CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

A Self-Trained Engine for a Chess-variant

Combining a Neural Network and Monte Carlo Tree Search to Create an Antichess Engine

Bachelor's thesis in Computer science and engineering

Gustaf Algeskog
Felix Erngård
Olof Forsberg
Nils Ivarsson
Zackarias Leesment
Tariro Muusha

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Bachelor's thesis 2024

A Self-Trained Engine for a Chess-variant

Combining a Neural Network and Monte Carlo Tree Search to Create an Antichess Engine

Gustaf Algeskog
Felix Erngård
Olof Forsberg
Nils Ivarsson
Zackarias Leesment
Tariro Muusha

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

A Self-Trained Engine for a Chess-variant
Combining a Neural Network and Monte Carlo Tree Search to Create an Antichess Engine
Gustaf Algeskog Felix Erngård Olof Forsberg Nils Ivarsson Zackarias Leesment Tariro Muusha

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiners: Patrik Jansson and Arne Linde, Department of Computer Science and Engineering
Graded by teacher: Niklas Broberg, Department of Computer Science and Engineering

Bachelor's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: AI-generated image of chess AI

Typeset in LaTeX
Gothenburg, Sweden 2024

A Self-Trained Engine for a Chess-variant
Combining a Neural Network and Monte Carlo Tree Search to Create an Antichess Engine

Gustaf Algeskog, Felix Erngård, Olof Forsberg, Nils Ivarsson, Zackarias Leesment, Tariro Muusha

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

Abstract

This work is the development of an Antichess engine using a neural network in the style of AlphaZero. Our work shows the applicability of an AlphaZero-style engine applied to a Chess variant, using Monte Carlo tree search and a neural network to guide the search. The final neural network uses a residual network architecture with 4 residual blocks, and training of the neural network has been done through games of self-play. Our results show that additional training would have increased the networks accuracy further. The engine is capable of play against intermediate opponents, however it lacks the skill to face more skilled opponents and engines. The neural network managed to significantly improve the positions searched by Monte Carlo tree search when compared to a flat search, making it more accurate at the same number of positions searched. However, it was unable to gain a competitive edge due to the slow execution speed of the neural network.

Keywords: Artificial Intelligence (AI), Computer Chess, Chess, Antichess, Neural Network, Convolution, Monte Carlo tree search (MCTS), Machine learning

Sammandrag

Detta arbete presenterar utvecklingen av en Antichess-dator som använder ett neuralt nätverk i stil med AlphaZero. Arbetet visar tillämpligheten av en schakvariantdator i AlphaZero-stil som tillämpas på en schackvariant, med hjälp av Monte Carlo tree search och ett neuralt nätverk för att vägleda sökningen. Det slutliga neurala nätverket använder en residual nätverksarkitektur med 4 residual blocks, och träning av det neurala nätverket gjordes på självspelande partier. Vårt resultat visar att ytterligare träning hade ökat nätverkets prestanda. Antichess-datorn kan spela mot medelskickliga motståndare, men den saknar förmågan att möta skickligare motståndare och datorer. Det neurala nätverket lyckades avsevärt förbättra de positioner som söktes med Monte Carlo tree search jämfört med en nätverksfri sökning, vilket gjorde den bättre vid samma antal sökta positioner. Nätverkets fördelar begränsas dock av den långsamma exekveringshastigheten gentemot standard Monte Carlo tree search.

Nyckelord: Artificiell intelligens (AI), Dator Schack, Schack, Antischack, Neurala nätverk, Konvolution, Monte Carlo Tree Search (MCTS), Maskininlärning

Acknowledgements

We are grateful to our supervisor Andreas Abel for his support and insights throughout the project, going beyond what is expected of a supervisor. We would also like to thank Arne Linde for his generosity in lending us access to additional computational resources.

Gustaf Algeskog, Felix Erngård, Olof Forsberg, Nils Ivarsson,
Zackarias Leesment, Tariro Muusha, Gothenburg, October 2024

Contents

List of Figures

List of Tables

1

Introduction

Chess is a strategic two-player board game, where each player commands a set of 16 Chess pieces, with the goal of checkmating the opposing player's king. It originated from 6th century India but has evolved throughout the years and the version known today appeared in 15th century Europe [1]. Alongside the evolution of Chess, several variations of the classical game have emerged. Today, it is estimated that more than 2000 variants of Chess exist. More popular ones, such as Antichess, are being actively played worldwide [2].

1.1 Chess Engines

The aspiration of creating a self-playing Chess machine has existed for centuries. One example is *The Mechanical Turk*, an early Chess-playing machine constructed in 1770 that played strongly against human opponents for 84 years [3]. The machine however turned out to be an elaborate hoax with a human Chess player concealed inside. Another attempt was made in 1950 when Alan Turing created a Chess algorithm called *Turochamp*, but lacked a sufficiently strong computational device to run it [4].

