CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Self–Trained Engine for Atomic Chess

Bachelor's Thesis in Computer Science and Engineering

Emil Andersson

Simon Hammerlid

Gustav Karlsson

Sebastian Klang

Ken Porota Ndimurukundo

Elias Söderberg

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Bachelor's thesis 2025

A Self–Trained Engine for Atomic Chess

Emil Andersson
Simon Hammerlid
Gustav Karlsson
Sebastian Klang
Ken Porota Ndimurukundo
Elias Söderberg

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 Engine for Atomic Chess

Emil Andersson Simon Hammerlid Gustav Karlsson Sebastian Klang Ken Porota Ndimurukundo Elias Söderberg

Supervisor (Handledare): Jean-Philippe Bernardy, Department of Computer Science and Engineering
Examiners: Patrik Jansson and Arne Linde, Department of Computer Science and Engineering
Graded by teacher (Rättande lärare): 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

A Self-Trained Engine for Atomic Chess

Emil Andersson, Simon Hammerlid, Gustav Karlsson, Sebastian Klang, Ken Porota Ndimurukundo, Elias Söderberg

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis concerns the development of a chess engine to play a variant of chess called atomic chess, utilizing a neural network. The neural network is modeled after DeepMind's AlphaZero, which is a model that learned standard chess from only the rules, and no real-world games (hence "zero"). We demonstrate an adapted model that improves its playing strength in atomic chess, given enough training time. This is done using a *deep convolutional neural network*, trained with data generated by a modified *Monte Carlo tree search* in a process called self-play. These two components feed each other data in a cycle: the neural network guides the tree search, and the results of the tree search are then used to train the network, repeatedly. The result is a trained network that is shown to have improved from the untrained model, which corresponds to an unguided or randomly guided Monte Carlo tree search. An extensive background aimed at computer engineering students is also included, explaining the terms used in the thesis.

Keywords: Chess, Chess Variant, Atomic Chess, AlphaZero, AI, MCTS, Self-play, Chess engine, Neural Network

Sammandrag

Den här rapporten angår utvecklingen av en schackmotor för att spela en variant av schack som kallas atomschack, med hjälp av ett neuralt nätverk. Det neurala nätverket modelleras efter DeepMinds AlphaZero, vilket är en modell som lärde sig vanligt schack endast från reglerna utan några verkliga partier (därmed "zero"). Vi påvisar en anpassad modell som förbättrar sin spelstyrka i atomschack, given tillräcklig träningstid. Detta görs med hjälp av ett *faltningsnätverk* som tränas med data genererad av en modifierad *Monte Carlo-trädsökning* i en process som kallas självspel. Dessa två komponenter matar varandra data i en cykel: det neurala nätverket vägleder trädsökningen, och resultaten av trädsökningen används sedan för att träna nätverket, gång på gång. Detta resulterar i ett tränat nätverk som visas ha förbättrats från den otränade modellen, vilken motsvarar en icke vägledd, eller slumpmässigt vägledd Monte Carlo-trädsökning. En omfattande bakgrund riktad åt data- och IT-studenter inkluderas även och förklarar termerna som används i rapporten.

Nyckelord: Schack, Schackvariant, Atomschack/atomiskt schack, AlphaZero, AI, MCTS, Självspel, Schackmotor, Neuralt nätverk

Acknowledgments

We extend our sincere thanks to our supervisor, Jean-Philippe Bernardy, whose expertise and thoughtful guidance were invaluable throughout the course of this project. We are also thankful to the CSE department at Chalmers for giving us access to additional computational resources.

Emil Andersson, Simon Hammerlid, Gustav Karlsson, Sebastian Klang, Ken Porota Ndimurukundo, Elias Söderberg, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

The usage of artificial intelligence has increased greatly in the last three years [1], and the proof of their potential in various fields is only becoming clearer with time. One possible benchmark for the strength of artificial intelligence is its adaptation to chess, one of the most popular and well-studied board games in the world. Until recently, the strongest chess engines were those that had been configured by chess experts, using comprehensive evaluation functions [2]. But in recent years, a new type of chess engine has appeared. Starting with AlphaZero in 2017, certain kinds of engines have been able to master the game without any external help from humans, learning solely through self-play. At its introduction, AlphaZero was extremely competitive against the strongest traditional engines available at that time [3]. The aim of this report is to achieve something similar: a chess engine learning only by playing against itself. However, in this case, the engine will be tuned for a variant of chess known as atomic chess.

1.1 History

Chess history dates back to the 6th century in India, where its earliest known predecessor Chaturanga was played [4]. It was after the Arab invasion and conquest of Persia that the game was taken up by the Islamic world and later reached Europe via Spain and Italy. By the 15th century, chess had evolved into something very close to the version we know today [5]. It is now one of the most popular games in the world, being well-established in more than 200 countries [6].

Until 2017, the best chess engines were those whose evaluation functions were implemented by humans; that is, to determine whether or not a position is good, the engine would calculate a score based on different heuristics set by humans. But that would change when a team of programmers at the Google company DeepMind released an engine named AlphaZero, which had not only learned, but mastered the game of chess with its only knowledge of the game being the rules. This was achieved by continuously playing against itself, having knowledge of which moves are available and whether or not the game has ended (and subsequently who won). The engine achieved this with a combination of a neural network, together with an exploration technique, which is similar to how this report will construct the same kind of engine.

With this approach, AlphaZero was able to beat the previous reigning champion of chess engines, Stockfish, without losing a single game out of one hundred. Today, all the best engines in the world include neural networks somewhere in their evaluation functions, while humans are still fine-tuning them [7].

1.2 Purpose

The purpose of this project is to design and develop a self-trained chess engine for a chess variant. The goal is to create a model that learns and improves through self-play. By integrating a game tree search with a neural network, the engine will dynamically analyze and evaluate the state of the board during a game, similar to engines like AlphaZero.

The playing ability/performance of the engine will be evaluated through playing against both other versions and engines. Finally, this project aims to enable other engines to compete against the final product while also providing a user-friendly interface for players to challenge the engine.

1.3 Scope

The self-trained chess engine will specialize in atomic chess rather than supporting multiple variants, as adapting to multiple rule sets would require significantly greater development complexity and computational resources. Different variants of chess have unique rules, piece interactions, and strategic considerations. This would require custom adaptations in rule interpretation, search heuristics, neural network inputs, and training processes.

Supporting multiple variants would require extensive training and optimization, making it infeasible within the project timeline. By focusing on a single variant, the model can be fine-tuned to the distinct characteristics of atomic chess, maximizing efficiency and accuracy. A general-purpose engine would sacrifice specialization in favor of adaptability, likely resulting in weaker performance for each variant.

Self-play is a process in which an engine generates its own training data by repeatedly playing games against itself and learning from the results. This approach does not rely on external game records or stronger engines, but instead uses only the rules of the game to improve performance through self-competition. We are focusing on self-play rather than including real-world open-source data. This requires more time and resources, but is done purposefully in order to demonstrate the capacity of a computer to learn from scratch, relying only on the rules of atomic chess. In addition, one can argue that atomic chess lends itself to self-play because of the shorter matches, as explained in section 1.4.

1.4 Atomic Chess

Atomic chess is a chess variant that uses the same board setup and mostly follows the same rules as traditional chess, with a few key exceptions[8]. When a piece is captured, an explosion surrounds the captured square. This explosion removes the piece capturing, the captured piece, and all adjacent pieces (with the exception of pawns) from the board. A player cannot make a capture that leads to the explosion of their own king.

The explosion mechanic allows the removal of multiple pieces in a single turn and introduces the potential for indirect threats to the king. As a result, the average length of a match in atomic chess is shorter than in traditional chess. In February 2025, the average number of moves per game for atomic chess on lichess.org was 15.5[9], compared to 40.6 moves per game for traditional chess on the same platform[10].

2

Neural Networks

Neural networks or artificial neural networks are computational structures that were initially inspired by the way biological neurons transmit and process information in the brain [11]. The network is composed of interconnected units, called neurons, that process the input through successive layers. By iterating this process, the system continuously learns and progressively makes better predictions. This mechanism enables the neural network to approximate complex functions and distinguish patterns in the input data [12].

2.1 Fundamental Concepts

A neural network consists of an input layer, one or more hidden layers, and an output layer [11]. The input layer converts the raw input data to the correct shape. The hidden layers perform feature extraction and transformation by combining weighted inputs from the previous layer. Each hidden unit, or neuron, computes a weighted sum of its inputs and adds a bias term before passing the result through a nonlinear function called the activation function. The output a of a neuron with d number of inputs can be mathematically described by the following equation:

$$a = f(\mathbf{w}^T\mathbf{x} + b) \tag{2.1}$$

where $\mathbf{w} \in \mathbb{R}^d$ is the weight vector, $\mathbf{x} \in \mathbb{R}^d$ the input vector, $b \in \mathbb{R}$ the bias, and $f : \mathbb{R} \to \mathbb{R}$ the activation function [13].

(The theory behind activation functions is expanded in Section 2.1.1) The output layer combines the results from the hidden layers and produces the prediction of the network. This prediction can, for example, represent a class label, a numerical value, or a probability distribution [12]. Figure 2.1 presents a simple neural network consisting of an input layer, two hidden layers, and an output layer. The higher the value of a node, the greater its impact, and the lines represent the weights that lead to each subsequent node.

