A Self-Trained Chess Engine for Atomic Chess

Leveraging self-training for the creation of chess bots in underexplored variants

Bachelor's thesis in Computer science and engineering

Hannes Adolfsson
David Lewis
Anton Rahmn
Sigge Rajamäe
Edvin Rungardt
Marco Tafani

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Bachelor's thesis 2025

A Self-Trained Chess Engine for Atomic Chess

Leveraging self-training for the creation of chess bots in underexplored variants

Hannes Adolfsson
David Lewis
Anton Rahmn
Sigge Rajamäe
Edvin Rungardt
Marco Tafani

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

A Self-Trained Chess Engine for Atomic Chess

Hannes Adolfsson David Lewis Anton Rahmn Sigge Rajamäe Edvin Rungardt Marco Tafani

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiner: Patrik Jansson and Arne Linde, Department of Computer Science and Engineering
Graded by teacher: Yehia Abd Alrahman, Department of Computer Science and Engineering

Bachelor's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Abstract

Chess has long been a benchmark for artificial intelligence (AI) research due to its complexity and well-defined rules. Recent advances, such as AlphaZero, introduced self-learning AI through reinforcement learning and self-play, achieving superhuman performance without prior strategic knowledge, relying solely on the rules of the game. AlphaZero defeated the world-champion chess engine Stockfish after only four hours of training, leveraging large-scale computational resources to rapidly learn and refine its strategies.

This thesis presents the development of a chess engine for the chess variant Atomic Chess. The engine was developed in C++ and trained through self-play and reinforcement learning, taking inspiration from AlphaZero's approach. This project explores the extent to which a chess engine with this approach is feasible for the average enthusiast.

Cost-effective cloud-based virtual machine instances with powerful hardware were rented to manage training workloads. Given limited computational resources, we opted for a data-centric approach, focusing on refining the training pipeline to maximize the training data that could be produced, rather than hyperparameter tuning and experimenting with neural network architectures. The final engine was trained on approximately 450,000 self-play games in roughly 150 hours.

The final engine was deployed on the chess platform Lichess and achieved an ELO-rating of 1,729, which corresponded to the top 10th percentile of Atomic Chess players on Lichess. These results demonstrate that it is possible to achieve a competitive Atomic Chess engine within a budget of 3,000 SEK for cloud computation. This shows that strong self-play reinforcement learning agents for niche games can be developed without requiring large-scale computing infrastructure. These results highlight the viability of accessible, low-budget AI research for underexplored game variants.

Sammandrag

Schack har sedan länge använts för att evaluera forskning inom artificiell intelligens på grund av spelets komplexitet samt väldefinierade regler. Nya framsteg, såsom AlphaZero, introducerade självlärande AI genom förstärkningsinlärning och självspelande som uppnådde övermänsklig prestanda utan någon tidigare strategisk kunskap, enbart utifrån spelets regler. AlphaZero besegrade den dåvarande bästa schackmotorn, Stockfish, efter endast ett fåtal timmars träning, där den utnyttjade storskalig hårdvaruinfrastruktur för att snabbt bli bättre och finslipa sina strategier.

Denna avhandling presenterar utvecklingen av en schackmotor för schackvarianten Atomic Chess. Motorn utvecklades i C++ och tränade genom självspelande och förstärkningsinlärning, likt AlphaZero. Detta projekt utforskar om dessa metoder är rimliga för den vanlige entusiasten.

Kostnadseffektiva molnbaserade virtuella maskininstanser med kraftfull hårdvara hyrdes för att hantera träningen. Med tanke på den begränsade tillgången till hårdvara valde vi ett datacentrerat tillvägagångssätt med fokus på att förfina träningspipelinen för att maximera mängden träningsdata som kunde produceras, istället för att justera hyperparametrar och experimentera med neuronnätverksarkitekturer. Den slutgiltiga modellen tränades genom att spela approximativt 450,000 partier mot sig själv under en sammanlagd träningsperiod på 150 timmar.

Den resulterande schackmotorn gjordes tillgänglig på schackplattformen Lichess och uppnådde en ELO-rating på 1,729, vilket motsvarar den översta 10:e percentilen av Atomic Chess-spelare på Lichess. Resultaten visade att det är möjligt att skapa en motor för Atomic Chess som spelar på en hög nivå inom en budget av 3,000 SEK för att hyra hårdvara. Detta visar att starka självspelande förstärkningsinlärningsmotorer gjorda för nischade spel kan utvecklas utan att behöva storskalig hårdvaruinfrastruktur. Dessa resultat belyser möjligheten med lättillgänglig, lågbudgetbaserad AI-forskning för outforskade spelvarianter.

Acknowledgements

We would like to extend our gratitude to our supervisor, Andreas Abel, who has offered us his guidance and support throughout this entire project. Furthermore, the evaluation of the project would have been significantly more difficult had it not been for the players and creators of bots who competed against our engine on Lichess. Lastly, we would like to show appreciation to the many people who have developed the open source chess platform lichess.org.

Hannes Adolfsson, David Lewis, Anton Rahmn, Sigge Rajamäe, Edvin Rungardt, Marco Tafani. Gothenburg, June 2025.

Contents

List of Figures

List of Tables

1
Introduction

1.1 Background

Chess has been a central subject in artificial intelligence (AI) research for decades due to its complexity and well-defined rules. The development of chess engines has evolved significantly over time, from brute-force search methods to sophisticated AI-driven approaches. A major milestone in this evolution was Deep Blue's victory over Garry Kasparov, the chess world champion at the time, in 1997 [1]. Deep Blue was used by IBM as a demonstration of their processing prowess, but also became an example that handcrafted functions for the evaluation of chess positions could beat even the strongest human players.

In 2017, a new paradigm emerged with AlphaZero, which showcased the effectiveness of self-learning AI [2]. Unlike traditional engines with hand-crafted evaluation functions, AlphaZero relied solely on reinforcement learning and self-play, where it in simple terms played games against itself, learned by reinforcement from those games and then repeated this process to continue improving on itself iteratively. With this self-learning approach, AlphaZero progressed from having no prior knowledge of how to play chess to becoming the world's strongest chess engine within 4 hours of training.

Chess variants modify the standard chess rules, such as board size, piece movement, or win conditions. These variations present new opportunities for AI research due to the lack the well-established theory and human expertise in these variants when compared that which exists for regular chess. This makes them an ideal testing ground for self-playing AI engines, which develop strong strategies entirely on their own.

Since the creation of AlphaZero, computational strength of computer hardware has dramatically improved. The increased interest in machine learning has also led to improvements specifically in hardware that specializes in this. These factors result in a lower barrier of entry to train an AI model similar to AlphaZero and can therefore be achieved for a lower price today compared to in 2017 when AlphaZero was created. This is what this project aims to investigate.

1.2 Purpose

This project seeks to build a self-play engine for the chess variant named Atomic Chess, utilizing reinforcement learning to discover effective strategies without hand-crafted heuristics. It also aims to investigate how AI can successfully adapt to unfamiliar strategic environments while working within a level of expenditure on hardware that keeps the training easily replicable, and to discover which optimizations and techniques are the most impactful in improving the performance and training of a model as to train as efficiently as possible on limited hardware.

To be more specific, the research questions that this project aims to answer are as follows:

- **Q1:** How feasible is it to train a chess engine through self-play, similar to AlphaZero, using computational resources available to the common enthusiast?
- **Q2:** How can a chess engine using a self-training model be designed and optimized to work at this scope?

2

Theory

This chapter outlines the theoretical foundations of our Atomic Chess engine. We begin by describing the rules and unique mechanics of Atomic Chess, followed by an overview of chess engines and AlphaZero, the self-learning chess engine that this project is attempting to replicate. Key concepts in Monte Carlo Tree Search (MCTS) and neural networks are then introduced, focusing on the modifications and techniques relevant to AlphaZero and our implementation. The chapter concludes with a discussion of move generation techniques, bitboard representations, and related work, providing context for the design decisions presented in Chapter 3.

2.1 Atomic Chess

Atomic Chess is a variant of chess that preserves the fundamental movement rules of the original game while introducing a unique capture mechanic, where captures trigger explosions that remove surrounding pieces. This combination results in a game that is both familiar and strategically distinct.

2.1.1 Justification for Chess Variant Selection

There were several motivations for selecting Atomic Chess as the focus of this project. Firstly, the explosion mechanic accelerates the rate at which pieces are removed from the board, often leading to significantly shorter games compared to classical chess. This characteristic allows for a higher throughput of self-play games during training, which is particularly advantageous when computational resources are limited. The increased number of training iterations per unit time can lead to faster learning progress and more rapid model refinement. Secondly, because Atomic Chess shares the same piece movements as standard chess, a baseline implementation of classical chess could be reused to verify the correctness of the move generation system using established testing methodologies. Once validated, the Atomic Chess–specific rules could be layered on top, as detailed later in Section 2.1.2. This approach simplifies implementation by isolating the variant-specific logic.

Finally, while Atomic Chess is relatively well-known and supported by major chess platforms such as Lichess, it lacks the depth of established theory found in classical chess. The absence of extensive opening books and end-game tablebases makes it a particularly suitable environment for a self-learning engine, as it must develop strategies independently without relying on pre-existing human knowledge or heuristics.

2.1.2 Atomic Chess Rules

According to the rules outlined by Lichess [3] as well as further verification through testing scenarios on Lichess' Atomic Chess mode, pieces in Atomic Chess have the same moves as in regular chess. During a capture, both pieces are removed from play, and an explosion occurs on the ending tile of the capturing piece, which removes all non-pawn pieces in the 8 adjacent tiles in a 3×3 area as shown in Figure 2.1. A capture may neither explode your own king nor may it reveal a regular check on your king, unless it also explodes the enemy king in the process as shown in Figure 2.2. If a player is in check they can either evade it through regular means, but also by exploding the enemy king or the checking piece. Since both pieces involved in a capture explode, kings can not capture. This also allows kings to occupy tiles next to one another since the other king is not a threat. During en passant, both pawns involved in the capture are taken out of play, and the explosion is centred around the tile the attacking pawn lands on, shown in Figure 2.3.

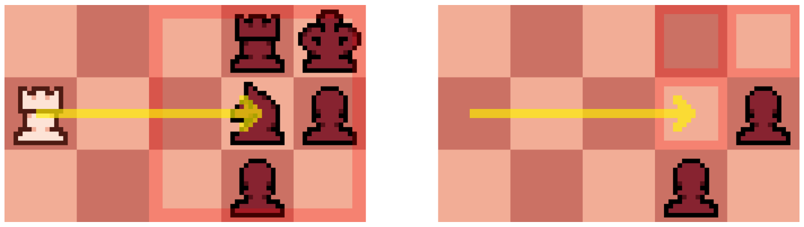


Figure 2.1: Example of white winning a game by the unique explosion mechanic in Atomic Chess. The move results in the capture of black's knight, king and rook while leaving the pawns unaffected.

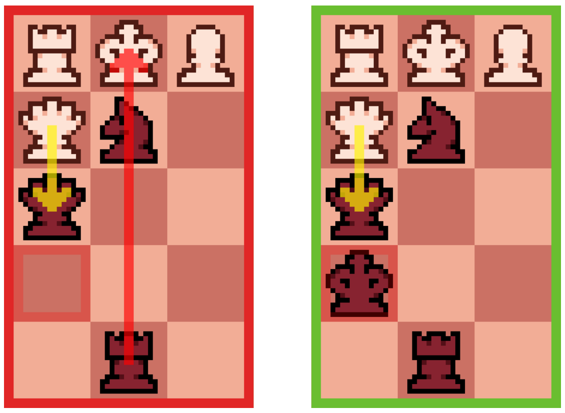


Figure 2.2: The left picture showcases an illegal move, since capturing the black queen would result in removing the black knight, in turn opening up a check on white's king. The right showcases a very similar situation, differing only by a black king, where the same move would be considered legal since it captures black's king.

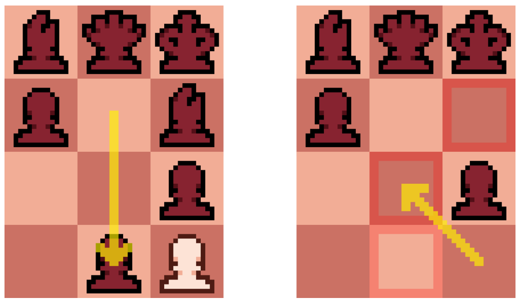


Figure 2.3: Example of white performing an en passant move. The move results in an explosion around the ending tile of the attacking pawn.

2.2 Chess Engines

Chess engines are computer programs made to play chess. For an engine to be able to play well, it needs to know what valid moves there are in a given chess position. To do this, the engine requires a way to represent the board to the computer. In this project bitboards are used for this. The engine then needs a way to generate what valid moves are available from a given board position. Relevant theories to this will be described in Section 2.5. With these parts, a typical chess engine searches for the best moves by simulating legal moves for both players and evaluating the resulting positions from these move sequences.

Standard chess is a well studied game and therefore several heuristics to evaluate board positions have been developed. These heuristics are also used by engines in conjunction with different algorithms to rank future moves based upon their strength. Examples of chess engines that use established chess heuristics are Deep Blue and Stockfish [1][4]. Atomic Chess does not have established strategies and heuristics to the same extent, making a different method of evaluation useful, which brings us to AlphaZero.

AlphaZero, introduced by Silver et al. [2], employs a generalized learning approach based on reinforcement learning and self-play. The system was designed to master the games of chess, shogi, and Go without any prior knowledge beyond the rules of the game.

AlphaZero utilizes a neural network to search and evaluate chess moves and positions. The neural network is given the state of the chess board and returns a value and move policy. This is used to guide the Monte Carlo Tree Search algorithm, further discussed in Section 2.3. The value of a board position is a value between -1 (black win) and 1 (white win) and the move policy is a list of probabilities corresponding to different moves.

To train the neural network, AlphaZero plays games against itself (self-play) and trains the network on these games. Once a game of self-play reaches a result, a data set is created based on the resulting search tree. This is then compared to new predictions on those positions and used to adjust the network to bring future predictions closer to the actual result (explained further in Section 2.4.2). As the process is repeated the neural network gets better at evaluating moves and board states which eventually, over many training iterations, results in AlphaZero discovering novel strategies beyond what humans understand.