In 1985 a group of graduate students in computer science at Carnegie Mellon University created *Deep Thought* as an unofficial project [5]. The computer used an evaluation function to predict the winning player by looking at pieces, positions, how well-guarded the king was, etc. This was one of the predecessors upon which today's engines are built.

In 2008, an open-source engine called *Stockfish* was developed and quickly emerged as one of the strongest engines in the world [6]. Almost a decade later, in 2017, the company DeepMind created AlphaZero, a system capable of independently teaching itself how to master various board games [7]. Unlike Stockfish, which relied on numerous rules and heuristics for evaluating positions, AlphaZero was pre-trained solely on the basic rules of Chess. Instead, AlphaZero utilised a deep neural network to approximate the best move [8]. It achieved this by playing games against itself, called self-play; learning the best moves by trial and error through reinforcement learning.

At first, AlphaZero played moves entirely at random; subsequently updating the neural network's parameters according to wins, losses, and draws, thus increasing the probability of picking advantageous moves in future game iterations [7]. To do this efficiently, AlphaZero utilised the *Monte-Carlo tree search (MCTS)* algorithm, which is guided by the network. It only searches 60 thousand positions per second, compared to Stockfish's 60 million positions per second. Despite analysing significantly fewer positions, AlphaZero surpassed Stockfish's performance. AlphaZero played 1000 games against the 2016 Chess champion, Stockfish, winning 155 games and losing only 6 of them with the rest being draws.

1.2 Purpose

The purpose of the thesis is to create a self-trained Chess engine for a Chess variant. By laying the foundations using a neural network and a game tree search, an engine should emerge, able to play against itself and humans.

Through self-play, the engine should improve its knowledge of the Chess variant by leveraging unsupervised learning. Self-play will continuously adapt play according to the outcome of its games; that is, the engine should evaluate the outcome of a game based on the moves it played to improve its performance further. Additionally, this improves the adaptability of the Chess engine as it can be re-calibrated to learn other variants of Chess or board games without changes to the core architecture.

1.3 Scope

The project will be limited to focus on developing an engine capable of self-play for the Chess variant Antichess. This variant was chosen because of its characteristics, leading to a significantly reduced game tree. A reduced game tree is beneficial as it reduces the number of learnable game states for the neural network. This results in less computational power required to achieve better performance, compared to more advanced Chess-variants.

2

Theory

This chapter introduces relevant theory and background knowledge in order to understand the implementation of the self-playing Antichess engine. First, the characteristics of Antichess are presented in 2.1 to highlight why Antichess was chosen and 2.2 explains a system for representing the relative skill level of players in Antichess. Throughout 2.3–2.8, theory for understanding the implementation of the engine is presented along with challenges in training sophisticated neural networks and solutions to mitigate these issues. Lastly, 2.9 describes how game tree search and neural networks can be combined to produce Chess engines.

2.1 Antichess

Antichess is a Chess variant originating from 1876 by the name of "Take Me Chess" and it maintains the same board setup as traditional Chess [9]. The primary distinction lies in the objective, where players aim to lose all their pieces instead of capturing their opponents' pieces. The rules of Antichess considered in this project are those used by the Chess website `Lichess`, where the standard rules and piece movements of regular Chess as set by FIDE apply [10], with the following deviations [11]:

1. The objective of the game is to have no legal moves available, either by losing all pieces or by being stalemated.

2. Kings lose their royal powers, meaning that the following changes occur: there are no checks, kings may be captured, a pawn may promote to a king, and there is no castling.

3. A player may not make a non-capturing move when there is a capture available. This is termed *forced capture*.

Forced capture often triggers a chain reaction, leading to subsequent captures and resulting in extended sequences of captures. This can make it challenging for players to assess whether a position is advantageous. For instance, consider the scenario depicted in Figure 2.1, where correct play by Black forces the white player, despite

having only a single rook, to capture all of Black's pieces. These prolonged chains of captures significantly reduce the game tree compared to traditional Chess, resulting in a smaller number of total nodes within the game tree. Antichess is also a much shorter game, often over in fewer moves than Chess. Chess averages 33.6 moves per game compared to the 24.8 of Antichess, based on `Lichess` games in December 2023 [12].