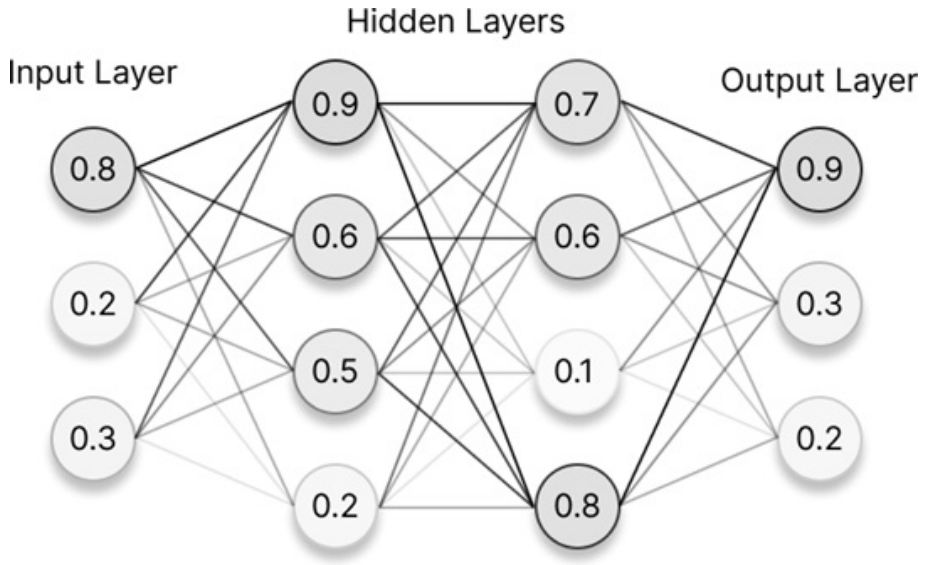

Figure 2.1: A simple neural network with two hidden layers.

2.1.1 Activation Functions

Activation functions are central to neural networks because they introduce nonlinearity, allowing networks to approximate highly complex real-world functions [13]. Two commonly used activation functions are the rectified linear unit (ReLU) and the hyperbolic tangent. ReLU, expressed mathematically as

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{2.2}$$

helps alleviate some training difficulties by preventing negative values from contributing beyond zero, which results in sparse representations that are computationally efficient to evaluate. The hyperbolic tangent, in contrast, maps all inputs to the range $[-1, 1]$ and is described by the following equation:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1 \tag{2.3}$$

.

However, because the hyperbolic tangent saturates as $|x|$ increases, it is affected by the vanishing gradient problem in deep neural networks [13]. Saturation in this context means that when $|x|$ increases, the derivative of tanh(x) becomes very small.

2.1.2 Training Procedure

Training a neural network involves iterative adjustment of the model's parameters, for example weights and biases, to minimize a chosen loss function [12]. The process typically begins with a forward pass, where input data is fed into the network and the

output, a prediction, is compared against the correct label or value. This comparison produces a loss, which serves as a quantitative measure of how poorly the network is performing on that training sample. Common losses include *mean squared error* for regression tasks and *cross-entropy loss* for classification tasks [11]. Once the loss is computed, backpropagation applies the chain rule of calculus to determine how each parameter influenced the error. The loss gradients are then used to update these parameters with respect to all weights and biases. When training a model, the data is typically split into a training dataset and a validation dataset. The training set is used to train the network, and each pass through it is referred to as an epoch. Meanwhile, the validation set is typically used to monitor performance and detect signs of overfitting or underfitting [12].

The backpropagation step mentioned above depends on the gradient, or rate of change, of the loss function. That is, the direction and magnitude of the fastest descent in the high-dimensional domain of weights and biases in the loss function. This gradient must be approximated. There is also the problem of local minima in the loss function. These problems can be dealt with in several ways, and the different methods that deal with these problems make up what are called *optimization methods*. One of the most successful and popular optimization methods is called *stochastic gradient descent* (SGD). It essentially works by using randomness in the selection of samples to optimize on, which is intended to minimize the impact of local minima while not being too slow. This method, along with a method called *Adam*, was used by the bachelor group of last year who did a similar project on antichess, which is another variant of chess [14]. Adam is a modified version of SGD that automatically calculates a *momentum* for each parameter; that is, it includes the first and second derivative of each parameter with respect to *time*, as well as loss [15]. Their conclusion was that SGD is far superior compared to Adam, at least in the case of antichess [14].

2.1.3 Key Hyperparameters

Hyperparameters are configuration values set before or during training that govern how a neural network learns. The *learning rate* is a key hyperparameter, dictating how aggressively the weights are adjusted during each update step [11]. If the learning rate is set too high, the network may exhibit unstable behavior or fail to converge to a good solution. If set too low, training can become prohibitively slow and might stagnate in suboptimal states. Another central hyperparameter is the *batch size*, which specifies how many training examples are processed and averaged before a gradient step is taken. Large batch sizes can smooth out noisy gradient signals but require more memory, whereas small batches introduce more stochasticity, which can help escape local minima, but may also destabilize convergence. Tuning these and other hyperparameters often demands a careful balance between efficiency and precision [12].

2.1.4 Underfitting and Overfitting

Two prominent problems in neural network training are overfitting and underfitting [11]. Overfitting arises when the network begins to memorize specific patterns, noise, or even outliers in the training data rather than learning a general mapping. This condition is typically recognized by a low training error accompanied by a substantially higher validation or test error. Underfitting occurs when the model cannot capture the underlying trends of the data, resulting in high training and test errors. Methods for combating overfitting include collecting more data, applying *regularization* such as *weight decay* or *dropout*, or employing *data augmentation*. Underfitting can often be remedied by using a deeper or wider architecture, adding more training epochs, or reducing overly aggressive regularization [11]. Monitoring validation metrics throughout training is considered a fundamental practice to ensure that the model is generalized properly.

2.2 Convolutional Neural Network

A convolutional neural network (CNN) is specialized in handling data that exhibit spatial or temporal dependencies, such as images, audio, or grid-based inputs. These networks are particularly effective at capturing local patterns through shared weights, thereby dramatically reducing parameter counts compared to traditional neural networks [11]. For reference, a fully connected network is a network where every layer is a fully connected layer, and a fully connected layer is a layer where every pair of input and output neurons has a corresponding weight and bias. The property of having fewer parameters makes CNNs less prone to overfitting on tasks like image classification or game-board analysis, and they have become an integral part of state-of-the-art systems in computer vision and in board-game AI systems. CNNs are well suited for chess-related tasks because the chessboard can be viewed as a structured input, where local patterns such as piece clusters and potential interactions may hold significant strategic value [12].

By systematically multiplying and summing over local patches, the filter can respond strongly to particular spatial structures, such as a corner or a specific arrangement of pawns. Convolutional operations thus improve representation learning by automatically detecting relevant patterns at each position of the input.

2.2.1 Mathematical definition of Convolutions

A *convolution* is a mathematical operation in which a small kernel (or filter) is shifted across the input, for example, an image, in order to detect local features [11]. For this project, the input would be a chessboard representation. Formally, a continuous one-dimensional convolution of a function f with a kernel g can be expressed as the function $f * g$, defined as:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t)g(x-t)dt \tag{2.4}$$

while in practical implementations with digital data, it is evaluated in discrete form:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f(m)g(n-m) \tag{2.5}$$

One can imagine the function g being flipped horizontally and sliding rightward over f. The higher the value of n, the farther g slides rightward. The result of the convolution is then achieved by multiplying these two curves point-wise and integrating (or adding). This result can be expressed concisely as the dot product of f and g when g is flipped horizontally and slid right by n.

The function $(f*g)[n]$ can be thought of as the "recognition strength" of the pattern represented by the kernel function g at location n in the underlying function f. The higher the value of the convolution at a given point, the more strongly the given pattern is exhibited at that location.

2.2.2 Channels, Groups and Output

Each convolutional layer has a number of input and output *channels*. Each of the output channels generally corresponds to some learned pattern in one or more input channels. All channels in the same layer have the same dimensions as each other. For example, in the case of chess, each channel is an 8×8 matrix of floating point numbers.

The total number of unique kernels in a convolutional layer depends on the number of input channels, the number of output channels, and the number of *groups*, which is a hyperparameter. One can think of a group as a collection of input and output channels that are all interconnected and independent of the other groups. For example, if a layer has 256 input channels, 128 output channels and 4 groups, then this is equivalent to 4 parallel convolutional layers, each having 64 input channels and 32 output channels. For this reason, the number of input and output channels each have to be divisible by the number of groups.

Inside each group, there is exactly one kernel for each pairing of input and output channels. In the example above, this means that the total number of learned kernels is equal to:

$$\begin{aligned}&\text{Groups} \cdot \text{InputChannelsPerGroup} \cdot \text{OutputChannelsPerGroup}\\ &= 4 \cdot 64 \cdot 32 = 8192 \text{ kernels}\end{aligned}$$

Note that each output channel has not one, but many (in this case 64) channels connected to it, each with a unique filter. The results of each of these filters are simply added to create an output channel.

By default, libraries such as PyTorch and Keras set groups=1, presumably due to

the fact that this is the only number that evenly divides all integers. However, if groups=input_channels, this is called a *depthwise convolution*, which is when each output channel is mapped-to by exactly one filter from one input channel. This requires considerably fewer parameters [16], but also cannot find any cross-channel patterns.

The output to each neuron in a multi-channel convolutional layer is similar to that of the linear layer defined in section 2. However, the simple dot product is replaced with the sum of convolutions associated with that output neuron:

$$a = f\left(b + \sum_{k=1}^{C_{in}} \text{Conv}_k\right) \tag{2.6}$$

Here a is the value of a given output neuron, f is the activation function (generally ReLU), b is the bias for this neuron, C_{in} is the number of input channels per group, and Conv_k is the result of the convolution yielded to the current coordinate by the specific kernel corresponding to the paring of input channel k to the current channel. Note that this operation has a bias for every output neuron, just as in the linear layer.

2.2.3 Two-Dimensional Convolutions

In two dimensions, f is an input *matrix* where each entry in the matrix is a floating point number, and the kernel g is a smaller matrix of learned parameters. Now, the kernel is slid sequentially across the entire image, and the new value *at each point* is the sum of the point-wise product of the values in the kernel with the values in the image that are "behind" the kernel surrounding that point. 2D convolutions are very suitable for image processing because of their good capability of extracting various features. One common use case of image processing is edge detection, where the convolution highlights areas with great contrasts between neighboring pixels. Figure 2.2 shows how the Scharr operator is applied to an image for edge detection.