A key distinction between AlphaZero and its predecessor, AlphaGo Zero, is its ability to generalize across multiple board games using the same learning framework. The results presented in the paper demonstrate that AlphaZero surpassed Stockfish, one of the strongest traditional chess engines, within a matter of hours of training.

The success of AlphaZero underscores the effectiveness of self-learning AI in complex decision-making tasks. By removing the dependency on human expertise, AlphaZero exemplifies the potential of reinforcement learning to discover novel and highly efficient strategies in strategic environments. Its approach has since influenced further AI research within non-deterministic board games such as a simplified version of Chinese Dark Chess where it converges towards optimal play [5]. Additionally, AlphaZero also has influence in broader AI research such as the prediction of protein structures with AlphaFold [6], although not using the same algorithm or network.

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) was not only used in AlphaZero but is a common search algorithm used in decision-making problems, particularly in games with large state spaces and high branching factors. Unlike other search algorithms like minimax, which rely on exhaustive exploration of the game tree, MCTS uses a probabilistic approach to balance exploration of new moves and exploitation of known promising moves. This makes it particularly well-suited for games like chess and its variants, where the number of possible positions is too large to evaluate exhaustively.

MCTS was chosen for this project because of its proven success in self-learning chess engines like AlphaZero. Its ability to dynamically balance exploration and exploitation makes it ideal for discovering strategies in a domain with limited prior knowledge, such as Atomic Chess. By combining MCTS with neural network-guided evaluations, our engine can efficiently navigate the game tree and develop a deep understanding of the variant's unique dynamics.

2.3.1 Standard Monte Carlo Tree Search

The implementation of MCTS used by AlphaZero features some key differences to the standard implementation. To better understand these differences, it is important to establish how standard MCTS works. MCTS is a tree search algorithm where each node represents a game-state and contains some statistics. It starts off with the root node which represents the current game state and performs search iterations according to a budget, which can be a set number of iterations or a time limit. Each MCTS search iteration operates in four main phases [7]:

1. Selection: Starting from the root node, the algorithm traverses the tree by selecting moves that maximize the Upper Confidence Bound for Trees (UCT) formula.

2. Expansion: When the algorithm reaches a leaf node (a node that has not been fully explored), it expands the tree by adding one or more child nodes representing possible moves from that position.

3. Simulation: From the selected leaf node starts a rollout that plays out random moves until it reaches a terminal state in the game (a player winning or a draw).

4. Back propagation: After the simulation, the result is propagated back up the tree (from the leaf node that started the rollout), updating the statistics of each node along the path.

The selection phase uses the Upper Confidence Bound applied to Trees (UCT) which, for all valid moves possible from a position, is the following formula:

$$\frac{w}{n} + C \cdot \sqrt{\frac{ln(N)}{n}} \tag{2.1}$$

The $\frac{w}{n}$ term is the winrate of the move corresponding to a node, where w is the value and n is the visit count of said node. The N variable is the visit count of the current state. The two terms added represent how exploitative and explorative the search algorithm is. The win rate is the exploitative part whilst the second term is the explorative. The explorative term has a constant C, known as the exploration parameter, which affects how likely the algorithm is to explore different moves instead of exploiting the same high win rate move. The parameter is set depending on the application and how you want the search to behave.

2.3.2 AlphaZero's Monte Carlo Tree Search

The differences to standard MCTS mainly revolve around the fact that AlphaZero integrates a neural network into the search algorithm. Since neural networks will be covered later in Section 2.4, we can for the time being think of the neural network as an oracle f. Given the current state of the chess board (s) our oracle will give us a value for how good the current position is for the current player (v) and a move policy for how good each move from this position is (p):

$$f(s) = (v, p) \tag{2.2}$$

The selection phase for this version of MCTS uses a different formula known as Predictor UCT (PUCT). For each valid move a from state s, the PUCT formula

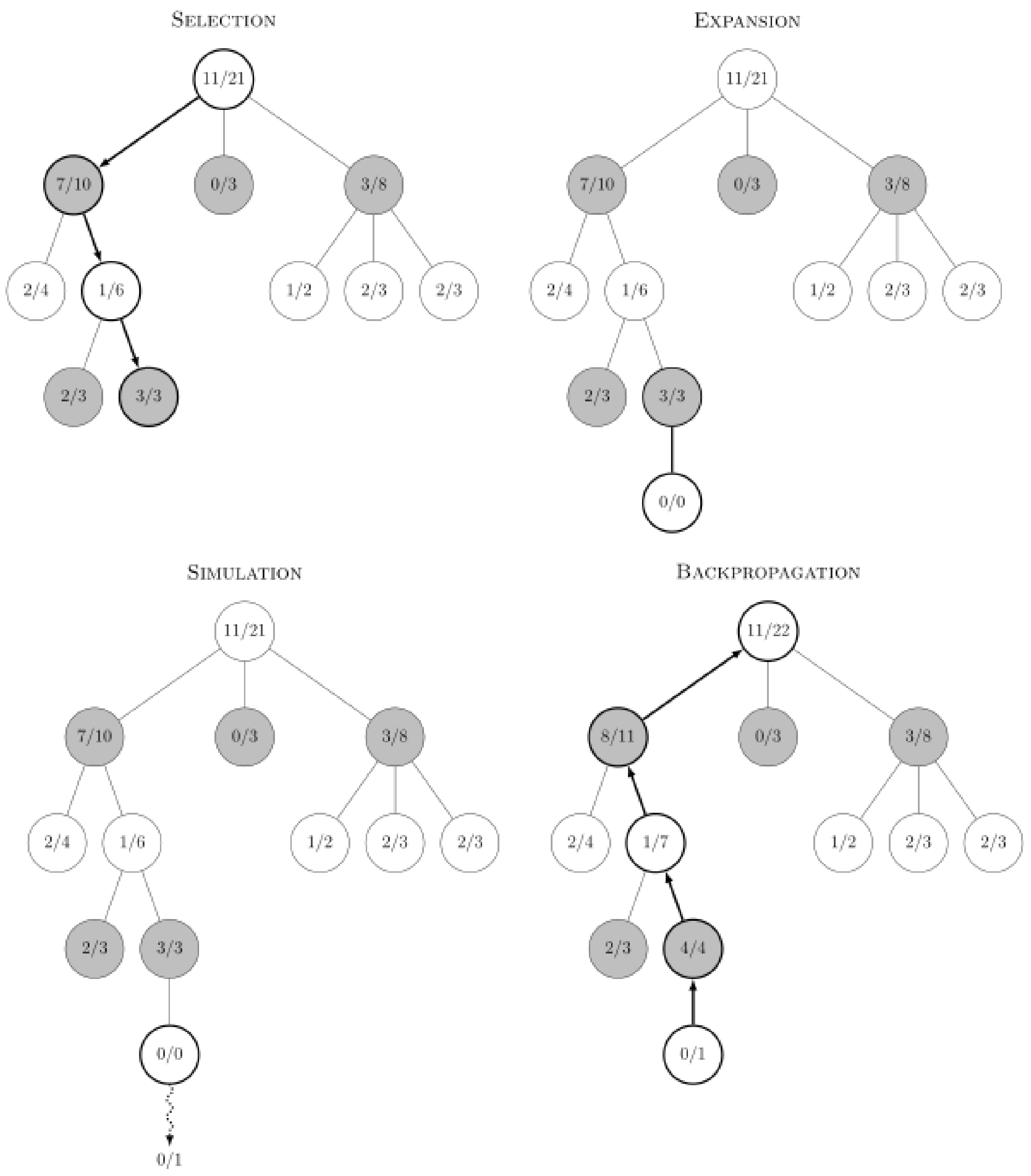


Figure 2.4: The different steps to Monte Carlo Tree Search visualised. Each node has a "visit count/value" pair. "MCTS-steps" by Robert Moss is licensed under CC BY-SA 4.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/?ref=openverse.

looks as follows [2]:

$$Q(s,a) + C \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)} \tag{2.3}$$

Compared to the regular UCT formula (Equation 2.1), there are some differences, mainly the addition of the prior probability $P(s,a)$ from the neural network. The $Q(s,a)$ term is the action value. The second term contains the exploration constant C, prior probability of action a as $P(s,a)$ and the visit counts N. Next in the expansion phase there are some big differences. When the search has reached a leaf node it expands the tree by creating new nodes for every valid move available. It then performs $f(s)$ on the current board state to get a value and move policy which it assigns to the corresponding nodes. After the expansion is done, there is no simulation with random rollouts, instead the value given by $f(s)$ is propagated. Once it has propagated up to the root of the search tree we have completed one iteration.

Figure 2.5 shows what the search tree looks like after two iterations from the beginning of the game. Firstly it expands the root node with every valid opening move for white and assigns a move policy value, P. Since it is the root it does not backpropagate any value and only increases the visit count of the root. Next iteration it chooses Knight to h3 as the most promising move and expands it with all the opening moves for black. It also assigns the board position a value of 0.1, which is then backpropagated to the root.

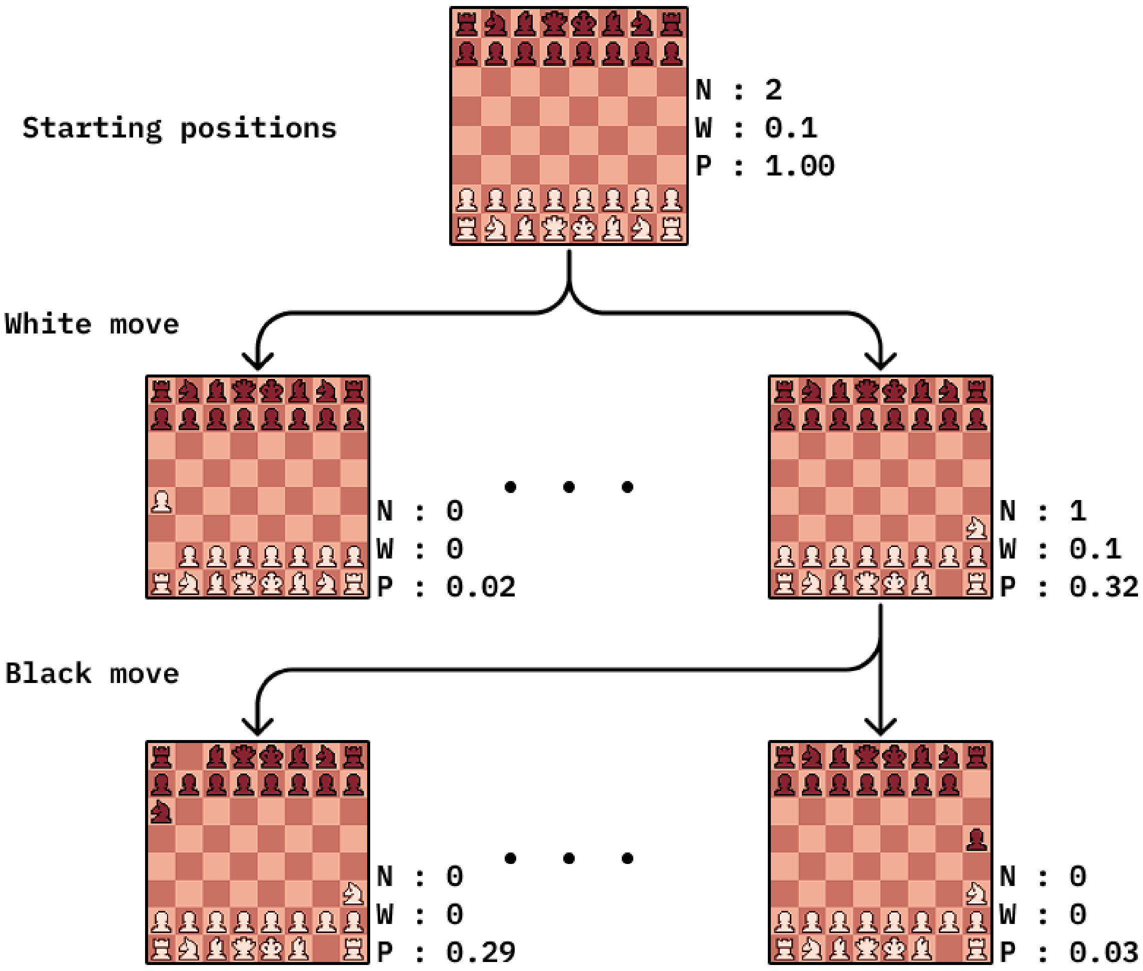


Figure 2.5: Example of tree after two iterations where N is visit count, W is value and P is policy.

Having introduced the search algorithm used by AlphaZero we can now hone in on the oracle $f(s)$ which is a neural network.

2.4 Neural Networks

Artificial neural networks are computational models that are inspired by the structure and function of neural networks in the biological brain and are an integral part of AlphaZero. The simplest neural networks are single artificial neurons, called perceptrons, that can be used to classify inputs. Looking at Figure 2.6 we see a perceptron that takes in the inputs x, and multiplies them with the weights w. These weights are used to tell the perceptron what parts of the input are important. For example if we want to classify an image the different inputs could be different pixels. With these inputs we compute the weighted sum and also add the bias θ, which is fed into an activation function.

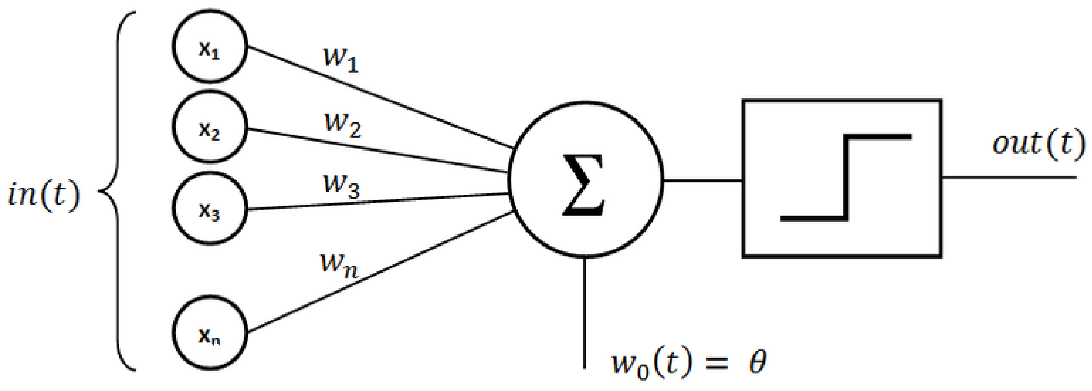


Figure 2.6: A visual representation of a perceptron. "Perceptron moj" by Mayranna is licensed under CC BY-SA 3.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/3.0/?ref=openverse.

