Champion. *Dissecting Stockfish Part 3: In-Depth Look at a Chess Engine*. Towards Data Science. July 22, 2024. URL: https://medium.com/data-science/dissecting-stockfish-part-3-in-depth-look-at-a-chess-engine-51b59e532bb4 (visited on 04/29/2026).

[4] *Crazyhouse. Captured Pieces Can Be Dropped Back on the Board Instead of Moving a Piece*. Lichess. URL: https://lichess.org/variant/crazyhouse (visited on 04/29/2026).

[5] crosky. *Crazyhouse, an Overview*. Lichess. URL: https://lichess.org/blog/VrQDNSoAACsA8sqc/crazyhouse-an-overview (visited on 04/29/2026).

[6] Johannes Czech, Patrick Korus, and Kristian Kersting. "Monte-Carlo Graph Search for AlphaZero". In: *arXiv preprint* arXiv:2012.11045 (Dec. 2020). Accessed: Feb. 19, 2026. URL: https://doi.org/10.48550/arXiv.2012.11045.

[7] Johannes Czech et al. "Learning to Play the Chess Variant Crazyhouse Above World Champion Level with Deep Neural Networks and Human Data". In: *Frontiers in Artificial Intelligence* 3 (2020), p. 24. DOI: 10.3389/frai.2020.00024.

[8] Google DeepMind. *AlphaZero and MuZero*. Google DeepMind. 2024. URL: https://deepmind.google/research/alphazero-and-muzero/ (visited on 04/29/2026).

[9] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[10] Internetmuseum. *Datorn Deep Blue besegrar världsmästaren i schack*. Internetstiftelsen. June 13, 2024. URL: https://internetmuseum.se/tidslinjen/datorn-deep-blue-besegrar-varldsmastaren-i-schack/ (visited on 04/29/2026).

[11] Maxim Khovanskiy. *AlphaZero Chess: How It Works, What Sets It Apart, and What It Can Tell Us*. Towards Data Science. May 2022. URL: https://towardsdatascience.com/alphazero-chess-how-it-works-what-sets-it-apart-and-what-it-can-tell-us-4ab3d2d08867 (visited on 04/29/2026).

[12] T. L. Makosso, A. Almaktoof, and K. Abo-Al-Ez. "Review of Different Types of Neural Network Architectures". In: *International Journal of Electrical Engineering and Applied Sciences* 7.2 (2024). URL: https://ijeeas.utem.edu.my/ijeeas/article/view/6213 (visited on 04/29/2026).

[13] Ibomoiye Domor Mienye and Theo G. Swart. "A Comprehensive Review of Deep Learning: Architectures, Recent Advances, and Applications". In: *Information* 15.12 (2024), p. 755. DOI: 10.3390/info15120755.

[14] Mistreaver. *History of Chess Computer Engines*. Chessentials. Jan. 23, 2019. URL: https://www.chessentials.com/history-of-chess-computer-engines/ (visited on 04/29/2026).

[15] Jeremy M. Norman. *Torres y Quevedo Invents El Ajedrecista, the First Decision-Making Automaton*. HistoryofInformation.com. Dec. 28, 2025. URL: https://www.historyofinformation.com/detail.php?id=569 (visited on 04/29/2026).

[16] okei. *Guide to Crazyhouse*. Lichess. URL: https://lichess.org/@/okei/blog/guide-to-crazyhouse/zaytMmOC (visited on 04/29/2026).

[17] PyChess. *Crazyhouse*. PyChess. URL: https://www.pychess.org/variants/crazyhouse (visited on 04/29/2026).

[18] Raven-Ivy-Serena. *Guide to Crazyhouse Chess*. Chess.com. URL: https://www.chess.com/blog/Raven-Ivy-Serena/guide-to-crazyhouse-chess (visited on 04/29/2026).

[19] Grant Sanderson. *But What Is a Neural Network? Deep Learning, Chapter 1*. 3Blue1Brown. 2017. URL: https://www.3blue1brown.com/lessons/neural-networks (visited on 04/29/2026).

[20] André Schulz. "The Thinking Game – How DeepMind Transformed Artificial Intelligence". In: *ChessBase* (Nov. 27, 2025), n.p. URL: https://en.chessbase.com/post/the-thinking-game-how-deepmind-transformed-artificial-intelligence.

[21] David Silver et al. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play". In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404.

[22] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv* (Dec. 2017). arXiv: 1712.01815. URL: https://doi.org/10.48550/arXiv.1712.01815 (visited on 01/24/2026).

[23] Stockfish Team. *Introducing NNUE Evaluation*. Stockfish. Aug. 6, 2020. URL: https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/ (visited on 04/29/2026).

[24] TCEC. *Top Chess Engine Championship*. Chessdom. URL: https://tcec-chess.com/ (visited on 04/29/2026).

[25] Joost VandeVondele. *Remove Classical Evaluation*. GitHub. July 11, 2023. URL: https://github.com/official-stockfish/Stockfish/commit/af110e02ec96cdb46cf84c68252a1da15a902395 (visited on 04/29/2026).

[26] visualdennis, FischyVishy, and QueenRosieMary. *Announcing the Crazyhouse World Championship 2025*. Lichess. 2025. URL: https://lichess.org/@/visualdennis/blog/announcing-the-crazyhouse-world-championship-2025/CMcYy3sy (visited on 04/29/2026).

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY AND GOTHENBURG UNIVERSITY
Gothenburg, Sweden
www.chalmers.se

GÖTEBORGS
UNIVERSITET

CHALMERS