2.2.4 Filter Characteristics

A convolutional layer consists of multiple filters or kernels, each trained to detect different features within the input map [11]. Key parameters include the filter dimensions, the stride, and any padding used:

- **Filter dimensions:** Size of the filter/kernel. An ordered pair of positive integers representing height and width.
- **Stride:** The distance that the kernel slides after each dot product. Usually 1.
- **Padding:** "Transient" or "virtual" squares added around the image in order to select the dimensions of the result. For example, to keep the same dimensions

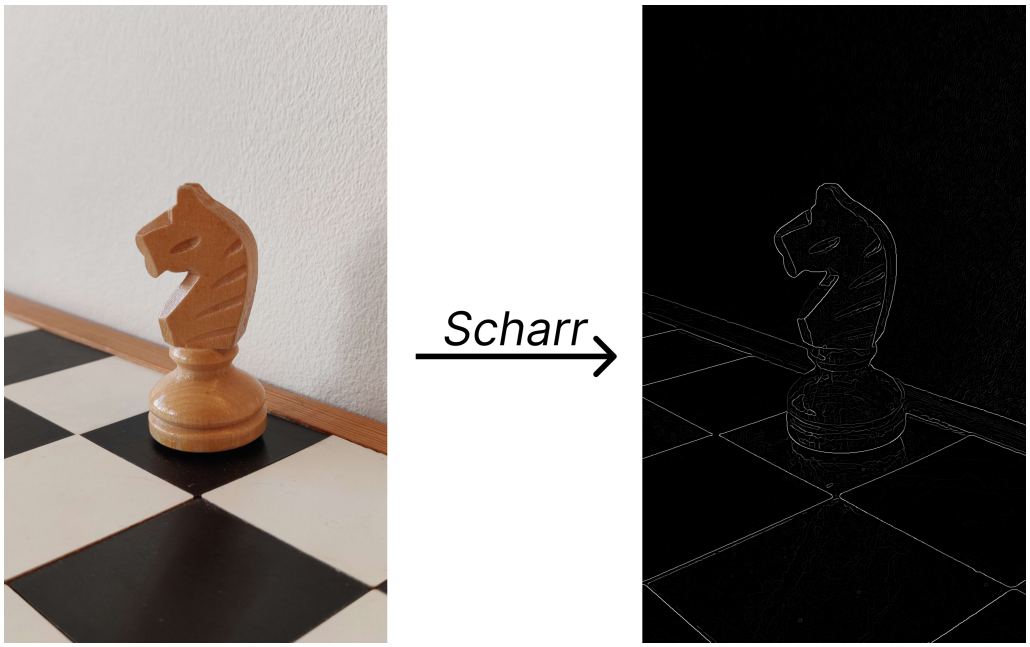

Figure 2.2: Edge detection using the Scharr operator.

of the image while using a 3x3 kernel, a single-square-width padding has to be added around the edge of the image, so that the middle of the kernel can reach the edge of the original image.

Each filter generates a feature map containing activation values for each position where the filter is applied. These feature maps are then passed through a nonlinear activation function, enabling the network to learn intricate hierarchies of features. After convolution layers, it is common to incorporate fully connected layers to combine features into a final output, especially in classification or regression tasks. This design promotes greater representational power for problems that exhibit local patterns [12], [11].

2.3 Residual Neural Network

As networks deepen, traditional architectures sometimes encounter difficulties in backpropagation due to gradients diminishing or exploding over many layers [11]. Residual neural networks, ResNets, address this by introducing shortcuts, or skip connections, that let layers learn perturbations relative to an identity mapping, rather than an entirely new transformation. This improvement allows very deep networks to be trained without severe degradation in performance [17].

Vanishing gradients occur when the chain rule of backpropagation propagates derivatives through many layers, each of which can scale the gradient down towards zero [11]. Especially in earlier architectures with sigmoid or other activations, these gradients quickly became infinitesimal, impairing the ability of the network to update weights in lower layers. The result is either painfully slow convergence or an in-

ability to capture complex dependencies. Although certain activation functions and careful initialization can partially mitigate vanishing gradients, deeper networks still require additional strategies to ensure effective training [17].

Skip connections solve much of the vanishing gradient challenge by providing a direct path through which gradients can flow from higher layers back to earlier layers [11]. If a layer ideally learns some function f, then a residual connection makes the computed output:

$$g(\mathbf{x}) + \mathbf{x} \approx f(\mathbf{x}) \tag{2.7}$$

This means that the system only needs to learn the difference of the ideal function $f(\mathbf{x})$ from the identity mapping $\mathbf{x}$. This difference is approximated by the *residual*, which is $g(\mathbf{x})$ above. The residual can also be expressed as the difference between the output and the input of the residual block. If the residual is small, then learning does not force drastic changes in weights, thus stabilizing training dynamics over many layers [17]. In practice, this allows depth to increase dramatically, which significantly benefits domains such as image recognition and board-game evaluation, where deeper models can capture complex, high-level features.

A typical *residual block* includes one or more convolutional operations, batch normalization to stabilize distributions across batches, and a nonlinear activation. The block then adds its input directly to its output via the skip connection before passing the result downstream [17] . This structure is replicated across many layers, making it possible to build networks of extreme depth. Such deep networks can capture nuanced representations with a level of detail that shallower architectures often fail to achieve [17]. Figure 2.3 illustrates the basic structure of a residual block, where the input x is initially stored to later be added onto the output of the convolutions.

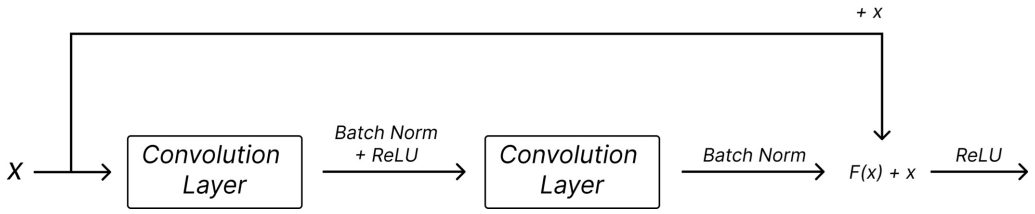

Figure 2.3: An example configuration of a residual block.

2.4 Introduction to Reinforcement learning

Reinforcement learning is a type of learning that is based on the idea of an *agent* learning to maximize the reward signal from an *interpreter* in the context of a given *environment*. It does not used pre-labeled data, unlike supervised learning. In our case, the agent is the neural network, the interpreter is the modified MCTS, and the environment is the chess board. The *exploration-exploitation dilemma* is also central to reinforcement learning. In our case, this dilemma concerns whether to

explore new nodes or to exploit good nodes, and is solved by the *UCT* and *PUCT* formulas, which are expanded upon in Sections 3.1 and 3.3, respectively.

3

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that combines random sampling with incremental tree expansion to make decisions. It gained fame for its success in complex domains like the game of Go, where classical search techniques had struggled [18]. MCTS performs a series of simulated play-outs to gradually build a game tree and converge toward strong moves. Importantly, it is an anytime algorithm, it can return a decision at any time, and more simulations generally improve the quality of the result. Unlike minimax-based methods, MCTS does not require a hand-crafted evaluation function, relying only on the basic game rules for simulations. This makes it applicable even when no reliable heuristic is available, a key reason why it has "succeeded on difficult problems where other techniques have failed" [18].

Each iteration of MCTS consists of four phases [18], often described as

1. **Selection:** Starting from the root node, recursively select child nodes according to a tree policy (e.g., a heuristic like UCT, which is explained in Section 3.1) until a node is reached that is not fully expanded. In other words, the algorithm descends the tree to find the most 'urgent' node that still has unexplored actions; see Figure 3.1.

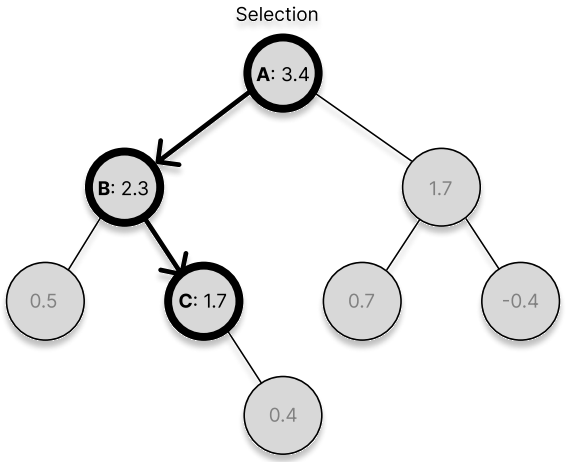

Figure 3.1: Starting from root node (A), the algorithm follows a selection rule to traverse the tree until reaching a node (C) which is not fully explored.

2. **Expansion:** Add one or more new child nodes to the tree by exploring an unvisited action from the selected node. Each new node represents a subsequent state resulting from that action; see Figure 3.2.

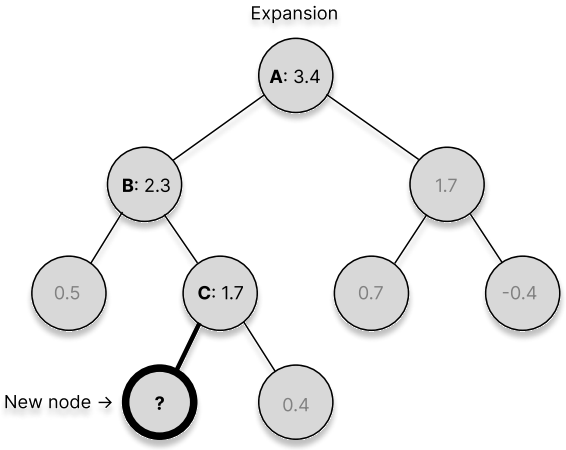

Figure 3.2: From node (C), a new child node is added to the tree, corresponding to an unvisited move.

3. **Simulation (Playout):** From the newly added node, run a Monte Carlo simulation to the end of the game (or a depth cutoff), using a default policy such as random moves or a light heuristic to estimate the outcome (e.g., win/loss or value estimate). This simulates playing out the game from that state; see Figure 3.3.