In Figure 2.6 the activation function used is a simple step function that gives 1 as an output if the sum exceeds zero and otherwise outputs 0. AlphaZero utilizes tanh and rectified linear unit (ReLU) as activation functions in its neural network:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

$$ReLU(x) = max(0, x) \tag{2.5}$$

Tanh is a function that outputs a number between -1 and 1. ReLU is a function that sets any negative values to 0 and otherwise uses the original input [8].

Perceptrons on their own are not particularly complex so for a neural network to be “smart” we need to combine multiple perceptrons into layers. For a neural network we will always have an input and output layer but we might also have layers between them known as hidden layers. The structure of layers can be seen in Figure 2.7. Every perceptron in the first hidden layer is connected to each of the three inputs, and same for the second hidden layer but it is connected to the first hidden layer. The output layer has two perceptrons delivering 2 values, so the whole network implements a function: $\mathbb{R}^3 \to \mathbb{R}^2$.

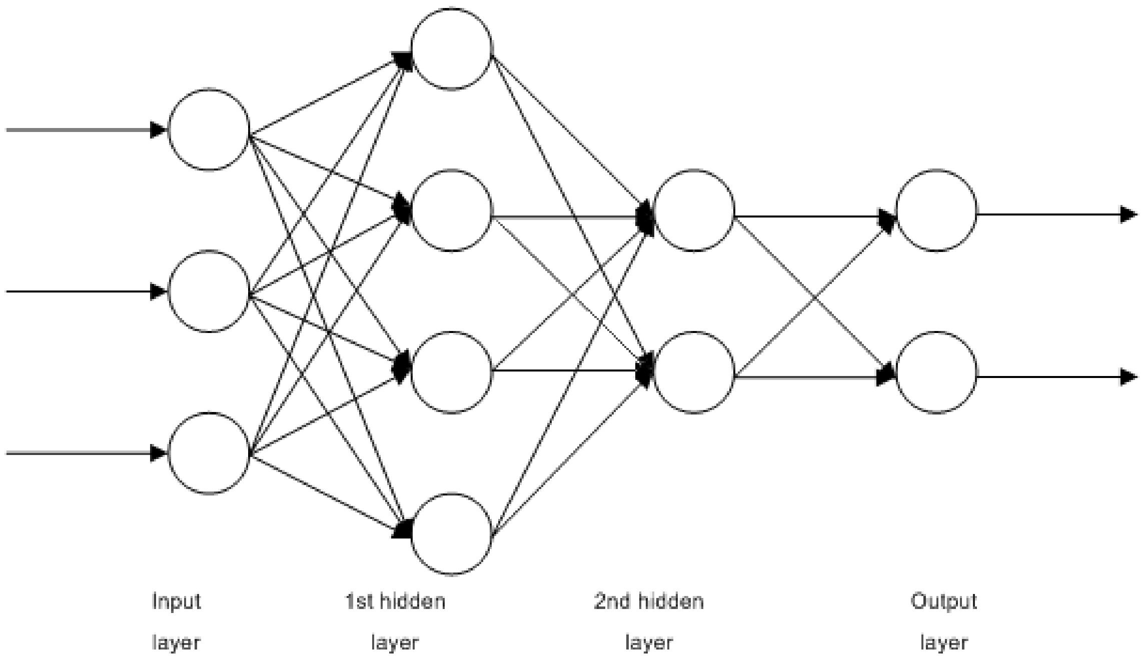


Figure 2.7: A visual representation of a multilayer neural network. "Multilayer Neural Network" by John Salatas is licensed under CC BY-SA 3.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/3.0/?ref=openverse.

2.4.1 Neural Network Layers

We may say that a neural network is a structured set of interconnected layers. The number and types vary between networks and applications. In the following sections, fully connected and convolutional layers are explained since they are relevant to chess engine networks.

Fully Connected Layers

The layers displayed in Figure 2.7 are so called fully connected layers. This is because the perceptrons in each layer has a connection to each perceptron in the previous layer. While this wiring structure is feasible for small networks, it has very poor scalability for more complex and deep networks such as those required for image identification, for example. If we want to identify an image that is 256×256 pixels, we need 65,536 inputs. Now say we have a fully connected hidden layer that has 20 perceptrons. This gives us 1,310,720 weights that we need to update when we backpropagate through the network. For tasks such as image recognition it has therefore become much more common to use convolutional layers.

Convolutional layers

In many tasks involving grid-like data structures such as images or chess boards, it is important to detect patterns that depend on the relative positions of neighboring elements. A convolutional layer is a type of neural network layer specifically designed for this. Rather than connecting every input to every output neuron as in a fully connected layer, a convolutional layer applies a small, learnable function called a kernel or filter over small regions of the input at a time [9].

In the 2D case, the kernel is a small matrix with a fixed size that moves across the input grid. At each position, it performs a calculation based on the input values it covers. The distance it moves between each step is called the stride. To ensure the output size remains consistent with the input, convolutional layers can use a technique called zero padding, which adds extra rows and columns of zeros around the edges of the input grid. This concept also generalizes to higher dimensions.

Multiple 2D grids can be forwarded through a convolutional layer together the resulting operations will treat the input as a tensor of size $x \times y \times num\_grids$. This allows for related information stored into equal sized 2D grids to be processed together, such as the RGB colour values of a coloured image. In this case, assuming we wanted set the 2D convolutions to be 3×3 in size, the effective size of the 3D convolution would be $3 \times 3 \times 3$ to account for the 3 RGB layers. Each convolution would then compute each kernel position using a set of $3 \times 3 \times 3$ weights to produce a final value.

The reason for their usage in fields such as image recognition and local pattern recognition in applications like board games is their ability to handle bigger inputs, unlike regular layers that would need a great number of perceptrons and connections and have a higher likelihood of overfitting and have difficulties learning meaningful patterns from training. Convolutional layers are a great fit for tabletop games like chess, played on a two dimensional grid since they can recognize patterns regardless of where they appear on the board, in other words, they have positional invariance.

1	0	0
0	1	0
0	0	1

Figure 2.8: Example of pattern detectable by 3×3 kernel. The diagonal forms a continuous line of values.

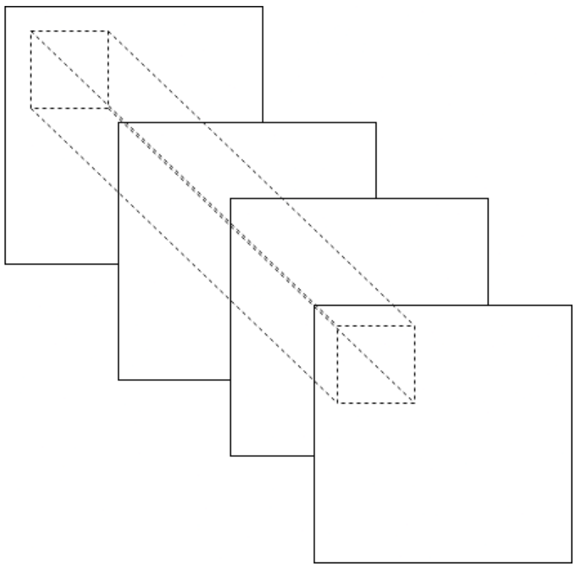


Figure 2.9: Example of convolutional layer with kernel where 4 grids have been forwarded together.

2.4.2 Learning Process of Neural Networks

Neural networks are not “smart” by design, in fact the weights start out with random values. To get “smart” it needs to learn by producing guesses and improving based upon them, also called reinforcement learning. To get an idea of the process we will first go through a general overview to then get into specifics about AlphaZero.

Neural Networks can be seen as a function that, when provided an input, should produce an expected output. This pair between input and expected output creates the basis for training the network. Giving the neural network an input should hopefully give us the expected output, but for a newly created neural network with randomized weights, this is highly unlikely. Comparing the difference between actual output with the expected output we can compute the loss, which is a measure for how “wrong” the predicted output from the neural network was [8]. The neural network is improved by reducing this loss, which is done by adjusting the network’s weights accordingly.

Each weight in the network contributes differently to the overall loss and during backpropagation, we compute how much a small change in each weight would affect the loss. This value is called the gradient for that specific weight. By backpropagating the loss through the network and calculating these gradients, we can determine in which direction to adjust each weight to reduce the loss. The amount the weights are changed is proportional to the magnitude of the gradient that was calculated. How much we move towards the direction which decreases the loss is also influenced by the learning rate, and the bigger the learning rate the more change in weight. This process of using training data to find out how “wrong” the neural network is repeats until we find an optimal solution to our problem. When deciding the learning rate we want to find one that is neither too high nor too low.

Figure 2.10 shows the downsides when using learning rates which are too high or too low. The weight θ_i and the effect of it on the loss function $l(\theta_i)$ are the axes. For the high learning rate example the weight is initialized low, but misses the global minima because of the slope and learning rate, and settles in a less optimal value. For the low learning rate example we only find the closest minima, which is not the global minima.

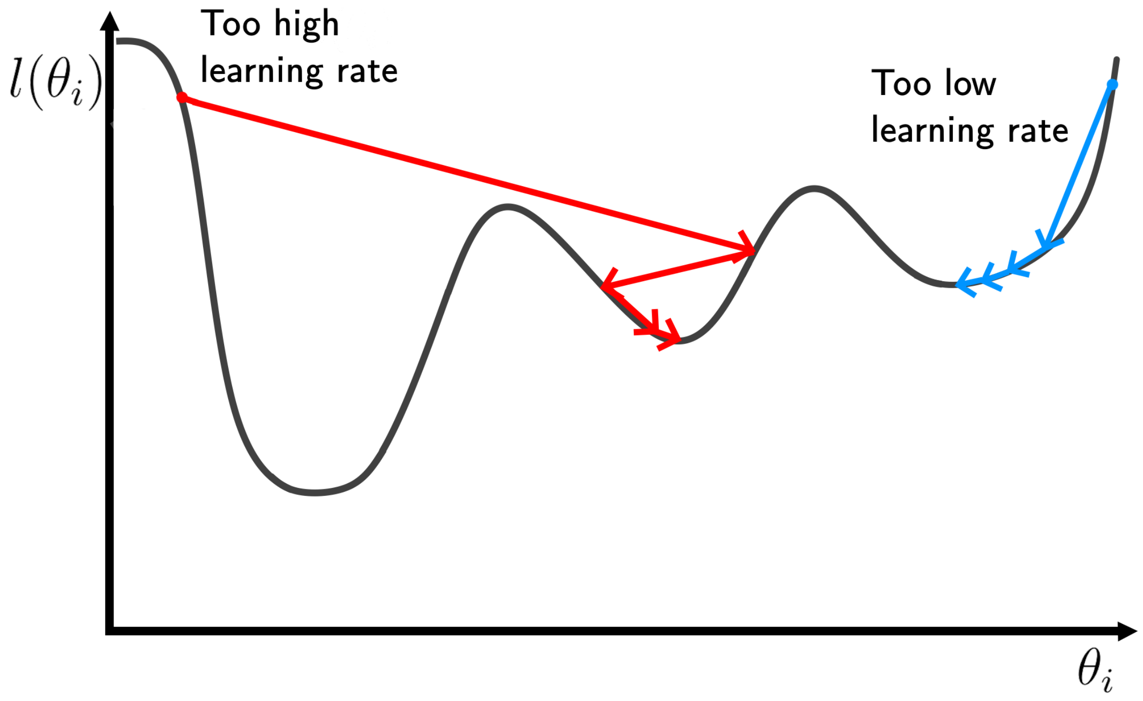


Figure 2.10: Visual representation of the difference between learning rate sizes.

If we look at AlphaZero's input and output, it takes in the current state of the chess board as the input and outputs a value and a move policy [2]. To get the expected output for the neural network we will use the results we get from the MCTS algorithm. A vector can be formed where each value is the number of visits to that node divided by total number of visits to all nodes from the starting position. With these values we can compare the output of the neural network with the move probabilities we get from our MCTS to compute a loss. This setup reflects a reinforcement learning framework, where the model learns to improve through self-play. AlphaZero's loss function utilizes a sum of cross-entropy losses between the MCTS policy prediction and a new model prediction with the mean-squared error between the value prediction with the game result. L2-regularization is also applied to prevent the weights in the model from endlessly growing. The loss function can be seen below in Equation 2.6:

$$l = (z - v)^2 - \boldsymbol{\pi}^T log\, \boldsymbol{p} + c||\theta||^2 \tag{2.6}$$

The first part of the loss function is the mean-squared error of the value v given by the neural network and z is the result of the self-played game, with it being 1 if the player who's turn it is won and -1 if they lost. The second part is the cross-entropy loss of the policy $\boldsymbol{p}$ given by the network. The vector $\boldsymbol{p}$ is multiplied by a transposed vector $\boldsymbol{\pi}$ which is the policy based on the tree search mentioned earlier. The last term is used for regularization for large weights and consists of the network parameters (the weights), θ, and a regularization coefficient, c. The coefficient determines how much to punish large weights to reduce overfitting.

AlphaZero addresses the issue of learning rate size by using different learning rates depending on how far it is into training [2]. At the start of training the learning rate is set to 0.2 and is then updated depending on how many times it has updated the weights in the network. After 100,000 updates the learning rate changes to 0.02, after 300,000 it changes to 0.002 and after 500,000 it changes to 0.0002.

2.4.3 Residual Networks

When backpropagating through deep neural networks we run the risk of getting vanishing or exploding gradients. As we travel backwards through the network to update weights the impact of the loss is either lessened or blown out of proportion due to the chain rule. To prevent this we can use something called a residual network, also known as ResNet. These networks introduce skip connections that bypass layers of the neural network and combine the output of the skipped layers with the input to the block, referred to as residual blocks. With this skip connection there is an alternate, less computationally heavy road backwards through the network that means earlier layers also get meaningful weight updates. This has proven to be effective at training and optimizing deep neural networks and is used in AlphaZero's network construction (discussed further in Section 2.4.4) [2][10].

Batch normalization is a widely used technique in residual blocks to stabilize and accelerate training in deep neural networks. It works by normalizing the inputs within a batch of inputs, adjusting them to have zero mean and variance based on the batch's statistics. This process reduces internal covariate shift, allowing the network to train faster and more effectively by maintaining more stable gradients throughout the layers [11]. In Figure 2.11 we see a residual block with a skip connection going above it. In the $F(X)$ path we have convolutional layers (Conv) and batch normalizations (BN) which are added before each ReLU function. After the second batch normalization, the skip connection is added to $F(X)$ before the next ReLU function which is the end of the residual block.

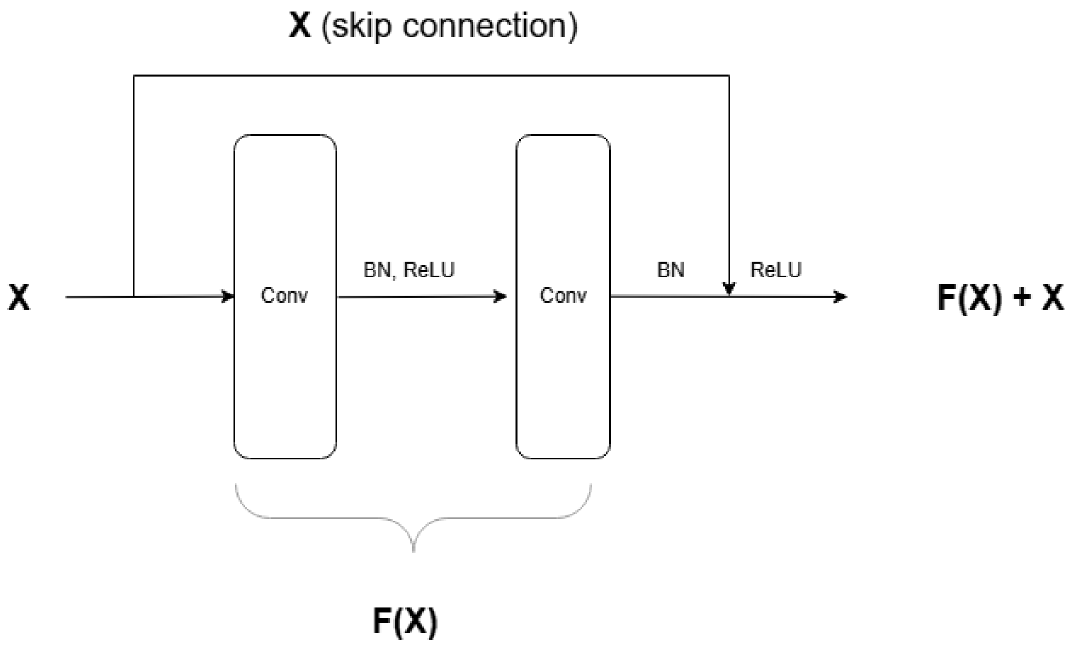


Figure 2.11: Visual representation of a residual block.

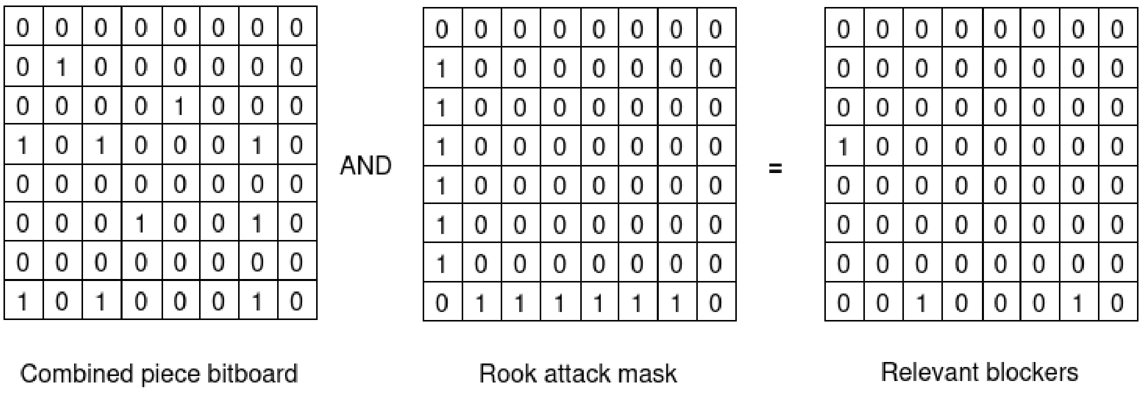
2.4.4 AlphaZero Network Construction

The Neural Network in use by AlphaZero as described in the Supplementary Materials [2] consists of a body, which is responsible for the majority of calculations in the network, which then feeds into two separate heads. The heads consist of a value head and a policy head, each of which are responsible for extracting features valuable for either the value or policy from the main body prediction and formatting them to become the value and policy output. The body consists of 20 parts, the first of which is a convolutional layer which is responsible for batch-normalizing the inputs into a range around 0 and rectifying all negative values to 0. This is followed by 19 residual blocks as seen in Figure 2.11. Each convolution layer applies 256 filters of kernel size 3×3 with stride 1. After the body it splits into the value and policy heads. The policy head applies an additional rectified, batch-normalized convolutional layer, followed by a final convolution of 73 filters. The value head applies an additional rectified, batch-normalized convolution of 1 filter of kernel size 1×1 with stride 1, followed by a rectified linear layer of size 256 and a tanh-linear layer of size 1.

2.5 Bitboards and Magic Number Move Generation

Bitboards and magic numbers are used to calculate the legal moves from any given position. Bitboards are an efficient way to represent chess positions using 64-bit integers, where each bit corresponds to a square on the chessboard. This representation allows for fast manipulation of piece positions using bitwise operations, rather than traditional array-based approaches for storing a 2D space like a chessboard. By leveraging the inherent parallelism of bitwise operations, bitboards enable highly optimized and computationally efficient legal move generation. Since there are multiple pieces and piece types in chess, each type of piece typically has its own associated bitboard.

The main advantage on the computation side is allowing for the pre-computation of possible moves for sliding pieces, making it possible to avoid evaluating whether each position the sliding piece could move to is blocked. Instead, since each bitboard is also a 64-bit number, all of them can be combined using OR operations into a single bitboard containing information regarding positions of all pieces and possible blockers. By then executing a bitwise AND operation on the combined bitboard and a mask of possible moves for the piece, keeping only an oversight of pieces capable of blocking the sliding piece. Since the bitboard containing information of all pieces can be computed once at the start of move generation for the turn, only the bitwise AND operation has to be performed to calculate all blockers making it extremely fast.



0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0
1	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0
0	0	0	0	0	0	0	0
1	0	1	0	0	0	1	0

0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0

Figure 2.12: Calculation of blockers for a rook in the bottom left corner.

At this stage, the only thing that remains is calculating which spaces the piece is able to move to. As shown in Figure 2.12, there are a total of 12 spaces which can be occupied by a blocking piece, meaning that in that instance, there exists 2^{12} or 4096 possibilities of blockers for a rook positioned in the absolute bottom-left square. This is a small enough number that it is possible to pre-compute all possible movements per blocker configuration and store them into a lookup table for later, meaning that the computation can be simplified to a simple value lookup making it extremely fast as well. This process can then be repeated for all 64 squares as well as once for both rooks and bishops creating a full set of lookup tables for all sliding pieces, with queens being able to use both rook and bishop lookup tables sequentially for their movements. After the lookup, an additional check is carried out to see if the target square is of the same colour as the piece the calculations are being carried out for, in which case the move is rejected as invalid.

After the allowed moves have been pre-generated, they can be stored into a hashmap which allows for compact storage and psuedo-constant lookup times. One further optimization that is often performed is utilizing a custom hashing algorithm utilizing magic numbers to avoid collisions. Since we can know all of the possible keys that will be input into the hashmap, it is also possible to pre-compute a magic number which will allow us to avoid collisions when looking up values, meaning we can skip the key comparison step as well as ensure constant time on lookups. The most common version of this hashing algorithm is to multiply the bitboard by the magic number, then keep only the most a set number of highest bits decided by the index position of the moving piece. Originally, these magic numbers were pre-computed using a brute-force search during development, due to each piece type and square needing their own number. These numbers have been widely disseminated onto the Internet since.

2.6 Parallelization

Parallelization is critical in this project due to the computational demands of training a self-learning chess engine. The two main processes that benefit from parallelization are MCTS and neural network training.

2.6.1 Monte Carlo Tree Search Parallelization

It is not trivial to effectively parallelize the MCTS algorithm in the case of a self-learning engine. One common strategy for pure MCTS is to do root parallelization, where multiple threads create separate MCTS trees in parallel and later sum the visit counts to get a more accurate result. However, this strategy will not work in the case of AlphaZero's modified MCTS implementation. The reason for this is that the random rollout step in the algorithm is switched out for a neural network that always values a position the same way. This results in multiple identical trees if multiple threads start from the same root, as the same path through the tree will always be chosen.

An alternative approach is tree parallelization, where multiple threads operate on the same search tree simultaneously. However, this can lead to performance issues, as threads tend to compete for access to the nodes with the highest PUCT scores, resulting in frequent contention. To address this, virtual losses are used, a proven technique for reducing such conflicts [12]. The core idea is to temporarily assign a "virtual loss" to nodes selected by other threads, artificially reducing their PUCT scores. This discourages other threads from choosing the same high-priority nodes, spreading the exploration more effectively and reducing the risk of redundant computations.

2.6.2 Training Parallelization

Similarly, neural network training involves processing large amounts of self-play data, which can be accelerated through parallelization. By distributing the workload across multiple CPU or GPU cores, the training process becomes more efficient, allowing for faster iterations and better utilization of limited computational resources. One example would be to utilize multiple threads to play individual games, meaning that the number of moves generated to train using could scale with the number of available cores on a CPU, assuming there is no bottleneck on the GPU.

Given the constrained hardware available for this project, parallelization is not just a performance optimization but a necessity. It enables the engine to complete more training iterations and play more games within a fixed time-frame, ultimately improving the engine's playing strength and the quality of the learned strategies.

2.7 Previous Bachelor Theses and Related Work

Projects similar to ours have been done as bachelor's thesis before, and we had access to two of those reports [13] [14]. By examining the challenges, limitations, and insights from earlier work, we can identify strategies to improve the current project and avoid common pitfalls. Note that these projects made engines for different chess variants, differing in complexity to Atomic Chess.

Below, we summarize the key takeaways from previous bachelor theses' findings that are relevant to this project.

- **Challenges and Limitations:** Both projects encountered significant challenges related to overfitting and computational resources. Overfitting was a recurring issue, where models performed well on training data but poorly on validation data. Techniques such as weight decay, dropout layers, and increasing dataset size were suggested as potential solutions. Additionally, computational constraints were a major bottleneck, particularly for training deep neural networks and running MCTS with high iteration counts. These limitations highlight the importance of balancing model complexity with available hardware and optimizing training efficiency.

- **Neural Network Architectures:** One project successfully implemented a convolutional neural network (CNN) with a reduced number of hidden layers (13 layers instead of AlphaZero's 19) to manage computational constraints. Despite the simplification, the network demonstrated good learning capabilities, suggesting that smaller-scale projects can achieve meaningful results with simpler architectures.

3

Implementation

This chapter aims to explain the components of the final product and how they are constructed. Firstly, an overview of the entire implementation is given. The chapter then goes into more detail explaining the individual components. The chapter will build on most of the concepts that were introduced in Chapter 2.

3.1 Implementation Overview

The architecture consists of four core components: a game representation system, a neural network, a Monte Carlo Tree Search (MCTS) algorithm, and a training framework. The board states and the move generation logic are implemented using bitboards and allows the program to compute legal moves for Atomic Chess.

The engine was developed in C++, primarily due to the language's strong performance characteristics and its compatibility with Libtorch, the C++ distribution of PyTorch. C++ also has a wide variety of other libraries which are made with performance in mind, which makes it a fitting language for a project where the importance of execution speed is paramount.

The network made with LibTorch adopts a residual network architecture similar to that used in AlphaZero, but scaled down in size. Our implementation processes board states through 12 residual blocks, producing two outputs: a policy vector where each value is an evaluation by the network on how advantageous that option would be if played, with the entire vector being normalized. representing the probability distribution over legal moves, as well as a scalar value estimating how advantageous the current position is for the turn player.

To guide decision-making during gameplay, the engine employs Monte Carlo Tree Search. The search algorithm uses the PUCT formula to balance exploration and exploitation and performs backpropagation to update node statistics based on network evaluations. Batch processing is used to accelerate inference across multiple nodes.

Training is organized into iterations where the engine plays games against itself, collects positions from the game in a replay buffer, and updates the neural network based on a batch of sampled data from the replay buffer. To train, debug, and evalutate the models and more, several modes were developed to help perform these tasks. These are the main modes:

- **Arena**: for head-to-head matches between two models.
- **Perft**: for validating move generation correctness.
- **Dojo**: for self-play training.
- **FEN**: for starting games from specific board positions.
- **Lichess-UCI**: Implements UCI (Universal Chess Interface), this is used to deploy the bot on Lichess.org.

Additionally, a graphical interface was introduced where the board state can be seen, both during training and of course when playing against the engine.

Now we will be diving deeper into the individual components and their implementation details.

3.2 Game Rules and Move Generation

Move generation is a foundational subsystem of the chess engine, responsible for enumerating all legal actions from a given board state. This component is used in conjunction with the neural network, which returns the move policy with the moves it thinks have the highest win probability. The move generator's list of legal moves is then used to invalidate all the illegal moves that the neural network suggested, leaving a complete move policy.

The implementation was carried out in two distinct phases: initially supporting standard chess to ensure correctness and ease of verification, followed by the incorporation of Atomic Chess mechanics. This staged development strategy enabled reliable testing and debugging prior to introducing the additional complexity of the atomic variant, and was feasible as the atomic features are simple to add on top of regular chess rules.