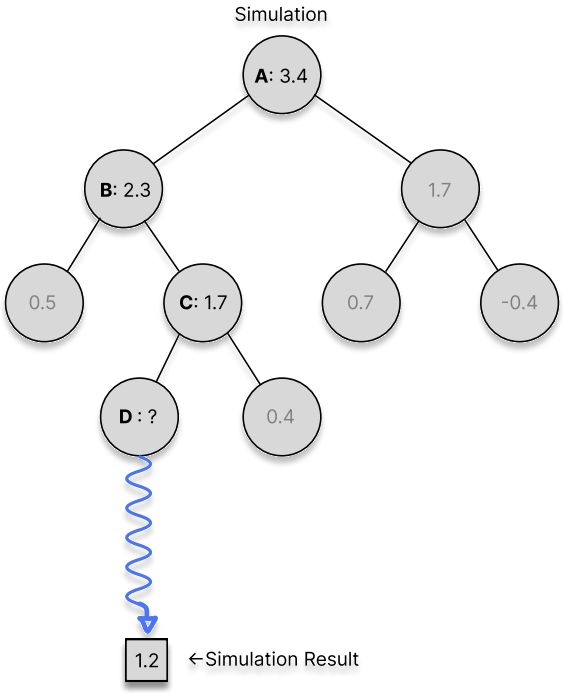

Figure 3.3: A simulation is initiated from the newly added node (D), producing an outcome estimate used to evaluate the state.

4. **Backpropagation:** Take the result of the simulation and propagate it back up the path of nodes selected in Selection, updating each node's statistics (e.g. visit count and total/average reward). This "backup" step informs the node values so that future selections down that path reflect the simulated outcome, see Figure 3.4.

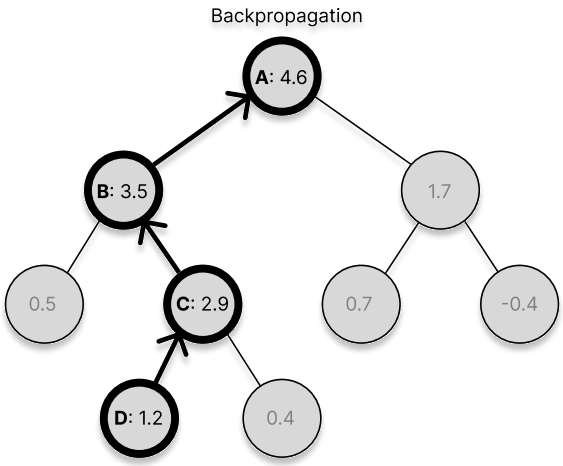

Figure 3.4: The simulation result is propagated up the tree from node (D), incrementally updating the value estimates for nodes (C), (B), and (A).

These four steps repeat until some computational budget is exhausted (e.g. a time

limit or a fixed number of iterations). At this point, the algorithm outputs a move for the root state, typically the child with the highest visit count or win rate. In summary, MCTS gradually grows a game tree in an asymmetric, focused manner: simulations are biased toward promising areas of the tree, and over many iterations the method statistically estimates the value of actions. Because it only requires the ability to simulate random outcomes, MCTS can work "with little or no domain knowledge" [18], but will converge toward optimal decisions given enough iterations. This flexibility and generality underlines its strong performance in a variety of domains.

3.1 UCT Score

At the heart of the MCTS selection phase is often the UCT algorithm (Application of the upper confidence bound to trees), which provides a principled way of balancing exploration and exploitation when choosing among children of a node. UCT was introduced by Kocsis and Szepesvári (2006) as an adaptation of the multi-armed bandit UCB1 algorithm to tree search [18]. The idea is that each move choice can be treated as an independent bandit problem: we have an estimate of that move's value (from past simulations), and we need to decide which move to try next. The UCT rule assigns each child node j an upper confidence bound value that intuitively equals the node's current average reward plus a bonus term (exploration term) that declines as the node is visited more often. One common form of the UCT formula is:

$$UCT_j = X_j + C_p\sqrt{\frac{2\ln N}{n_j}}, \tag{3.1}$$

where X_j is the mean reward(win rate or value) of child j from the simulations so far, N is the total visit count of the parent node, n_j is the visit count of child j, and $C_p > 0$ is an exploration constant that controls the trade-off between exploration and exploitation. The UCB1 formula guarantees that cumulative regret, the difference between the rewards actually received and the one that would have been achieved by always choosing the best course of action, grows only logarithmically over time. This ensures a near-optimal decision efficiency as the number of simulations increases [19]. In practice, C_p is tuned (often $C_p = \sqrt{2}$ or similar) to balance the reward scale. The $\ln n$ means that as the number of visits of the parent node increases, all children's bonuses shrink, but *comparatively*, rarely tried moves (low n_j) get a larger boost. Unvisited moves are usually given an infinite bonus (or a very large number) so that they are tried atleast once. The UCT formula provides a principled way to handle the explore/exploit dilemma: it will eventually sample all moves but will focus more and more on the moves that show higher promise in accumulated simulations. Under ideal conditions, the UCT algorithm is proven to converge to the optimal move given infinite simulations [20]. Although this convergence is primarily of theoretical interest in complex games, the ability of UCT to balance exploration and exploitation effectively was a key reason for the success of MCTS in domains with large branching factors, including Go and later its adaption to chess in systems

like AlphaZero.

3.2 Integration with Neural Network

Modern AI game systems enhance MCTS by integrating deep neural networks into their search functions, combining the strengths of statistical search with learned knowledge. Instead of using random rollouts for simulations, approaches such as AlphaGo Zero and AlphaZero use a neural network to provide a policy (move prior probabilities) and a value (position evaluation), which guide the MCTS. At the start of a simulation (node expansion), the network is consulted to obtain a probability distribution of the moves and an estimated win value for the state [21]. These policy priors are then used in the selection phase to bias the tree search towards promising moves, and the value estimate is used as the outcome of the simulation (replacing a lengthy random playout). In effect, the neural network serves as an informed heuristic for MCTS: the policy network narrows the search by suggesting likely good moves, and the value network evaluates positions without having to play out to the very end [21]. The policy guidance ensures that fewer simulations are wasted on implausible moves, while the value function provides a smoother and more informative evaluation signal than the binary outcomes of random playouts, allowing MCTS to converge toward stronger positions more efficiently. In summary, combining MCTS with deep learning allows the algorithm to learn which moves to explore, continually improving search focus and accuracy over time.

3.3 PUCT: Incorporating Policy Priors into UCT

A crucial innovation in deep learning–guided MCTS is the PUCT algorithm, which stands for Predictor + UCT. PUCT modifies the traditional UCT selection formula (3.1) by injecting the neural network policy prior to the exploration term. UCT balances exploration and exploitation using a generic exploration term, while PUCT refines this process by weighting the exploration bonus with the prior probability $P(s, a)$ predicted by the policy network. This change enables the search to focus more on promising moves early, while still allowing for statistical correction as more simulations are conducted. The selection rule in PUCT chooses the action a for state s that maximizes:

$$PUCT = Q(s, a) + c_{\text{puct}} \cdot P(s, a) \cdot \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}. \tag{3.2}$$

Where the notation is as follows: $Q(s, a)$ is the current estimated value (expected reward) of choosing the action a in state s (based on simulations so far), and $N(s, b)$ is the count of visits from the parent state s (the sum of visits over all actions b from s), and $N(s, a)$ is the visit count for the specific action a. The constant c_{puct} is an exploration parameter (analogous to C_p in UCT) that controls how strongly the

algorithm favors the policy prior, relative to the value $Q(s, a)$. The factor $\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ ensures that the less visited moves are given a larger potential boost, while the factor $P(s, a)$ scales this bonus according to the prior probability of that move[21].

By multiplying the exploration bonus by $P(s, a)$, PUCT effectively prioritizes actions that the policy network deems more likely to be strong, while not losing the balance that UCT provides. If two actions have been tried equally often, the one with a higher prior probability $P(s, a)$ will receive a larger exploration term [21]. In contrast, if a move has a very low prior, its exploration bonus will be correspondingly small. However, PUCT does not blindly follow the policy network. As $N(s, a)$ grows through repeated simulations, the denominator $1 + N(s, a)$ causes the exploration term for that move to shrink. In the long run, when a move has been explored many times, the selection score will be dominated by the term $Q(s, a)$ (the empirical value) rather than the prior [21]. This ensures that the influence of the neural network's prior diminishes as more evidence (simulation data) is gathered, so the algorithm can correct itself if the policy prior was misleading.

Through this mechanism, PUCT achieves a more directed, yet robust search. Early in the search, it biases heavily towards moves that the learned policy suggests are promising, which helps MCTS spend its simulation budget more wisely on plausible moves. Later in the search, as more simulations accumulate, PUCT relies increasingly on actual outcomes (Q values) to fine-tune the decision, so even moves that had a modest prior probability can be chosen if they yield good results. A key example of this approach in action is AlphaGo Zero, where the integration of policy guidance in the tree search was essential to its success [21]. PUCT allowed the system to navigate vast game spaces efficiently, combining the pattern recognition of neural networks with the strategic balance of classical search.

4

Implementation

This chapter describes how the chess engine was implemented and evaluated. It starts with the software environment and programming choices. The structure of the neural network is then presented, followed by the implementation of the Monte Carlo Tree Search (MCTS) and the atomic chess rules. Subsequently, the training process is described, including how the data was generated and used to improve the model over time. The chapter also covers the methods used for performance evaluation, along with the hardware resources used during training and testing.

4.1 Software and Programming Environment

The code is implemented in Python. There are a couple of different factors that influenced this choice. From the research we conducted on neural networks and their software implementations at the start of the project, Python and C++ were determined as the two candidates due to their AI libraries. Most of the project members were much more comfortable with Python. Furthermore, although the group knew that C++ is in general faster, many Python libraries such as Numpy have their functions implemented in C, resulting in faster performance [22]. Using such libraries could reduce the potential performance gap between the two. The group's limited experience with C++ meant that if C++ were to be used as the primary language, lots of time would be spent learning the language, and the risk of bugs and errors would increase. Therefore, it was concluded that the software should be written in Python. For AI libraries, PyTorch was decided upon because the majority of group members had previous experience with it.