3.2.1 Bitboard-Based Representation

Bitboards are used to represent the board state, where the 64 squares of the board are represented using 64-bit integers. Separate bitboards are maintained for each player's pieces, so there are 12 bitboards in total, 6 unique piece types per player. Each bitboard also encodes the occupancy and attack patterns for every piece type on the board.

3.2.2 Standard Chess Move Generation

Legal move generation is sorted by how the pieces move, with specialized logic for sliding, non-sliding, pawns and castling.

Sliding Pieces (Bishop, Queen, Rook) : Sliding piece attacks are computed using directional occupancy masks and magic bitboards, explained in Section 2.5.

Meaning it works by multiplying the occupancy on relevant squares by a magic number. Then it shifts the result, and using it as an index into a precomputed lookup table. This table stores all possible attack patterns for each unique blocker scenario along a given direction.

Non-Sliding Pieces (King, Knight): Kings and Knights always have the same movement pattern, and cannot be blocked by other pieces in the same way, except if allied piece is on the square already. But this means that the non-sliding piece attack sets are stored in static lookup tables that are indexes by their square.

Pawns: Pawns are more complex as they have many unique features. They require asymmetric logic for forward movement as the piece only moves in one direction relative to the player. Diagonal captures, are easy to compute as they work the same way as a non-sliding piece. Promotions is handled using separate logic that simply checks whether a pawn has reached the last rank. Similarly, single and double advances are checked depending on rank. Lastly, en passant legality is verified after move generation to ensure it does not expose the king to discovered checks.

Castling: Castling rights are validated via bitwise tests: verifying that both the king and the rook in question have not moved, that the path is unoccupied, and that intermediate squares are not under attack.

3.2.3 Handling Piece Constraints

Check Detection: All opposing piece attacks masks are checked to determine if the side to move is in check or not. When a king is under check, its legal moves are restricted to three actions. These are either capturing the checking piece, blocking its line of attack, or moving the king to a non-attacked square.

Pin Detection: Pinned pieces are identified by scanning all straight lines extending from the king's square, in the same manner a queen can move. If a friendly piece lies on the same line between the king and an opposing bishop, rook, or queen, and there are no other pieces blocking the path, it is considered pinned. Such a piece is then restricted to moving only along that line, if at all.

Legal move generation begins with a set of pseudo-legal moves, which are then filtered by applying the constraints described above. The filtering logic then excludes moves that violate these constraints.

3.2.4 Atomic Chess Extension

After implementing and verifying that the standard rules of chess detailed above worked, the Atomic Chess additions were added.

Explosion Capture: After a capture, the piece that made the capture is removed. Then a 3x3 explosion mask is applied to the area. The board state is updated accordingly by clearing the relevant bits in each affected bitboard, leaving pawns unaffected, as they are immune to nearby explosions.

Modified Legality Checks: Captures that would result in the player's own king being removed are disallowed. This requires a re-evaluation of the resulting board state post-explosion, ensuring that all remaining pieces are valid under the variant's rules.

Pinned pieces are allowed to move in a way that checks their own king if the resulting move would result in capturing an opposing king through an explosion. This requires the logic handling pinned pieces to be re-evaluated as pinned pieces could never be moved in regular chess.

Impact on Attack Evaluation: Attack generation routines were extended to consider the effects of explosions on adjacent pieces, influencing king safety and positional evaluation.

3.3 Monte Carlo Tree Search Implementation

Monte Carlo Tree Search (MCTS) was chosen as the engines decision-maker because of its use in AlphaZero, where they proved that it is an effective implementation when using neural networks and in environments with high branching factor, such as a complex game like chess that can alter the board state drastically in just a few moves. The implementation is based on the PUCT formula similar to AlphaZero's implementation.

3.3.1 Monte Carlo Tree Search Structure

A tree-node data structure is the core of the MCTS, which represents a node in the game state search tree. Each node maintains the following:

- A reference to its parent and a list of children.
- The move that led to the node.
- The game state the node represents
- PUCT statistics: total accumulated value, visit count, and the prior probability provided by the neural network.

The MCTS steps are similar to what was described in Section 2.3.2. In our implementation, the Monte Carlo tree is constructed by first initializing a root node and then performing a series of iterations, where each iteration executes the following steps.

1. **Selection:** Selection works by, at each level of depth, selecting the child node corresponding to the move with the highest PUCT score, calculated as:

$$\text{PUCT}(a) = \frac{w}{n} + p \cdot C \cdot \frac{\sqrt{N}}{1+n} \tag{3.1}$$

   where n is the visit count of move a. The total visit count of the current parent node is N, the average value (win rate) of simulations through move a is then $\frac{w}{n}$, the prior probability of move a provided by the neural network is p and C is the exploration constant.

2. **Expansion and Evaluation:** When the search reaches a leaf node, an expand function is called. This function performs the following steps:

   - Generates all legal moves from the current game state.
   - Evaluates the position using the neural network, which returns a policy vector (prior probabilities) and a scalar value (position evaluation).
   - Applies softmax normalization to the policy outputs.
   - Initializes child nodes for each legal move using the normalized priors.

3. **Backpropagation:** Once a leaf node has been expanded and evaluated, the scalar value is propagated back through the visited path. During this phase, the total value and visit count for each node along the path are updated. The value is inverted from positive to negative at each step to reflect the opponent's perspective.

3.3.2 Batching Selections

In order to leverage the strength of GPU parallel processing, some additional modifications to the original MCTS algorithm were made. In the original algorithm, only one node could be selected at a time and have its game-state be processed by the neural network. This however, does not leverage the parallel nature of the GPU. Instead we want to send a big batch of multiple game-states to the network and evaluate them all at the same time. So by running multiple selections and batching their game-states for a batch evaluation, the search will be much faster on a GPU.

To make this change work, already selected nodes are skipped during the selection phase to not evaluate duplicate positions. But this introduces an issue where all selections will try to go down the same path in the tree. This is resolved through the use of virtual loss.

3.3.3 Final Move Choice

Once exploration has ended, a move suggestion can be extracted from the tree structure. Two different strategies are implemented for final move selection and are used depending on the circumstance. The first strategy is choosing the node with the highest visit count, pure exploitation. This is used during real play against other opponents, because we want to play what the neural network thinks is best. Nodes having high visit counts indicate that the move is likely to be good because it has been selected more often than others using the PUCT-formula.

The second strategy is proportional sampling, which results in more exploration. This is used during self-play to encourage variations in the training games. The strategy works by looking at all the root's children's visit-counts and divide them by the total visits for all the children to get a list of probabilities. Then choosing the move randomly based on the probabilities. For instance, if you are at one node with three child nodes, with 70, 20 and 10 visit-counts respectively. Then we will randomly pick between them, with the nodes having a 70%, 20% and 10% chance of being chosen respectively. This will most often result in the best move being picked, but allows for less visited nodes to be expanded and explored as well.

3.4 Neural Network

The neural network used in the engine is a deep convolutional neural network, similar in structure to that of AlphaZero, but scaled down in size.

Network Input

To process the game state with the neural network, the board must first be encoded as tensors. Each of the 12 bitboards are converted into an 8x8 tensor. These tensors are then stacked along the channel dimension, forming a final single tensor with 12 channels, similar to how colour channels work in image processing. This is illustrated in Figure 3.1. This final tensor is the input to the neural network without any other information.

Additionally, because chess is both spatially and rule-symmetric, we can mirror the board vertically and swap the colours of the pieces when it is black's turn. This ensures that the network always sees the position from the perspective of the player to move. As a result, the network does not need to learn separate strategies for white and black, or learn to shift its perspective.

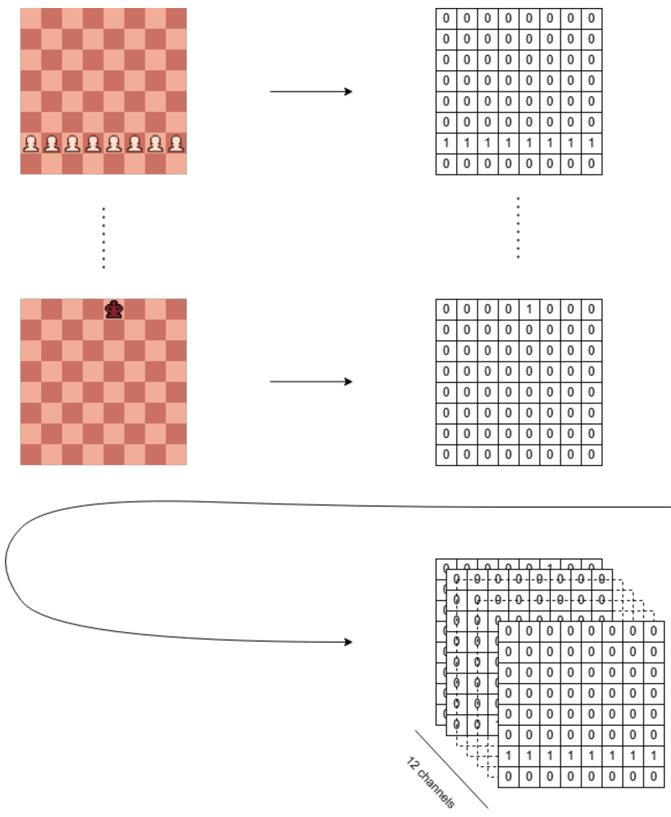


Figure 3.1: Conversion from a board position to a tensor.

Network Structure

Once the game-state has been converted into a tensor and passed into the neural network as an input. The input tensor of size $8 \times 8 \times 12$ first goes through an initial convolutional layer consisting of 128 filters, each with a kernel size of $3 \times 3 \times 12$ and a stride of 1, which produces a tensor of size $8 \times 8 \times 128$. This is followed by 12 residual blocks, each with 128 filters of kernel size $3 \times 3 \times 128$ and stride of 1, outputting tensors of the same size as the input, as illustrated in Figure 2.11. Due to the resource constraints of this project, a shallower network was implemented than the 19 residual blocks used by AlphaZero as was discussed in Section 2.4.4.

Lastly, the network needs to output the policy and value heads. These are responsible for guiding the search and evaluating the game state. The policy head begins with a 2 filter $1 \times 1 \times 128$ convolution that reduces the number of channels to 2, yielding a $8 \times 8 \times 2$ tensor that is flattened to a vector of size 128. This is followed by a fully connected layer that outputs the move policy head with all the move probabilities, where the element at each position in the output represents a move. For instance, the first element in the output (a1a2) represents a move for square a1 to a2 which contains the likelihood that this move is the best move. The value head starts similarly to the policy head with a $1 \times 1 \times 128$ convolution to reduce the channels to 1, followed by a ReLU activation. The resulting tensor is then flattened to a vector of size 64, and passed through a ReLU-activated hidden layer outputting the same size, before being reduced to a single value through a last perceptron. The final output is a single scalar value representing the expected outcome of the game from the current position. This separation of the network value and policy head allows the network to perform both move prediction and position evaluation. The overall structure of the network is illustrated in Figure 3.2.

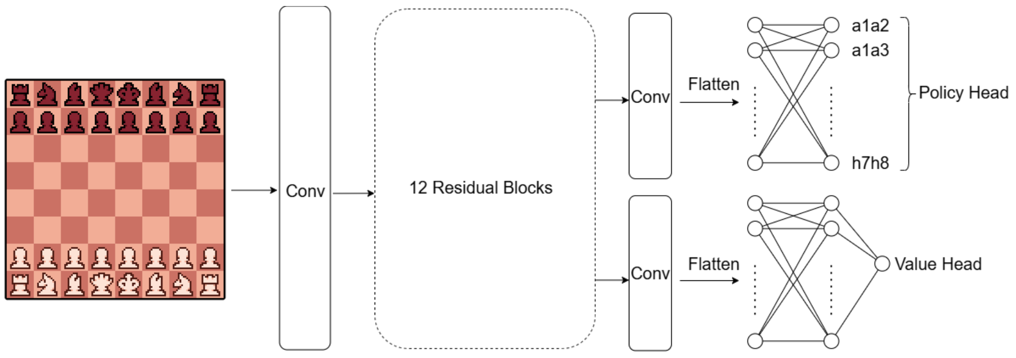


Figure 3.2: Structure of the neural network.

3.5 Training Setup

The training setup, like most components, draws inspiration from AlphaZero. The iterative process goes through two main phases, generating training game by self-play, then using this data to update the neural network.

3.5.1 Self Play

During training, the engine goes through multiple training steps. For every training step, the engine uses self-play to generate a set amount of training data. By using proportional sampling in self-play, see Section 3.3.3, we ensure a diverse set of training games. Each data sample consists of a chess position, the MCTS probability distribution for that position and the final outcome of the game.

By training the network on these data samples, the model should gradually improve as more generations are completed. The assumption here is that the MCTS move distribution and final outcome is more accurate than the output from the network. So by having the model continuously try to mimic this, it should improve a small amount each training step.

3.5.2 Calculating Loss

The loss between a data sample and the model's prediction is calculated in two parts. First, the value loss is calculated to determine how accurately the model can predict the final outcome of the game. This is calculated as the mean squared error between the final game outcome (z), and the model's prediction (v), where a 1 represents a win, 0 a draw, and -1 a loss for the acting player. The term then becomes:

$$(z - v)^2 \tag{3.2}$$

Second, the policy loss determines how accurately the model can predict what the best moves are. This is calculated as the cross-entropy between the MCTS move probability distribution and the model's policy head. This term becomes:

$$-\boldsymbol{\pi}^T log\, \boldsymbol{p} \tag{3.3}$$

These two values are then added together and the resulting loss function is written in Equation 3.4.

$$l = (z - v)^2 - \boldsymbol{\pi}^T log\, \boldsymbol{p} \tag{3.4}$$

A Stochastic Descent Optimizer is used to optimize the model. The optimizer is originally set with a learning rate of 0.01 and a momentum of 0.9. The optimizer is also set to a weight decay of $1e^{-4}$.

The loss Equation 3.4 does not include a term responsible for L2-regularization unlike the formula used by AlphaZero, as seen in Equation 2.6. This is because the weight decay parameter in the optimizer will ensure that L2-regularization is carried out each time the optimizer is used.

3.5.3 Replay Buffer

After a batch of training data is created, it is stored in a replay buffer of training samples. By using a buffer of training examples, the model can sample from a diverse and robust dataset, not only the latest games. Over time, the oldest data is overwritten to prevent the model from training on outdated training data. Another feature added to prioritize newer training data is "prioritized experience replay" [15]. By using this, newer samples are prioritized when sampling from the replay buffer.

The size of the replay buffer varied during development but was set between 100,000 to 200,000 samples. For each training step, a mini-batch of 4,096 is sampled from the replay buffer and used to update the model, similar to what AlphaZero did [2]. For each of these training steps, about 2,056 additional samples are generated through self-play and added to the buffer.

3.6 Parallelization

Parallelization enables multiple training games to run concurrently, significantly increasing the speed at which data samples can be generated. At the start of training, a variable number of permanent worker threads is spawned. All workers independently play games and, at the end of each game, create samples and add them to the common replay buffer. Once enough samples (per training step) have been created, the first worker to end its game is responsible for launching the training routine that updates the weights of the network, at which point all other threads remain idle. Importantly, workers may still have ongoing games while weights are updated, but they are never selecting moves, i.e. performing MCTS due to the back-propagation algorithm not being threadsafe.

In practice, this was simply done by using a reader-writer mutex on the actual model object, where updating the weights of the network is considered a writing action, and selecting a move is considered a reading action. In essence, all workers shared a single model, and updates to the weights of the neural network were coordinated in a round-robin fashion.

4

Method

This chapter describes the methods used to train and assess the strength of our engine.

4.1 Training Process

Training a large convolutional neural network and generating self-play games is very computationally costly, therefore, we opted to train only a single neural network model and not try any other architectures. Neural network weights between different architecture can not be reused or transferred to a different network structure. If we had explored multiple different architectures, then we would essentially be wasting large amounts of time computing training models that would later be discarded, so we decided against this.

Instead we trusted the general neural network structure that AlphaZero had described and stuck with the scaled-down version of it described in Section 3.4. Therefore, the majority of the time spent after creating an initial working training pipeline was spent refining the pipeline as opposed to hyperparameter tuning the network. The main adjustments to the training pipeline are the following:

Multithreading optimizations: The self-play and training loops were better optimized to better utilize the available hardware and reduce bottlenecks, resulting in a higher throughput of games. The primary changes were running the self-playing games in parallel and letting an idle thread start the training routine without waiting for the others to finish playing games.

Resizing of replay buffer: The size of the replay buffer was altered in the middle of training from 100,000 to 200,000, helping to balance the trade-off between data freshness and remembering long-term strategies.

Adjustments to learning rate: The learning rate was also tuned manually during the training to prevent stagnation and continue the learning effectively.

One problem with this manual tweaking and altering of the training pipeline is that it makes tracking progress more difficult as there are more changing variables. Making it difficult to pinpoint exactly what is contributing and the magnitude of the contribution to the learning process. It is also hard to quickly benchmark if the changes made were in fact an improvement as training is a very slow process and it takes hours before a noticeable difference in playing strength can be observed. Therefore we mostly tweaked based on our hypothesis around the previously mentioned trade-off between data freshness and remembering long-term strategies.

4.2 Evaluation of Models

Throughout the training, multiple training checkpoints were made between training iterations. To evaluate the strength of these models and also the final strength of the best model, two methods for evaluation are used.

4.2.1 Internal Evaluation

To compare our different models an arena mode was developed. In this mode two models play against each other for a set number of games. The first model plays as white and the second model as black. The number of wins for white, black and the amount of draws that the games result in are then outputted. The set number of games per side was set to 500, then the sides were reversed for another 500 games, to create 1,000 games between two models in total. This makes it easy to calculate the win rate between the models on white / black / both.

4.2.2 External Evaluation

To evaluate the final model, the chess engine was deployed on Lichess.org on a dedicated bot account. This was done with the chess engine mode Lichess-UCI, where a computer runs our engine and upon receiving a match request from Lichess using the Universal Chess Interface (UCI) it accepts the request and plays the game. In this mode the engine always plays with 100% exploitation. The engine was hosted on a consumer grade PC with an Intel i5-12400f CPU and a NVIDIA 3060TI GPU. The MCTS-iterations were set to 50,000, which results in a thinking time of approximately 10 seconds per move.

Other players and chess engines with established ELO-ratings can be challenged from a bot account, which allows us to get an ELO for our engine. Bot accounts are however not allowed to queue for the standard “find a game” function, which limits the players we could find to those who were on the Atomic Chess top 200 leader board. Additionally, there are not many other bot accounts on Lichess that play Atomic Chess at an intermediate level, with most playing either on a very high level or a very low level.

Lichess also provides data as to where the bots ELO-rating ranks in terms of the entire active playerbase, which is a valuable metric better at clarifying its playing strength rather than a somewhat arbitrary ELO-rating.

4.3 Hardware Utilization and Cost

To handle the computational demands of training deep reinforcement learning models we utilized cloud-based GPU virtual machines (VMs) from TensorDock [16]. TensorDock is a cloud computing service specializing in on-demand GPU instances well-suited for machine learning workloads.

We rented two VMs, one was mostly running the current best available version of the training pipeline, ensuring uninterrupted progress toward a final model. A second VM was periodically brought online to test modifications to the training pipeline or to evaluate experimental configurations. This approach allowed us continue to improve the training without risking ending up without a final satisfactory product.

To optimize costs we powered VMs on and off as needed. Because TensorDock charges different hourly rates depending on whether a VM is active or idle, this flexibility helped us stay within the projects budget constraints. The total target expenditure for the entire project was 3,000 SEK The rented hardware configurations and their associated costs are shown in Table 4.1.

Table 4.1: Rented hardware and associated costs

Unit	**GPU**	**CPU**	**SEK/h (On)**	**SEK/h (Off)**
1	NVIDIA RTX A6000 PCIE 48GB	Intel Xeon Gold 6226 (14 cores)	4	0.1
2	NVIDIA A100 SXM4 80GB	AMD EPYC 7513 (32 cores)	14	0.1

5

Results

This chapter presents the outcomes of the project by looking at the playing strength achieved by our engine, as well as comparing different models and evaluating the key components of the system individually.

5.1 Engine Strength

In this section we present the strength of our best performing engine in the form of comparisons to earlier versions and also as an ELO progression.

5.1.1 Model Comparisons

After choosing the final training setup, after some experimenting and making sure the training pipeline worked, the model went through 8,000 training steps using a replay buffer of 200,000 positions. In Table 5.2, we can see the win rates for the final model with 8,000 iteration steps matched against the models at previous steps in the training phase. It is important to note that less than 1% of the games typically resulted in draws, therefore games ending in draws were discarded when calculating the win rates.

The number associated with each model represents the number of training iterations it has undergone during the final training phase. However, this number does not reflect the total number of training iterations from the start of the project. The final training phase was conducted on an already partially trained model, referred to as Model-0. The performance of Model-0, which can be seen in Table 5.1, shows that it beat a completely untrained model with randomized weights in 98.0 % of games.

Table 5.1: Comparing win rate model-0 to untrained

	Untrained
Model-0 (white)	99.2%
Model-0 (black)	96.8%
Model-0 (both)	98.0%

The training that was previously done on Model-0 was done with a suboptimal training pipeline where we sampled many smaller batches every time we generated new positions, in contrast to the current one where a batch of 4,096 is sampled each training iteration, increasing stability. Mostly this part of the training was done in order to test that the training was working at all, and due to the limited compute resources, we decided to keep the model that the initial training had given us in order to not waste resources. This unfortunately resulted in us not having any presentable data on the initial training, yet it is important to note that this early training was minimal and only accounts for a small factor of the training that was done in the final model. This can be seen in Table 5.2 where Model-8000 has a win-rate of 83.6% against Model-0.

Table 5.2: Model-8000's win rates against the other models over 500 matches (1,000 for both). Config: MCTS-iterations: 3,000

	Untrained	M-0	M-2000	M-4000	M-6000	M-8000
Model-8000 (white)	100%	88.8%	56.1%	52.3%	57.1%	-
Model-8000 (black)	100%	78.4%	49.7%	44.4%	44.4%	-
Model-8000 (both)	100%	83.6%	52.9%	48.4%	50.8%	-

Something that can be observed in Table 5.2 is that Model-4000 seems to outperform Model-8000. To further investigate this, we ran the same test again with Model-4000 as the main model to see if it was a superior model. Those results can be seen in Table 5.3.

Table 5.3: Model-4000's win rates against the other models over 500 matches (1,000 for both). Config: MCTS-iterations: 3,000

	Untrained	M-0	M-2000	M-4000	M-6000	M-8000
Model-4000 (white)	100%	86.0%	54.8%	-	55.1%	55.6%
Model-4000 (black)	100%	81.5 %	53.0%	-	48.6%	47.7%
Model-4000 (both)	100%	83.7 %	53.9%	-	51.9%	51.6%

Comparing the win rates for the two models we see that Model-4000 is the slightly stronger model overall, which is a result of its better performance when playing with the black pieces. Model-8000 might be worse overall but has a higher win rate than Model-4000 when playing white.

To help visualize these results we can convert the win rates into difference in ELO rating, a more commonly understood metric of strength in chess. This can be done with the simple formulas that dictate how the ELO system in chess works. An arbitrary ELO rating of 1,000 is set for Model-8000 which will help us see how the ELO changed in the final training phase.

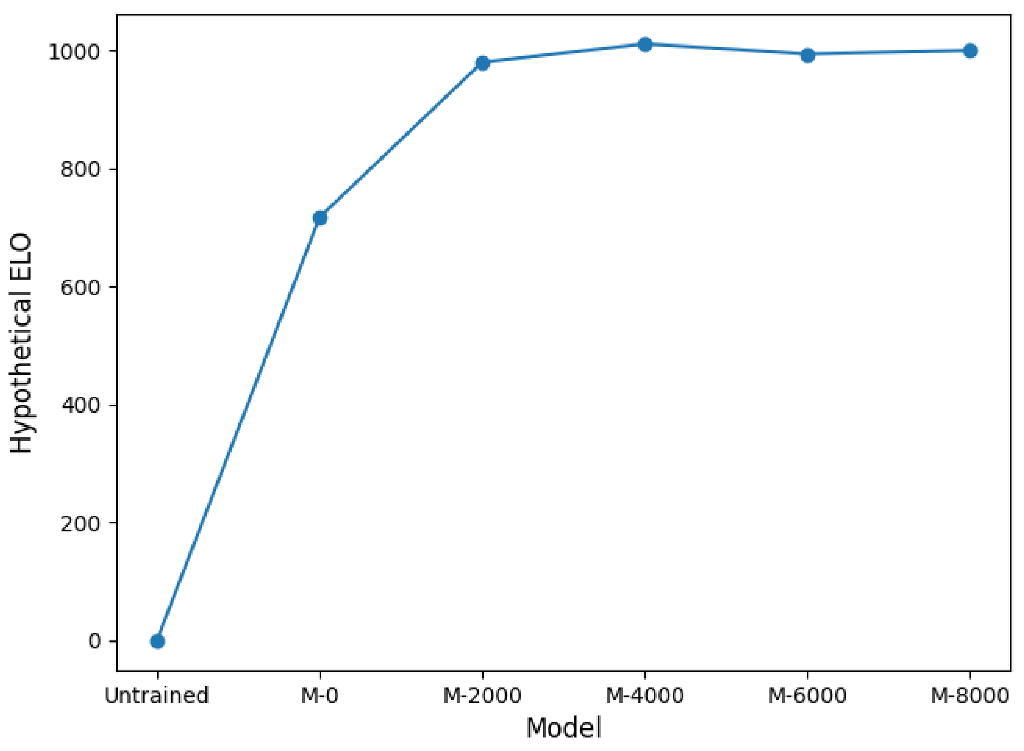


Figure 5.1: ELO progression during the final training phase. The last model, Model-8000 was set to an arbitrary ELO of 1,000, the other ELOs are then calculated in relation to their win rate against Model-8000

.

Another interesting metric to look at Model-4000's and Model-8000's relative strength when playing white compared to black. Looking at Table 5.4 we can observe the same thing that we found when comparing the win rates of the models against other models in Tables 5.2 and 5.3. Model-8000 is stronger than Model-4000 at playing the white pieces, but worse when playing the black pieces.

Table 5.4: The win rate of the model when playing 1,000 games against itself

	White	Black
Model-4000	52.9%	47.1%
Model-8000	55.9%	44.1%

5.1.2 Final Playing Strength

By looking at the results in Tables 5.3 and 5.2, Model-4000 was chosen to be matched against other bots online to get a strength estimate determined by the ELO rating. After 40 games against players and other chess engines, the engine got an ELO rating of 1,729. On Lichess this corresponds to a rating better than 90% of Atomic Chess players that week, which can be seen in Figure 5.2.