4.2 Implementing the Neural Network

Our network is a deep residual convolutional neural network. The terms "residual" and "convolutional" were introduced in the previous chapter in Sections 2.2 and 2.3 respectively. "Deep" means a large number of hidden layers, which are in this case grouped into residual blocks in the middle of the network.

4.2.1 Structure

The structure of the network is visualized in Figure 4.1 and goes as follows:

Input Layer Represents the state of the board as a series of parallel channels, each with dimensions 8×8, with each neuron in a channel representing a square in some way. A channel can, for example, represent the positions of the white pawns another, the black knights, et cetera.

Deep Section By far the largest part of the network, but also very regular. It consists of 20 residual blocks, each containing two convolutions with ReLU activations. Each of these blocks is identical in terms of configuration, that is, in terms of dimensions and hyperparameters. Each convolution takes 256 channels (again, each channel being of size 8×8) and also produces 256 channels.

Output Section There are two outputs: the *policy vector* and the *value*, which are generated by the *policy head* and *value head* respectively. These heads are parallel to each other after the deep section.

- *Policy Head*: Performs a convolution with kernels of size 1×1, then convolves it into 73 channels in order to generate an representation similar to what AlphaZero used as a policy output then flattens this tensor into a 4672-length vector, and then runs a fully connected linear layer to our 1968-length policy vector, as defined in section 4.2.2.

- *Value Head*: Does a 1×1 convolution to one single output channel, which is equivalent to a linear combination of all the channels' individual squares into a single channel. Next, it is flattened to a 64-length vector, then linearly transformed to 256 features, then linearly transformed again to one feature. Finally, a hyperbolic tangent is applied to get a value between -1 and 1.

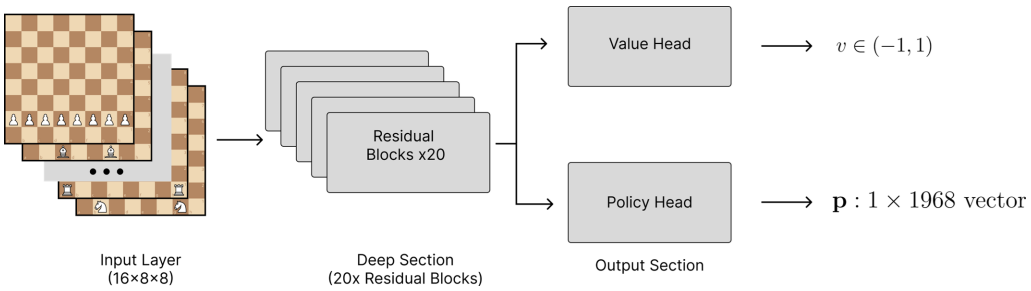

Figure 4.1: The different parts of the network.

4.2.2 Policy vector

To represent moves, an approach similar to LeelaZero was decided [23]. A move consists of 3 parts: the start position of the piece, the end position and in the

case of promotion, what piece gets promoted into. `a7a8q` would represent a pawn moving from square a7 to a8 and promoting to a queen. `b1d3` represents the piece that is currently on b1 moving to d3. That means moving two squares north-east. The hypothetical move `b1d4`, for example, does not exist in the encoding of the policy vector, as no piece can ever make this move. A policy vector of length 1968 was encoded by generating (once, at the start of the program) a list of all of the possible knight and queen moves from each square, as well as all of the promotion moves. This is because every non-knight piece (i.e. pawns, rooks, bishops and the king) can only make moves from a subset of the possible queen moves. This process generated a list of strings called `idx_to_move` that represented the move on each corresponding index of the policy vector. Then a reverse mapping was also generated called `move_to_idx`. This avoided the problem of otherwise having to loop through the whole vector on each move in the MCTS to find the correct index.

4.3 Implementing Monte Carlo Tree Search

MCTS was implemented similar to traditional MCTS with some minor differences in the simulation, selection, and expansion phases. The simulation phase consists of using a neural network to evaluate how good a position is, instead of randomly simulating the game until the game ends. In the selection phase, a PUCT score is used to guide the search. Finally, when expanding the tree from a leaf node with N legal moves, N children are created at once. The child with the highest PUCT score is then visited and its value is backpropagated. When an unvisited leaf child is visited, its value is backpropagated. Only when a leaf child is visited for the second time does the tree expand. The reason behind this design is that it is faster to batch multiple inputs into the neural network, making the MCTS faster.

4.4 Implementing Atomic Chess

When developing a program with the intention of using it to train a neural network, it is crucial that the program is fast and optimized. During the generation of self-play games, extensive pressure is put on the program to generate all available moves from a state as the AI searches down the MCTS tree. Intuitively, one might want to write the program in a way in which the board is represented by a 8×8 2D array. The array may then represent the ranks and files, where each entry contains information about the occupancy of the corresponding chess square: a so-called *square-centric* board representation. However, even when programmed in a way that is computationally effective (for example, mailbox representation), it is still believed to be slower than the more commonly used *piece-centric* approach, often combined with *bitboards* [24]. This section describes the general concept of bitboards and how they are used to generate moves.

4.4.1 Using Bit-Boards

Bitboards are a piece-centric method of representing chess positions, differing from the traditional square-centric approach. Instead of storing information on the occupancy of every square, only the positions of each individual type of piece are stored. Since a chessboard consists of 64 squares, a 64-bit integer is sufficient to tell whether a position is occupied by a piece. For example, a single 64-bit integer can store the positions of all the white-pawns by setting all the bits corresponding to their respective positions to 1, while leaving the bits for empty squares to 0. To differentiate between pieces, color, and information regarding *en passant* and castling, a total of 16 integers were used to store a single board state. Figure 4.2 shows an unsigned 64-bit integer representing the "white pawn"-bitboard in a starting position:

```
wp = 0x000000000000FF000.
```

Figure 4.2: Bitboard for the White Pawns.

The bitboard can be visualized by writing it as a matrix in the form of a chessboard as in Figure 4.3 (the 0s being replaced by dots):

```
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
1 1 1 1 1 1 1 1
. . . . . . . .
```

Figure 4.3: Visualizing the bitboard for the white pawns.

The goal of using such an unorthodox approach is to generate moves by utilizing bit-wise operations, which are executed extremely fast by modern computers.

4.4.2 Generating Moves

Now it is time to show where bitboards get their power, namely: move generation. Using the bitboard for the white king on `h4` in Figure 4.4, we present how one could generate a new bitboard consisting of all available squares to which the white king could move. For the sake of simplicity, we assume that the white king is the only piece on the board.

```
wk = 0x0000000001000000

. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . 1
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

Figure 4.4: Bitboard for the white king.

Consider that we want to access the square above the white king, then we could return this bitboard with a simple bit operation as in Figure 4.1.

```
north = wk << 8
```
(4.1)

The same operation is used to gather the rest of the surrounding squares (left is `wk << 1`, south-west is `wk << 9`, etc.), but there is one problem. Notice that the position of the king is on the edge of the board. Usually, we would access the square to the right by shifting the bitboard of our king one step to the right, but in this case, such an operation would yield the left-most square of the rank directly under our king (specifically, `a3`). The same problem occurs for the north-east and south-east squares; these moves are invalid. Therefore, we need to make sure that we mask them out to mark them as unavailable. To achieve this, the first file is masked out when calculating squares to the right (as in Figure 4.5), as the king cannot enter the first file from the right. The mask we use is a bitboard with `1`'s on the first file and `0`'s elsewhere. The same is done for squares left of the king, with a mask for the last file. Finally, we can return all the different directions of the king (as in Figure 4.6), with the allied occupant squares masked out, as they are not accessible.

```
north_east = (wk << 7) & ~fileA
east       = (wk >> 1) & ~fileA
south_east = (wk >> 9) & ~fileA
```

Figure 4.5: Masking out moves outside of the board.