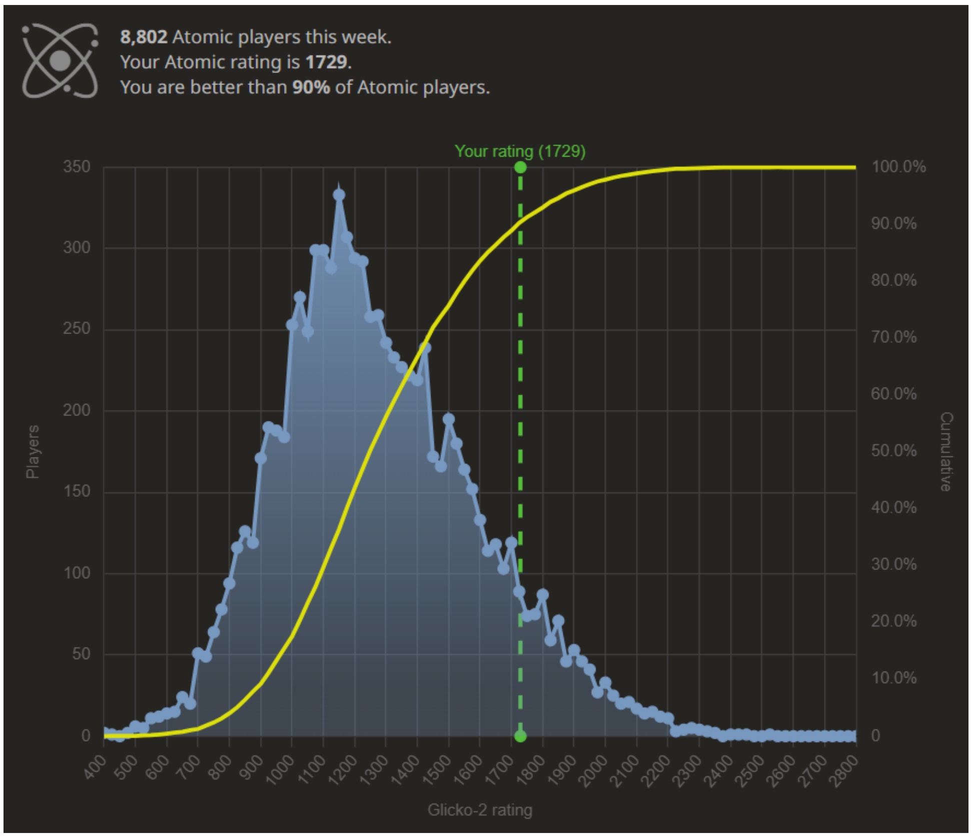


Figure 5.2: Atomic Chess ELO distribution and the projects final engines placement on the chess platform Lichess [17]

The accuracy of this rating is hard to judge. The only atomic players we were able to find and play against were the players on the top 200 leader-board, all having a ELO rating above 1,950. Additionally, there are not many different Atomic Chess engines on Lichess, so the engine was either matched against engines much stronger or much weaker than itself.

A few observations were made while playing against the engine and watching it play stronger players on Lichess. The engine has learned several sharp tactics and strategies in the opening and general strategies that hold true for all parts of the game. But when it comes to tactics specific to the mid- and end-game, the engine becomes noticeably weaker. The engine also does not have a great understanding of being resourceful with its pieces. For instance, when being several pieces down in material, the engine still plays moves that trades its own pieces for the opponents or for some positional winning. This goes against a common playing principle in chess that states you should not trade down pieces when you are behind, because you will be left in a situation with only pawns while the opponents has major pieces (rook, queen, etc.) left. It should be noted that we are not very familiar with Atomic Chess specific tactics and it is possible the engine is making good moves that we simply do not understand and therefore discount them as mistakes.

Additionally, the engine is not very good at time management. The engine currently is fully idle during the opponents turn, which means the engine wastes time every turn as when its turn starts, it still has no idea what it wants to do. A similar issue is the fact that the engine does not take its time remaining at all into account, it only does a hardcoded set number of MCTS-iterations, which typically takes the same time for every turn. This is opposed to a human that naturally make moves quicker when low on time in order to not lose on time.

5.2 Neural Network and Training

In this section we cover the results of the training of our neural network.

5.2.1 Training Process

Several training strategies were explored to identify the most effective approach. Two key factors were found to significantly impact training outcomes: replay buffer size and sampling strategy.

With a very small replay buffer, fitting only the samples generated from previous training step, the model did not seem to learn anything. Even with 8 hours of training the model could not out-perform an untrained model. When matched against the untrained model it achieved a win rate of only 47% for the 500 games played (ignoring the draws). However, increasing the buffer size allowed the model to surpass an untrained version within the same training duration, highlighting the importance of sample diversity for effective learning.

Initial experiments also involved sampling the entire replay buffer during each training step. While this approach led to rapid loss reduction and short-term performance gains, it introduced significant overfitting. Models trained this way struggled when matched against their earliest versions, indicating a lack of generalization.

To address this the strategy was adjusted to use mini-batches for each training step. This change dramatically improved long-term model strength, allowing it to consistently outperform all previous versions. For our final implementation we found that using a mini-batch size of 4,096, while generating 2,048 new positions per step, provided an optimal balance. This effectively resulted in each sample being utilized approximately twice, promoting both stability and sustained improvement.

5.2.2 Learning Verification

To assess whether the network could learn simple tactical patterns a mate-in-2 test was conducted. The model played 50 games against itself from a fixed position where a forced mate was achievable. Figure 5.3 shows the move probability predicted by both MCTS and the neural network over training iterations. MCTS consistently favoured the correct move and after a few games the network started favouring that move as well.

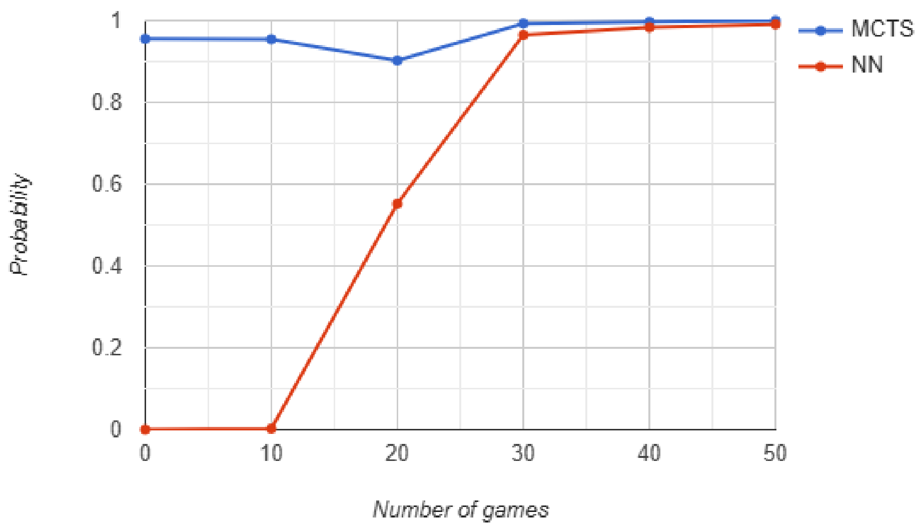


Figure 5.3: Probabilities for the known best move from MCTS and the neural network in a fixed mate-in-2 test. MCTS ran with 10,000 iterations per move.

5.2.3 Training Metrics

The systems throughput during training was measured. With the final configuration (12 residual blocks, 128 channels, 800 MCTS iterations per move on unit 2 in Table 4.1 with all 32 cores) the system achieved:

- **Games generated per hour:** 3,200
- **Positions generated:** $16,384,000 = 2,048 \frac{\text{Samples}}{\text{Training Step}} \cdot 8,000$ Training Steps

These results highlight the computational demands of reinforcement learning using MCTS and neural networks, even on simplified chess variants. Generating data samples concurrently was likely the single most important optimization step, as the throughput of generated samples per unit time increased linearly with the number of workers up to 15 workers, on the hardware configuration used for training. In a later version of the program we were able to scale the number of workers up even further to around 64, but due to unresolved bugs and time constraints these changes were not used during final training.

5.2.4 Virtual Machine Costs

Table 5.5: Final cost per unit

Unit	Expenditure in SEK
1	1,212.32
2	627.39

The results show that the total expenditure for all virtual environments totalled 1839.71 SEK, which is under the goal of 3,000 SEK set for the project.

5.3 Move Generation Component

To verify the correctness of move generation and evaluate its performance, we built a routine for conducting Perft tests. Perft tests take a position as input and output the number of positions, or nodes, found for a given depth, as well as how long the search took. Additionally, Perft divide tests are used to find bugs in move generation, where the number of nodes found for each move from the root position is also recorded. With this, we were able to verify the correctness and measure the speed of our move generation. Running on an AMD Ryzen 7 5800H, a consumer grade CPU, using all 16 cores, our Perft routine found roughly 250,000,000 to 360,000,000 nodes per second, depending on the root position. The output of our Perft routines was compared to the output of Fairy Stockfish's Perft routines for at least ten important positions, totalling over a hundred billion positions, ensuring the same total number of nodes was found for each position [18].

In the engine the move generation runs on independent cores. The single-core performance was benchmarked by repeatedly generating all legal moves for randomly generated positions. The system consistently achieved a rate of 44,519,649 positions per second.

5.4 Monte Carlo Tree Search Optimizations

In this section we present the results of different optimization methods for MCTS.

5.4.1 Batching Selections

By batching together multiple game-states before sending them to the neural network, the performance increased significantly. This takes advantage of the GPUs efficiency when it comes to parallel computations. In Table 5.4 we can see what speed-up was achieved for different number of selections per batch. Our implementation also used virtual loss to minimize competition between threads when it comes to selecting multiple nodes.

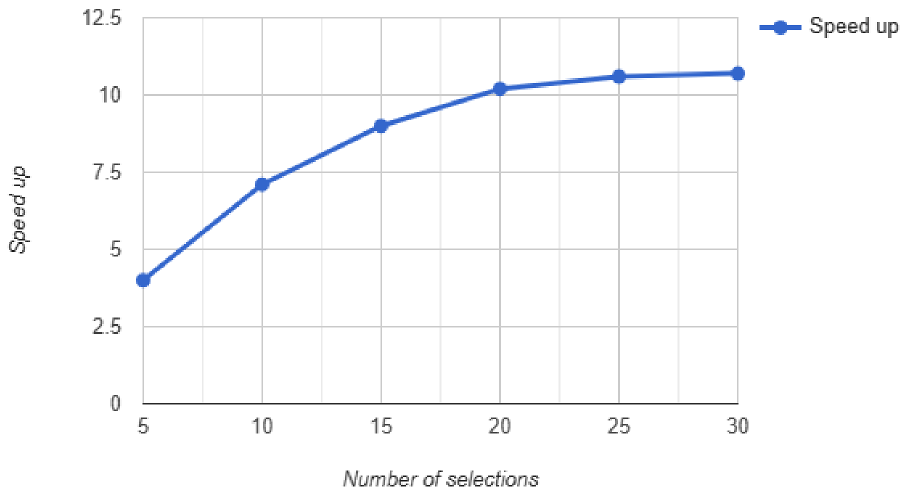


Figure 5.4: Speed up of the engine using different numbers of selections that are batched and sent to the network.

5.4.2 Virtual Loss

Virtual loss was used in the engine to decrease the number of collisions when selecting multiple MCTS nodes during the selection phase. To test the effects of virtual loss, the engine was run with and without virtual loss with different numbers of MCTS iterations. In Table 5.6 we can see that the results show a significant decrease in collisions when using virtual loss. Additionally, while more MCTS iterations lead to more collisions, the increase is less than proportional with virtual loss. However, it is important to note that virtual loss optimizations might lead to distortion when it comes to the final visit counts for the MCTS algorithm. Though, by observing the games the engine played, no significant decrease in strength seemed to happen.

Table 5.6: Average number of select collisions per move in MCTS with and without virtual loss for 1,000 and 5,000 iterations per move.

MCTS Iterations	No Virtual Loss	Virtual Loss
1000	11 000	300
5000	48 000	380

5.5 Time Analysis of the Engine

This section focuses exclusively on the inference-time performance of the engine during gameplay. The training phase, while computationally intensive in total, represents a minor proportion of runtime relative to the inference cost per move and is not a limiting factor in real-time decision-making. As such, the analysis presented here concentrates solely on the performance of move generation, neural network inference, and MCTS search during self-play games.

To identify computational bottlenecks in the engine, runtime profiling was conducted during a standard self-play game on a PC with a Intel i5-12400F CPU and an NVIDIA RTX 3060 GPU. These measurements will differ across different hardware configurations. The measurements include both the time distribution between major components and absolute execution times per move. The results are summarized in Table 5.7.

Table 5.7: Execution Time and Proportion by Component

Component	**Time Per Move**	**Percentage of Total Time**
Neural Network Inference	120 ms	65%
Monte Carlo Tree Search	65 ms	33%
Generating Legal Moves	$<$ 1 ms	$<$ 1%

The results show that neural network inference and MCTS search together account for nearly all execution time during gameplay, with neural network inference alone taking approximately twice as long as the MCTS search. Move generation, on the other hand, is computationally negligible, requiring under 1 millisecond per move and less than 1% of total execution time.

6

Discussion

This chapter discusses the project results, limitations encountered, and lessons learned throughout development. It analyzes the final results in relation to the original project goals. The chapter then covers suggestions for future work and explores directions for improvement. Lastly, the chapter concludes with answers to our research questions.

6.1 Analysis of Final Product and Results

This section discusses the results from Chapter 5. We begin by reflecting on the final engine strength.

6.1.1 Final Engine Strength Discussion

The final engine achieved an ELO rating of 1,729, placing it in the top 10th percentile of the active Atomic Chess player base on Lichess. Even though these results do not match the super-human performance of AlphaZero, they match the ambitions for this project. The training pipeline was therefore shown to be effective in producing a decent engine, yet the results also revealed several limitations that merit reflection.

One of the more interesting findings was the inconsistent performance progression across training iterations. Unexpectedly, Model-4000 ultimately outperformed Model-8000 in overall strength, particularly when playing with the black pieces. This suggests that more training does not necessarily correlate with better performance between the training iteration 4,000 and 8,000. The reasons for this will be discussed further in Section 6.1.2.

The engine also demonstrated some suboptimal patterns of play. It demonstrated strong opening play but became noticeably weaker in mid- and end-game decision-making. For instance, the engine had a tendency to trade down pieces even while behind in material, which goes against chess strategy fundamentals and indicates a lack of long-term planning. One possible reason for this is that Atomic Chess simply has inherently more emphasis on the opening since there are many openings that lead to a quick checkmate. This rewards aggressive, short-term strategies that sacrifice material in order to find a checkmate early. Despite this, it is still undesirable to be weak in the end-game, so there are several potential augmentations to the training pipeline that could counteract this issue. Some solutions could be starting a few self-play games from middle- or end-game positions, or adjusting the exploration/exploitation balance. These ideas will be discussed in Section 6.3.2.

Finally its lack of time management further limited its practical playing strength. A simple fix to would be a variable MCTS iteration count that is based on the time setting for the match. This would mean a longer thinking period for longer time controls and setting a cutoff for the amount of time left, forcing the MCTS iterations to shrink down from the fixed MCTS iteration count to match the remaining time. Another possibility would be to use an additional machine learning algorithm to adjust the amount of MCTS iterations.