```
wk_moves = nw | n | ne | w | e | sw | s | se
wk_moves &= ~occupancy_white
```

Figure 4.6: Generating all possible moves for a king, where n, w, e, and s stand for the cardinal directions that the king can move in on the board.

Similar operations are performed for the pawns and the knights; it becomes more difficult when considering sliding pieces, as the occupancy of the board may hinder

further movement of a sliding piece. The operations involved are beyond the scope of this report, but in short they include clever bit operations together with board masks that have been calculated in advance.

Every move initially generated is called a pseudolegal move, as it does not ensure that the move is actually legal. In order for a move to truly be legal, some requirements need to be met. First, it is illegal for a player to put one's own king in check, which means that all pseudo-legal moves must be sifted to remove such moves. This is also relevant in the event of a castle, as a player cannot commit to castling if any of the king's traveling squares are under attack. Second, as a capture in atomic chess blows up all surrounding pieces, it is possible to destroy your own king. It must therefore be ensured that a move does not remove your own king in order to be deemed legal.

4.4.3 Verifying the Implementation

It was very important to ensure that the logic of the atomic chess rules was correct. If the chess engine were to be trained with a rule's implementation that incorrectly judges any legal move as illegal or vice versa, the training would be wasted. The implementation was therefore extensively tested to ensure that there were absolutely no errors present.

The piece logic was implemented sequentially, meaning each piece's move logic was developed singly. Once the pieces were finished, the logic for mechanics such as check: explosions, starting board position, etc. was tested and implemented similarly.

Some games were played manually to uncover obvious issues. With those fixed, the engine was set up to play some self-play matches, playing random legal moves. If any errors occurred, the game and its data was reviewed to find the bug and fix it.

Finally, 100,000 atomic chess games, originally played by human users on lichess.org during February 2025, were downloaded from the lichess open *database* [9] and replicated move-by-move by the engine. This large-scale test aimed to uncover any last rare edge-case bugs. Once these issues were resolved, it was deemed that the implementation was sufficiently verified to confidently proceed with the training.

4.5 Training

With all components put in place, the network is finally ready to begin training and improve its chess-playing performance; the ultimate goal of the report. To do so, the model must develop a general understanding of which moves are good and which are not. Although the learning process might initially be unclear, a combination of qualitative training data and effective learning techniques enables the model to make progress.

4.5.1 Generating Training Data

In order for the model to learn and improve, it needs data to train on. This data is generated by having the model play a set number of games against itself. Since each self-play game is independent, they are played in parallel to accelerate the process. When deciding which move to play, an MCTS is initialized with the current board state set as the root node. Between different moves, the root node is changed to its child node that corresponds to the move made. This results in reusing the relevant part of the MCTS tree, which speeds up subsequent searches. Upon testing, an MCTS consisting of 800 searches (starting from an empty tree) with a c_{puct} value of 2 took approximately 4.5 seconds on an L40s node described in Section 4.7. Of this time, 65% was spent evaluating the neural network.

Once a set number of MCTS explorations have been completed, a vector $\boldsymbol{\pi}$ is returned that contains the distribution of visits between all possible moves. Recall from 3.3 that the idea is that a higher visit count corresponds to a better move. The result of the MCTS explorations is a more accurate prediction of which moves are better and which moves are worse. To make sure that the model generates a variety of games, the chosen move is not always the one with the highest probability, but instead each move has a chance based on its individual probability. Each game is played until it is terminated, either by a player winning the game or by the game ending up in a draw. The result of the game is called the *outcome* of the game and is denoted by z. After all the games are played, the last step is to convert the games into data points that the model can actually use.

In order for the self-played games to be useful to the model, the games are separated into all their individual states. Having the states, they are grouped with the MCTS predictions for that state together with the outcome of the game. The outcome being a scalar of 1 or -1 depending on whether white or, respectively, black ended up winning, with 0 accounting for a draw. Figure 4.7 shows the definitive structure of a data point, where *state* represents a board position as a matrix of bitboards, $\boldsymbol{\pi}$ represents the probability vector given by the MCTS, and z represents the result of the game as a scalar value. These data points contain valuable information about each played state of the board, making them suitable for training a model.

$$(\text{state}, \boldsymbol{\pi}, z)$$

Figure 4.7: Structure of a data point for training.

4.5.2 Learning from the Training Data

Once the data points are collected, training can begin. Recall that a data point consists of the state of a board, together with the MCTS predictions and a determined outcome. Note that although the MCTS predictions were based on the initial predictions of the model, the search process updated them to better match each move's

true worth. They may therefore be regarded as more accurate than the predictions of the model alone and are the underlying reason for why the training works.

Because of the key supposition that the MCTS predictions are more accurate than the model alone, it makes sense to try and have the model predict these values rather than its initial guess. To help guide the shift of the weights in the right direction, the model uses a *loss function*. The loss function is used to determine how to adjust the weights of the neural network to best match the MCTS-assisted predictions. The loss function is as follows:

$$L = (z - v)^2 - \boldsymbol{\pi}^\intercal \log(\mathbf{p}) + c||\boldsymbol{\theta}||^2 \tag{4.2}$$

where $\boldsymbol{\pi}$ and z are the MCTS predictions and the respective outcome (the training data) from a given state[1], $\mathbf{p}$ and v are the model's move predictions and its guess of the outcome, respectively[2] (the model's guess), c is a real-valued constant, and $\boldsymbol{\theta}$ is the vector of all weights and biases in the model [3].

The first term is known as the squared error of z and v (also commonly known as the mean squared error when calculating several data points), the second term is the cross-entropy loss between $\boldsymbol{\pi}$ and $\mathbf{p}$, and the third term is called L_2 *regularization*. This last term counteracts overfitting and parameter explosion by not allowing any specific weights and biases to become too large. The loss function written in our program code only includes the first two terms, while the third term is implied by a parameter called *weight_decay* in the SGD module of PyTorch (see Section 2.1.2 for information on SGD).

The aim is to reduce the loss function for each data point as much as possible, which translates into having the model guess the MCTS predictions and the outcome as close to true as possible. Having calculated the loss for all data points, the weights of the neural network are updated accordingly to minimize the total loss of all data points (more extensively touched on in 2.1.2).

4.5.3 Model selection

After generating the self-play games, the resulting data are split into training and validation sets. 80% of the self-play data is used to train the model, while the remaining 20% is used for validation by computing the loss without updating the weights. To ensure optimal performance, the model with the lowest validation loss is saved at the end of each generation and used to generate self-play games for the subsequent generation. If the validation loss of the current generation does not improve on that of the previous generation, the final model of the current generation is still saved and used for further training. The validation loss, mentioned in Section 4.5.2, is a measure of how different the prediction of the neural network is from the

[1] z being a scalar of -1, 0 or 1, representing who won.

[2] Unlike z, v is an estimated guess and therefore lies in the range $(-1, 1)$

selection of the MCTS search.

4.6 Evaluating the Result

There are two main ways in which we can evaluate our results: Performances in matches against previous versions and performances against other engines. Validation loss was in some part used to measure improvement in the training process and to choose which neural network models were saved. However, the strength of engines is measured in actual gameplay. Therefore, to assess the practical playing strength, models from different stages of the training process are evaluated through direct competition. These internal matches help identify which version performs best during games. Furthermore, to ensure that the model's tactics are not exploiting "its own idiosyncrasies" (or previous generations'), the selected models are also tested against external engines. This provides a more robust evaluation of general playing strength and guards against overfitting to self-play scenarios.

4.6.1 Internal tournament

For internal matches, selected models were evaluated against each other, an untrained version, and one model that just chooses random legal moves. Each model played one match against every other model, and this was repeated a certain number of rounds. To ensure that the matches were fair, for each match-up, every model started every other match. Therefore, an even number of rounds was played so that each model got to start an equal number of matches. The models will then be ranked depending on their score, which is calculated as a one point for every win, half a point for every draw and a zero new points for a loss.

4.6.2 Evaluation against external engines

The top three models from the internal tournament were evaluated against Fairy-Stockfish to measure their playing strength against external engines. From previous testing, the engines had already done very well against Fairy-Stockfish level 1 but struggled more against levels 2 and 3. Therefore, the 3 models were evaluated against level 2 and 3.

4.7 Hardware

Minerva, a compute cluster that is hosted by the CSE department at Chalmers/GU, was used to train the neural network and evaluate the chess engine. Minerva has many nodes available, with two different configurations, which are shown in Table 4.1.

Nodes	CPU	Memory	GPUs	Hard disk(s)
L4 node(s)	INTEL(R) XEON(R) GOLD 6548N (2×64 cores)	512 GiB	8×NVIDIA L4 (24 GiB)	2×1.92 TB SSD (RAID-1)
L40s node(s)	INTEL(R) XEON(R) GOLD 6548N (2×64 cores)	512 GiB	4×NVIDIA L40s (48 GiB)	2×1.92 TB SSD (RAID-1)

Table 4.1: The specs of the compute cluster used for training and evaluation.

5

Result

This chapter presents the results of the implementation. It includes a description of how the models were evaluated and the subsequent results. The hyperparameters used during the training are presented. Finally, it concludes with a brief explanation of the interface that allows external engines and players to compete against our engines.

5.1 Evaluation

For the tournament specified in Section 4.6.1, eight trained models, one untrained model, and a bot that only played random moves were entered. The models played each other 20 times each, for a total of 180 rounds per model. We use the term model, apart from Random_Moves, as each participant in the tournament is the same engine, just having been trained on different numbers of data points. The models playing in the tournament are described in Table 5.1. During the tournament, all models, with the exception of Random_Moves, had a search count of 800. It was 800 to maintain consistency between the evaluation and the training, where the search count was 800 as well (as mentioned in Section 4.5.1). Likewise, all models except Random_Moves used a c_{puct} value of 1 when selecting moves using the $\boldsymbol{\pi}$ vector from the MCTS. In contrast to the idea of maintaining consistency, a different c_{puct} value is chosen for evaluation than for training. As explained in Section 3.3, a higher c_{puct} value encourages exploration, while a lower value instead prioritizes moves that are more explored and "proven". Therefore, we decrease the c_{puct} value to 1. Before selecting the final move using $\boldsymbol{\pi}$ as a probability distribution, each element in $\boldsymbol{\pi}$ was first raised to the power of 4 and then normalized to skew the distribution toward stronger moves.

Model	Datapoints trained on (millions)	Generations	Validation loss
8_8	4.72	30	1.40
8_7	4.16	27	1.26
8_6	3.59	24	1.25
8_5	3.06	21	1.25
8_4	2.33	17	1.29
8_3	1.49	12	1.32
8_2	0.83	8	1.30
8_1	0.39	5	1.69
8_Untrained	0	0	–
Random_Moves	–	–	–

Table 5.1: Description of the different models playing in the tournament. Models with a name beginning with *8_* use MCTS with a neural network to select its moves, the only difference being the neural network's parameters. The model Random_Moves selects a move randomly from a list of legal moves using a uniform probability distribution.

Model	(W/D/L)	Final score
8_5	133/4/43	135
8_8	113/16/51	121
8_6	116/5/59	118.5
8_3	114/8/58	118
8_4	111/13/56	117.5
8_7	100/13/67	106.5
8_2	84/8/88	88
8_1	70/5/105	72.5
8_Untrained	13/3/164	14.5
Random_Moves	7/3/170	8.5

Table 5.2: Result of the tournament, in order of score. A win grants 1 point, a draw 0.5 points and a loss 0 points.

Every single trained model won 20 out of their 20 games against Random_Moves and, except for one draw between model 8_7 and 8_Untrained, the same was true against the untrained model. This result unarguably shows that the model improved from training.

Interestingly, the most trained model did not perform the best in the internal tournament. The progression of the relative strength of the different models can be viewed in figure 5.1.

The top three models of the internal tournament were then evaluated against an external engine known as Fairy-Stockfish[25]. Once again, each model played 20

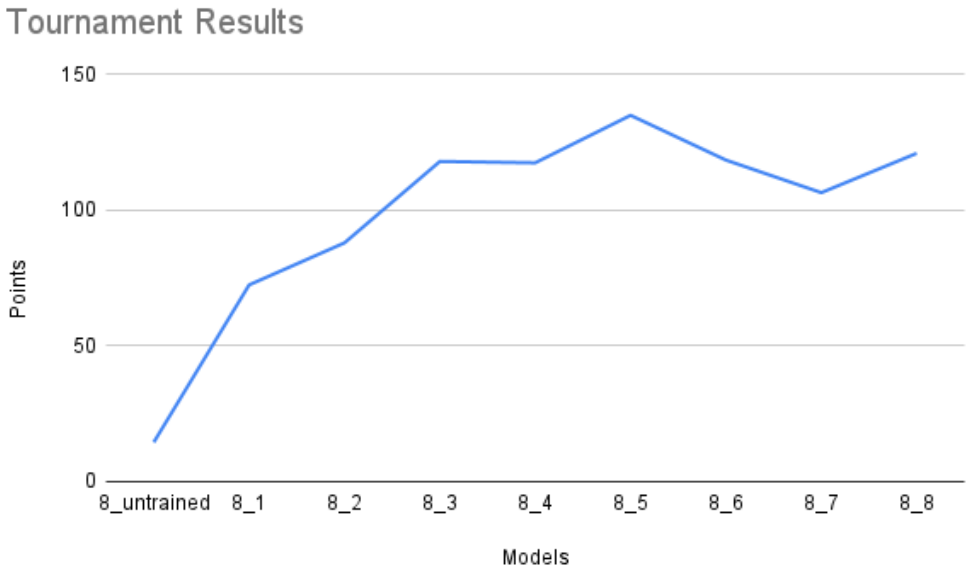

Figure 5.1: Tournament result of the different models in order of least to most trained. 8_1 is the predecessor of 8_2, and so on. Therefore it can be viewed as the progression of the neural network's strength through training. Note: The difference in number of datapoints trained on between the models in the x-axis is not uniform.

matches against their opponent, Fairy-Stockfish level 2. The c_{puct} value and search count was the same for each model as in the earlier internal tournament. The moves were chosen by selecting the node with the highest number of visits in $\boldsymbol{\pi}$. The result is shown in table 5.3. Models 8_8 and 8_5 did very similarly, while model 8_6 did considerably worse. The unexpected result therefore seems to persist, model 8_5 does better than the more trained model 8_6, both against internal and external engines.

Model	**(W/D/L)**	**Final score**
8_8	13/0/7	13
8_5	12/0/8	12
8_6	7/2/11	8

Table 5.3: The scorelines of the top three models from the internal tournament, against Fairy-Stockfish level 2. A win grants 1 point, a draw 0.5 points and a loss 0 points.

Against Fairy-Stockfish level 3, the three models did considerably worse.

We increased the difficulty, matching the same three models against Fairy-Stockfish level 3 with the same settings as when they played against level 2. The best result was 4 wins and 16 losses for model 8_5. The results can be found in 5.4. One of the wins and one of the losses for model 8_8 can be viewed in Portable Game Notation

(PGN) in appendix A.

Model	(W/D/L)	Final score
8_5	4/0/16	4
8_6	3/0/17	3
8_8	2/0/18	2

Table 5.4: The scorelines of the top three models from the internal tournament, against Fairy-Stockfish level 3. A win grants 1 point, a draw 0.5 points and a loss 0 points.

Since the change in performance of the different models for our engine was so drastic, we decided not to evaluate the models against any higher levels of Fairy-Stockfish.

5.2 Hyperparameters

The model generated 250 self-play games at the beginning of each generation. As stated in Section 4.5.1, these games were played in parallel using 25 concurrent processes. To avoid training on low quality data, i.e. excessively long matches with redundant moves, a maximum limit of 150 moves per game was implemented. The model was then trained on the resulting data from the self-play games for 10 epochs, using a batch size of 256. Optimization was performed using stochastic gradient descent (SGD) with momentum and a learning rate scheduler. The momentum coefficient was set to 0.9, and the weight decay parameter was 0.0001.

Substantial effort was devoted to selecting an optimal learning rate and scheduler. AlphaZero, for example, used a starting learning rate of 0.2 and then decays it three times with a factor of 0.1 [3]. However, applying 0.2 and 0.1 as the learning rate during training caused both the training and validation losses to explode. Therefore, it was decided to set the initial learning rate to 0.05.

Initially, a scheduler was implemented that reduced the learning rate after a predetermined number of epochs with a fixed factor. Although this strategy drove both the learning rate and the validation loss to very low values, the resulting models performed poorly in gameplay. To adapt more dynamically to the model's actual progress, PyTorch's ReduceLROnPlateau scheduler was implemented. The scheduler reduced the learning rate by a factor of 0.5 when the validation loss did not improve over three consecutive epochs, that is, when a plateau was detected. As the validation loss was not a definitive indicator of playing strength, and considering that AlphaZero's lowest learning rate during training was 0.0002, a minimum threshold of 0.0001 was enforced. The progression of validation loss and the learning rate across epochs is shown in Figure 5.2.

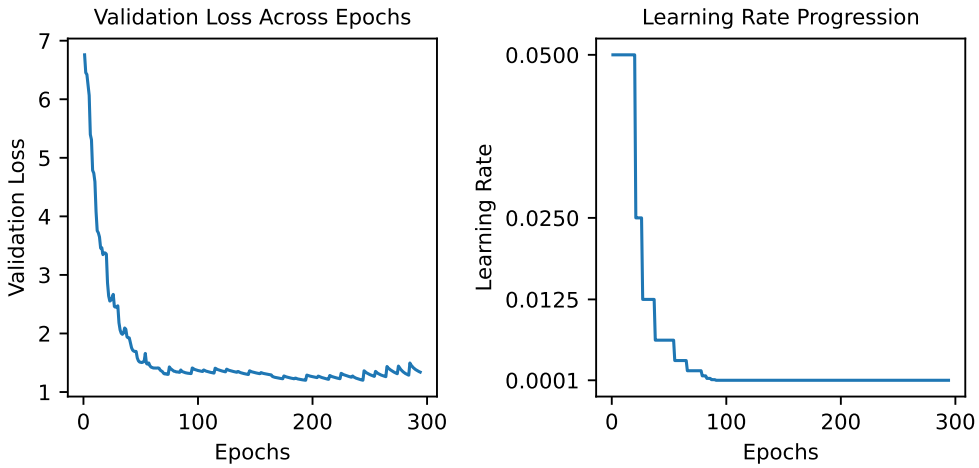

Figure 5.2: Learning rate progression and validation loss across epochs. Note that each training generation consists of 10 epochs. This means that the training data and the validation data are different every 10 epochs.

5.3 Lichess Integration and Interface

It is possible to play against the engine and the engine can play matches with other engines through lichess.org. This functionality was enabled using the lichess Bot API [26], which facilitated connecting our engine to lichess and setting up matches with Fairy-Stockfish.

6

Discussion

This chapter reviews the project results with a focus on how the engine behaved during training and evaluation. Particular attention is paid to areas where limitations become apparent and decisions made during implementation that had a notable impact on performance. In addition to identifying challenges, this chapter outlines possible improvements that could be explored in future research efforts.

6.1 Analyzing the Evaluation Results

The results show that the trained models were better than the untrained model and the engine playing random moves. Interestingly, model 8_5 achieved the highest score in the internal tournament and performed similarly well when evaluated against external engines. This was a bit surprising, considering that the only difference between model 8_5 and 8_7, for example, was that it had trained less. Model 8_5 did have the joint-lowest validation loss at 1.25, but did much better than model 8_7 which has a validation loss of 1.26. The reason could possibly be that we do not decrease the learning rate below 0.0001. This threshold is reached long before model 8_8 and the learning may therefore have stagnated.

Due to time constraints, we did not keep training the model. If we look at figure 5.1, we see that the model gets progressively better again after the big downturn in performance from model 8_5 to model 8_6. If there had been more time for training, it is possible that the trend would have continued and that a later model would surpass model 8_5.

6.2 Limitations and Potential Improvements

Throughout the development of this project, several limitations in the implementation were identified, primarily related to the efficiency of the training process and the consistency of the engine's play during evaluation. This chapter aims to discuss some of these challenges encountered, how they were solved, and possibly how they could be improved upon.

6.2.1 Speed of the Self-Play Phase of Training

A big bottleneck of our implementation is the speed with which self-play games are completed. An MCTS of 800 searches took approximately 4.5 seconds, as described in Section 4.5.1. In our experience, the average length of a self-play game varied greatly. However, for estimation purposes, we can assume an average of 60 moves. This meant that a single self-play game could take more than 4.5 minutes to generate. Both parallelization of the self-play phase by utilizing a worker pool and optimizations done to the MCTS algorithm increased the performance significantly, but it was still slow. The usage of Python to implement the MCTS and the underlying chess representation is thought to be the main reason why it is slow. When investigated, approximately 65% of the time inside the MCTS is spent evaluating the neural network. Ideally, most of the time should be spent evaluating the neural network.

During self-play, a problem arose because some games were very long (> 200 moves), which slowed training a lot. Because we used a c_{puct} value of 2 in the PUCT formula, the MCTS favored a more balanced exploration of the game tree. This led to the probabilities of the legal moves from the $\boldsymbol{\pi}$ vector being relatively uniform (compared to how they would look with a lower c_{puct} value). Because this probability distribution was used to choose the move, this could lead to the engine playing suboptimally, which is thought to be the reason why some games during self-play were very long. To combat this issue, the fifty-move rule and three-fold repetition were introduced, which showed promising results in reducing the occurrences of long games. A threshold of maximum 150 moves was also implemented. If this limit was reached, the self-play game would automatically be restarted to ensure that the training process did not stall.

Another reason why we chose to discard long games is that, based on simple reasoning and observation, the quality of these games were generally poor. They often ended with both colors seeming to aimlessly move their kings. We determined that the engine would be better off not training on these long games.