The engine being idle during the opponents turn also causes a large time-loss. The obvious solution to this issue would be to continue MCTS indefinitely when waiting for the opponents move. When the opponent finally makes a move, the engine could then use the relevant subtree that has been built corresponding to that move.

6.1.2 Neural Network and Training Process

A key part of the project is the neural network and training setup, which due to the limitations in the hardware, the complexity of the network and number of training samples that can be generated, is constrained. Therefore, unlike AlphaZero and other strong self-learning engines, we used a relatively shallow network structure due to the inference time benefits.

There is an important trade-off between the speed of neural network inference and the complexity of the network. A shallow network enables faster position evaluations, which allows the engine to generate a larger volume of training data through self-play games in a given amount of time. This increases the quantity of training examples, which can help reduce variance in the learning process. Additionally, the shorter inference time allows for more MCTS iterations to be ran when searching for moves. On the other hand, having a too shallow of a network would create a bottleneck where the neural network can not find any more patterns beyond a certain point due to being too simple. A deeper network will be able to identify more complex patterns, resulting in an engine that can find better strategies.

It hard to tell whether our model size was a bottleneck or not. Looking at the hypothetical ELO progression plot we can see that the progress seems to halt after 2,000 training iterations. The reason for this could be that the model is not complex enough to find new patterns beyond this playing level. It could also mean that the neural network forgot strategies it had previously learnt. Despite this, we suspect that what we interpret as the progress halting is simply that the model is not trained on enough data, meaning that when comparing the models between iterations 2,000 and 8,000 we are simply looking at iterations too close to each other. If the training had continued to something like 100,000 training steps we could have seen more meaningful progress. This idea will be discussed further in Section 6.1.4.

6.1.3 Computational Performance Analysis

Part of the purpose of this project was to investigate the possibilities of creating and training a strong self-learning Atomic Chess engine with limited hardware and resources. Both standard chess engines and reinforcement learning algorithms are computationally demanding, which posed many challenges. Therefore, a lot of the time spent on the project was on optimizing the engine for performance and ensuring we were training as effectively as possible. The optimizations for performance included our successful multithreading of self-play, allowing us to play several games at once, as well as batching of tensors before being processed by the neural network. In Figure 5.4, we can see that this optimization significantly improved the computational speed. Without batching, the final model would be significantly weaker due to lacking training.

The time analysis results presented in Section 5.5 indicate that neural network inference constitutes the primary computational bottleneck during gameplay, accounting for approximately 65% of the total time spent, followed by MCTS, accounting for approximately 33% and 1% for move generation. It is important to note that these measurements reflect the performance only on one PC with NVIDIA RTX 3060 GPU and an Intel i5-12400f CPU, different hardware setups will show slightly different result, but the results still give a decent overview.

These results are somewhat in line with our expectations. We can firmly conclude that our move generation is highly optimized, and if something needs optimization it is either the neural network inference or MCTS. On the other hand it is hard to draw meaningful conclusions from these results as we do not know what an ideal self-training engine should look like, perhaps an ideal engine would do only inference and the overhead from MCTS would be minimal. This hypothesis suggests that MCTS is the component that likely would be the component that has most optimization potential.

6.1.4 Training Data Volume

The factor we believe to be our biggest limiter in terms of engine strength is the small training data volume. According to the AlphaZero paper, the chess version of AlphaZero required approximately 300,000 training steps, with every step training on a mini-batch of 4,096 positions, to surpass the best engine at the time, Stockfish [2]. This amounts to approximately 1.2 billion positions. Comparing this to our 16.8 million positions, it paints a clear picture, we only trained on 1.4% of the data they trained on to beat Stockfish, not to mention they later continued training for another 400,000 steps. The time it took AlphaZero to generate and train on those 1.2 billion positions was 4 hours. While for this project we trained for a total of around 150 hours.

While such a disparity is expected given the substantial difference in hardware, it underscores the central role that data generation throughput plays in the effectiveness of creating self-play reinforcement learning systems. The relatively limited volume of training data that could be generated within the project's constraints remains the primary factor identified for improvement in order to develop a stronger engine.

6.2 Engineering Decisions and Their Effects

This section discusses the decisions made in the construction of the chess engine and the impacts it had on the project.

6.2.1 Effect of Replay Buffer

The testing we performed in terms of measuring the effects of the replay buffer showed a clear improvement in the model's training. It is not clear if the model could have become proficient enough to also beat untrained models if allowed to train long enough, but the results are enough to demonstrate that the replay buffer is a good optimization if not vital for the training of the model.

Given that a very basic version of a replay buffer is not hard to implement, consisting only of a list of played moves, where the oldest ones are constantly cycled out to keep the same size on the list, we believe this to be a very worthwhile optimization.

6.2.2 Reflection on Programming Language

The programming language C++ was chosen for this project as it is often used in programs where performance is key. This is likely a reason for why our move generation is so fast. Additionally, the fact that we could use C++ for all parts of the engine likely reduced the overhead which is good for speed. There is however a big challenge with using C++ that we encountered, which is the handling of memory allocation. For instance, Libtorch handles memory allocation internally, creating tensors and returning references to memory region in RAM and VRAM, C++ itself provides no guarantees that these regions remain valid at the language level. This occasionally results in "use-after-free" errors that are difficult to detect. In hindsight, a language like Rust may have been better suited for this project as it prevents many of these problems through strict compile-time memory safety checks. That said, integrating Rust with existing deep learning libraries would likely have introduced its own complications, possibly requiring a mixed-language setup with Python or alternative frameworks. Despite these issues, we remain reasonably satisfied with the choice of C++ given the project's demands for performance, control, and ecosystem support and due to our product was beyond our expectations for the project.

6.2.3 Financial Implications

While the model itself was able to be trained for a cost which was lower than the target cost set for the project, we also as previously concluded that the model has likely not yet been fully trained. Given that we used approximately 40% of our allocated budget at 8,000 generations, it would mean that we could train to a maximum of roughly 20,000 generations before exceeding the budget.

In comparison, AlphaZero was trained for 700,000 generations [2] which shows that if we were to train a model of equivalent size we would still be able to train much further than 20,000 generations, which in turn implies that we would far exceed the budget if our goal was to train the model to its maximum strength.

We could also further implement optimizations in order to better utilize the virtual machines we had rented, as all of them were rented at a set price per hour, which would allow us to train further for the same total price.

As such, we are unable to determine exactly whether or not we could have trained the model for the price point assuming all possible optimizations were implemented, but does lead us to believe we could not train an equivalent model for that price. We were able to train up a model that plays reasonably well, which means that if you were to want to train up a model which plays with relatively limited strength in comparison to AlphaZero, that could be achieved within a limited budget.

6.3 Future Research

Given additional time and resources, several areas would have been prioritized. As has been stated multiple times in this report our main limit was the amount of compute power that we had access to. If we had more resources for this project the natural next step would be to continue to train the model and see at what point it stops improving.

6.3.1 Symmetry Augmentation

Increasing the rate at which we could generate training samples was one of the more difficult challenges to overcome and when achieved, resulted in much better training performance, which means it stands that further optimization in this field would be beneficial. One technique that was explored in order to improve this was symmetry augmentation. The underlying idea is that due to the symmetric nature of a chessboard, it might be possible to generate two training samples for the price of one. This would be done by mirroring a game sample along the vertical symmetry line for each sample we generate, meaning the board will be mirrored horizontally.

Doing this would essentially copy the same game with each player switched, due to the two sides king and queen starting on opposing sides. This also plays into the fact that our model does not separate between white and black meaning that mirroring would not compromise the the model weights. Therefore, by only applying this single optimization, we are potentially doubling the number of training samples generated per unit time. Unfortunately, we were not able to implement this in the given time for the project.

There is potentially also another more complex approach that adds this feature by using the symmetrical invariance of the neural network weights themselves. The convolutional layers have some inherent spatial context and symmetry to them, which might make it possible to create a modified convolutional layer that takes advantage of this fact to be able to store two mirrored positions on opposing sides of the board as the same convolution. This could be used to downsize the neural network which would improve its speed. This idea is quite immature and would likely be much more complex and difficult to implement than the first alternative and could perhaps be a separate research topic to study on its own.

6.3.2 Improving the Middle and End-game Strength

One issue we encountered during training was that many games did not last very long, due to the explosive nature of our chosen variant. In Atomic Chess, there are many opening tricks that try to give checkmate directly from the start which meant the engine spent a lot of time learning different openings while later stages of the game did not get as much training. This also became apparent when playing against the engine ourselves, as it performed better in the opening than in the end-game.

To address this, we considered ways to incentivize longer games. One potential approach involves adding temperature proportional sampling. The idea is to scale the move selection probabilities using an exponent, making the probability distribution either sharper, making it more deterministic or flatter, more exploratory.

One theory we had, but were unable to test, was that by forcing the model to play more deterministically in the opening, this might help it avoid early forced checkmates and reach deeper positions. This would in turn increase its exposure end-game scenarios, and subsequently help its performance in the end-game. While we expect a sufficiently complex network to eventually learn end-game play, we believe this adjustment could help accelerate that process.

Another less theoretical approach we considered was to just weigh samples by the total number of moves in the games that they were drawn from, or simply by the full move counter of the sample itself, thus training more heavily on longer games and end-game scenarios.

An additional idea that could have been used to improve later stages of the game would be to not only train from the regular starting position but also generate a set of middle-game and late-game positions, and then starting the self-play games from those positions. Though it is very possible that a method like that might lead to overfitting to certain positions, so a large set of unique positions would have to be generated.

6.3.3 Additional Neural Network Inputs

Currently, the neural network's inputs are very simple, only consisting of the board-state and nothing else. Something that might be interesting to explore would be additional inputs to the neural network. For example, the network could receive the game's move count, giving it awareness of how far into the game it is. It could also track move repetition, allowing it to recognize when a position is being repeated for the third time, which results in a draw. En passant rights are also an input that could help the engine spot en passant moves. Another alternative would be to add history of past positions, which was done by AlphaZero. These extra inputs might provide the neural network with a better overview of the game and enable it to learn more patterns and complex strategies.

6.4 Conclusion

In this section, we conclude the report by answering our research questions.

6.4.1 Question 1

How feasible is it to train a chess engine through self-play, similar to AlphaZero, using computational resources available to the common enthusiast?

This project has proved that it is feasible to apply self-training techniques, even when constrained by hardware resources. According to our evaluation on Lichess, the resulting engine was significantly stronger than the average human player, and this was done while staying far below our budget of 3,000 SEK. This goes to show that even without both a prior dataset to train on and a prior understanding of the game's strategies, a decent engine can be developed.

6.4.2 Question 2

How can a chess engine using a self-training engine be designed and optimized to work at this scope?

In order for a self-learning chess engine to be successful, it needs to train on a sufficiently large number of training steps, where the quality of each training step is mostly determined by the total number of samples in the replay buffer and the number of samples that were generated since the last training step. Thus, the throughput of samples generated per unit time is a determining factor when it comes to actual model performance. We found that limited access to processing power mainly becomes a problem of optimizing for hardware utilization and minimal congestion.

To optimize performance under these conditions, we employed a shared-worker architecture, where multiple threads generated self-play games in parallel and wrote to a shared buffer. Model evaluation was also batched to exploit the parallelism of the GPU, as opposed to serial inference. These decisions significantly increased training throughput without requiring additional hardware. This design demonstrates that with careful engineering, it is possible to develop a self-learning AI model for a complex game like Atomic Chess on moderate hardware. The pipeline was cost-effective and scalable, and this suggests that others would be capable of duplicating such efforts even with limited budgets.

Bibliography

[1] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291.

[2] 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.

[3] Lichess. *Atomic Chess*. https://lichess.org/variant/atomic. Accessed: 2025-04-16. n.d.

[4] Chess.com. *Stockfish*. Accessed: 2025-05-15. URL: https://www.chess.com/terms/stockfish-chess-engine.

[5] Chu-Hsuan Hsueh et al. "AlphaZero for a Non-Deterministic Game". In: *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*. 2018, pp. 116–121. DOI: 10.1109/TAAI.2018.00034.

[6] J. Jumper, R. Evans, A. Pritzel, et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596 (2021). Published 15 July 2021, pp. 583–589. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

[7] Oleg Arenz. "Monte Carlo Chess". MA thesis. Technische Universitat Darmstadt, 2012.

[8] Steven S. Skiena. *The Data Science Design Manual*. Springer, 2017. URL: https://doi.org/10.1007/978-3-319-55444-0.

[9] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE]. URL: https://arxiv.org/abs/1511.08458.

[10] Guoping Xu et al. "Development of residual learning in deep neural networks for computer vision: A survey". In: *Engineering Applications of Artificial Intelligence* 142 (2025), p. 109890. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2024.109890. URL: https://www.sciencedirect.com/science/article/pii/S0952197624020499.

[11] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/abs/1502.03167.

[12] Guillaume M. J. -B. Chaslot, Mark H. M. Winands, and H. Jaap Van Den Herik. "Parallel Monte-Carlo tree search". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5131 LNCS (2008). Cited by: 184, pp. 60–71. DOI: 10.1007/978-3-540-87608-3_6. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-55249093890&doi=10.1007%2f978-3-540-87608-3_6&partnerID=40&md5=7f4c0968675d81db80dd6bf369609169.

[13] Gustaf Algeskog et al. *A Self-Trained Engine for a Chess-variant.* Unpublished manuscript. Bachelor's thesis, Chalmers University of Technology. 2024.

[14] Abdulrazak Ahmedmohamed et al. *A Self Trained Engine for the Game of Antichess.* Unpublished manuscript. Bachelor's thesis, Chalmers University of Technology. 2025.

[15] Tom Schaul et al. *Prioritized Experience Replay.* 2016. arXiv: 1511.05952 [cs.LG]. URL: https://arxiv.org/abs/1511.05952.

[16] TensorDock Inc. *TensorDock - Cloud GPU Servers.* https://tensordock.com. Accessed: 2025-05-18.

[17] Lichess. *Lichess.* https://lichess.org. Accessed: 2025-05-19. n.d.

[18] Fairy-Stockfish contributors. *Fairy-Stockfish.* https://github.com/ianfab/Fairy-Stockfish. Accessed: 2025-05-18.