6.2.2 Enhancing the Training Data

One notable observation from the games against Fairy-Stockfish was that our engine rarely won beyond the 30-move mark. Currently, the engine has only been trained on games starting from the standard starting position. A potential improvement could involve generating a portion of self-play games that begin directly from end-game positions. However, a challenge arises from the fact that the engine plays atomic chess. Atomic chess has not been studied as extensively, and therefore there is a scarcity of documented endgames. Instead, these end-game positions would likely have to be found or created by the group itself.

As previously mentioned, we were unable to train the model for as many generations or with as many data points as we had hoped. Generating the games during each generation was the most time-consuming stage during training. To be able to

train on more data points per generation without increasing the runtime, we could introduce a game buffer. That is, store the games produced in one generation and reuse them for several subsequent generations.

6.2.3 Balancing Randomness with Strong Move Selection

When evaluating the performance of the engine playing at full strength, it can reasonably be assumed that selecting the move from the vector $\boldsymbol{\pi}$ with the highest probability would correspond to playing a strong move. However, since our engine's MCTS uses a neural network instead of a random play-out phase, it is ultimately deterministic in theory. When tested, two different versions of our engines playing against each other, while using this method to choose the strongest move, would consistently play the same games. Comparison of engine strength across multiple games is thus pointless, and some degree of randomness is needed.

In the self-play phase of the training, the vector $\boldsymbol{\pi}$ is used as a probability distribution to choose the move. However, even with a lower c_{puct} value in the PUCT, choosing moves in this way still posed a high risk of selecting weaker moves. To minimize this, skewing the probability distribution towards the stronger moves by applying a power of 2 or higher and then normalizing the vector seemed to work. It remains unclear whether this is the optimal way to choose a move or if there is a better way.

6.2.4 Finding the Optimal Hyperparameters

A lot of future work is possible regarding finding the optimal hyperparameters for the neural network. Most of our work was spent adjusting the optimizer parameters, where the learning rate and its progression received the most attention. Looking at Figure 5.2, we can see that the 2 graphs have similar shapes, which is reasonable since the progression of the learning rate was tied to the validation loss not decreasing. It remains unclear whether the steep decline was optimal or if a slower progression would have been better. It is also theorized that the minimum threshold learning rate of 0.0001 should have decreased further towards the last generations, since the validation error increased slightly.

It is also unclear whether 250 self-play games per generation was sufficient or if it should have been increased. The idea was that more games would lead to more diversity in the training data of each generation, resulting in a more balanced model. However, since fewer self-play games would result in each generation training more quickly, it could be argued that the next generation should begin training earlier with a possibly more improved model (in terms of playing strength). In theory a stronger model generates better data points because the games are of higher quality.

6.2.5 Network Structure

One significant question regarding potential future research in this area is that of network structure, and more specifically network depth. This project follows AlphaZero in using a total of 40 convolutional layers in the deep section, but one could

hypothesize that chess variants with smaller game trees, which includes atomic chess, may not require as many convolutions as standard chess in order to optimally improve its performance. Future work could include exploring whether a significant reduction in the number of parameters and a more efficient training phase can be achieved by reducing this number.

6.3 Analyzing the Engine's Gameplay

When testing our engines against Fairy Stockfish it became evident that it had difficulty identifying indirect threats to its own king. This resulted in many instances where the opponent's threat to explode a piece adjacent to the king in the following move was overlooked, thus leading to a loss. In theory, MCTS with enough searches should be able to identify indirect threats. This could be a result of the output of the neural network not being ideal and misguiding the search, or that simply not enough searches were made.

6.4 Societal and Ethical Considerations

Some of the possible points that could be discussed in this section include cheating and energy use. The environmental impact of this project is minimal; all model training and evaluation were performed using a single GPU and CPU on a shared computational cluster. Cheating is also not a significant concern in this context. The engine is not very strong and is compatible only with atomic chess. Due to the insignificant impact of our project on these factors, it was decided that these questions are not relevant to our project.

6.5 Use of Generative AI

AI, more specifically ChatGPT and writefull (built-in grammar and spelling checker on overleaf), were used to proofread and identify spelling and grammatical errors in the report. These tools were only used to review language and suggest corrections and were not used to generate any of the text included in the report.

7

Conclusion

This thesis presented the development of a self-trained atomic chess engine. The engine consists of a slightly modified Monte Carlo tree search utilizing a deep convolutional neural network to evaluate positions in the simulation phase. The results show that it improved its performance over time by solely playing games against itself. These results show that it is possible to apply reinforcement learning through self-play to atomic chess.

Future work may explore different network structures, and explore possibilities such as shallower networks and training self-play games starting at endgame positions. Reducing network depth could lead to faster training times, especially given that atomic chess may not demand as much representational capacity as standard chess. Generating endgame positions manually or semi-automatically may also help address the current imbalance in the training data. Additionally, improvements to the self-play phase, such as a faster MCTS implementations, could significantly improve training efficiency. Finally, tuning hyperparameters like the learning rate schedule or the number of self-play games per generation remains an open area for experimentation.

Bibliography

[1] McKinsey & Co., *The state of ai in early 2024: Gen ai adoption spikes and starts to generate value*, 2024. [Online]. Available: https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai, Accessed: Feb. 13, 2025.

[2] M. Hamzah, *Game theory: How stockfish mastered chess*, 2022. [Online]. Available: https://blogs.cornell.edu/info2040/2022/09/30/game-theory-how-stockfish-mastered-chess/, Accessed: Feb. 11, 2025.

[3] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: 1712.01815 [cs.AI]. [Online]. Available: https://arxiv.org/abs/1712.01815.

[4] History of Chess, *The Origins of Chess*, 2011. [Online]. Available: https://web.archive.org/web/20110514202930/http://history.chess.free.fr/origins.htm, Accessed: Mar. 16, 2025.

[5] Chess.com. "Origins of Chess." (2025), [Online]. Available: https://www.chess.com/article/view/origins-of-chess. Accessed: Mar. 16, 2025.

[6] FIDE. "FIDE Member Federations Directory." (2025), [Online]. Available: https://directory.fide.com/list/member_federations/main?show=continent. Accessed: Mar. 16, 2025.

[7] S. Team, *Introducing NNUE Evaluation*, Aug. 2020. [Online]. Available: https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/, Accessed: Apr. 1, 2025.

[8] lichess.org. "Atomic." (2025), [Online]. Available: https://lichess.org/variant/atomic. Accessed: Apr. 29, 2025.

[9] Lichess. "Lichess database." (2025), [Online]. Available: https://database.lichess.org/#variant_games. Accessed: Mar. 13, 2025.

[10] Chessgames.com. "Chess statistics." (2025), [Online]. Available: https://www.chessgames.com/chessstats.html. Accessed: Apr. 29, 2025.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2017, ISBN: 978-0-262-03561-3. [Online]. Available: https://www.deeplearningbook.org/.

[12] D. Kriesel, *A Brief Introduction to Neural Networks*. 2007. [Online]. Available: available%20at%20http://www.dkriesel.com.

[13] H. H. Aghdam and E. J. Heravi, *Guide to Convolutional Neural Networks*. Springer, 2017. DOI: 10.1007/978-3-319-57550-6.

[14] G. Algeskog, F. Erngård, O. Forsberg, N. Ivarsson, Z. Leesment, and T. Muusha, *A self-trained engine for a chess-variant*, 2024. [Online]. Available: https://www.cse.chalmers.se/~abela/AntiChess2024.pdf, Accessed: Apr. 2, 2025.

[15] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1412.6980.

[16] P. Contributors, *Conv2d - pytorch 2.6 documentation*, 2024. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html, Accessed: Apr. 6, 2025.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[18] C. B. Browne, E. Powley, D. Whitehouse, *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. DOI: 10.1109/TCIAIG.2012.2186810. [Online]. Available: https://doi.org/10.1109/TCIAIG.2012.2186810.

[19] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2–3, pp. 235–256, 2002. DOI: 10.1023/A:1013689704352.

[20] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Proceedings of the 17th European Conference on Machine Learning (ECML)*, ser. Lecture Notes in Computer Science, vol. 4212, Springer, 2006, pp. 282–293. DOI: 10.1007/11871842_29. [Online]. Available: https://doi.org/10.1007/11871842_29.

[21] D. Silver, J. Schrittwieser, K. Simonyan, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017. DOI: 10.1038/nature24270. [Online]. Available: https://www.nature.com/articles/nature24270.

[22] N. Developers. "What is numpy?" (2025), [Online]. Available: https://numpy.org/doc/stable/user/whatisnumpy.html. Accessed: May. 18, 2025.

[23] Leela Chess Zero Developers, *Leela Chess Zero Wiki*, https://github.com/LeelaChessZero/lc0/wiki, Accessed: Feb 18, 2025.

[24] phhnguyen, *Neural network training discussion*, 2025. [Online]. Available: https://talkchess.com/forum3/viewtopic.php?f=7&t=76773#p885878, Accessed: Mar. 17, 2025.

[25] Fairy-Stockfish Developers. "Fairy-stockfish." (2025), [Online]. Available: https://fairy-stockfish.github.io/. Accessed: May. 19, 2025.

[26] lichess-bot-devs. "Lichess bot." (2024), [Online]. Available: https://github.com/lichess-bot-devs/lichess-bot. Accessed: Apr. 2, 2025.

A

Our engine v.s. Fairy-Stockfish level 3

A.1 Win

[Site "https://lichess.org/KUdIDncd"]
[Date "2025.05.19"]
[White "lichess AI level 3"]
[Black "OppenheimerZero"]
[Result "0-1"]
[Variant "Atomic"]

```
1. Nf3 e5 2. d3 Nh6 3. Bg5 f6 4. e4 Ng4 5. Kd2 Bb4+ 6. Kc1 Bd2#
{ Black wins by checkmate. } 0-1
```

A.2 Loss

[Site "https://lichess.org/9rhgsszX"]
[Date "2025.05.19"]
[White "lichess AI level 3"]
[Black "OppenheimerZero"]
[Result "1-0"]
[Variant "Atomic"]

```
1. e4 b5 2. Qh5 g6 3. Qg5 f6 4. Qd5 Bh6 5. Qxd7#
{ Game ends by variant rule. } 1-0
```