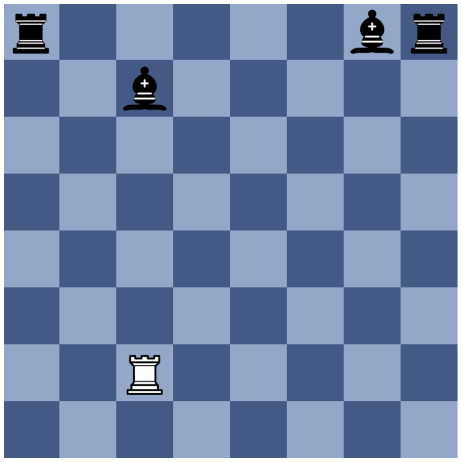
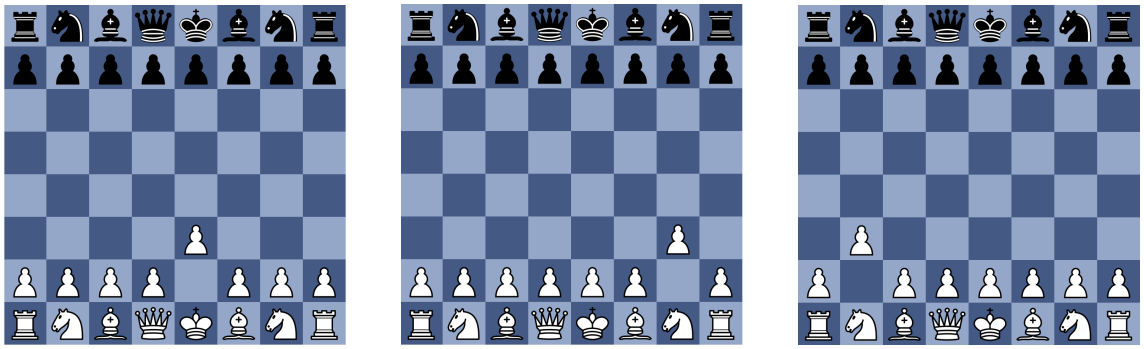


Figure 2.1: Antichess positions with white to move in a losing position, forced to capture the bishop. Pieces by Burnett, licensed under CC-BY-SA 3.0 [13].

Due to these factors, it was proven by Watkins in 2017 that if both players play perfectly, White is always going to win [14]. Since the result with perfect play is known, Antichess is a "solved" game. Within Antichess there are multiple openings which are solved, but the second and third most common openings are still unsolved [12, 15]. See figure 2.2 for the three most commonly played first moves for White. The fact that these unsolved openings are played shows that while solved it is still an interesting problem to evaluate arbitrary positions in Antichess.



(a) The most commonly played first move, which is solved.
(The Watkins' Opening)

(b) The second most commonly played first move, which is unsolved.
(Swedish Opening)

(c) The third most commonly played first move, which is unsolved.
(German Opening)

Figure 2.2: The most commonly played opening moves in Antichess for White, from left to right. Pieces by Burnett, licensed under CC-BY-SA 3.0 [13].

2.2 Chess Rating

Chess utilises a rating system called Elo to represent the skill levels of its players [16]. There are several ways to calculate the Elo, with one presented in equations 2.1 and 2.2 based on the game's expected outcome.

$$expected\_outcome = \frac{10^{Elo_t/400}}{10^{Elo_t/400} + 10^{Opponent\_Elo_t/400}} \tag{2.1}$$

$$Elo_{t+1} = Elo_t + k \times (outcome - expected\_outcome) \tag{2.2}$$

Equation 2.1 calculates the expected outcome from the player's perspective, used in calculating the player's new rating after the match Elo_{t+1} in equation 2.2. Herein k is a scaling factor representing the maximum winnable Elo from a single game.

2.3 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a probabilistic decision-making algorithm for decision trees. As a Monte Carlo method, MCTS uses repeated sampling to predict the optimal action from the tree [17]. Each node stores several pieces of information to guide the search process. One of these is the number of visits to the node itself. Another is the estimated value of the decision, used to provide insight into the desirability of actions leading from that node. Both of these metrics are used to help balance the exploration between desirable nodes, and under-explored nodes.

The MCTS algorithm consists of four main parts: selection, expansion, simulation, and backpropagation [17, 18]. This process repeats until the computational budget or a sufficient level of confidence is reached; at that point, a decision is selected.

The selection stage begins with looking at all the possible moves from a root state s_0 [17]. Subsequently, the next state s_1 is determined by a selection policy, which employs a strategy to choose the next child node from a state. This policy guides the decision-making process by balancing between exploration and exploitation. With a smart selection policy, promising moves are explored more thoroughly. The selection is repeated $s_0 \Rightarrow s_1 \Rightarrow ... \Rightarrow s_{n-1} \Rightarrow s_n$, where s_n is an unexplored node. The novel state s_n is then appended to the tree, see figure 2.3.

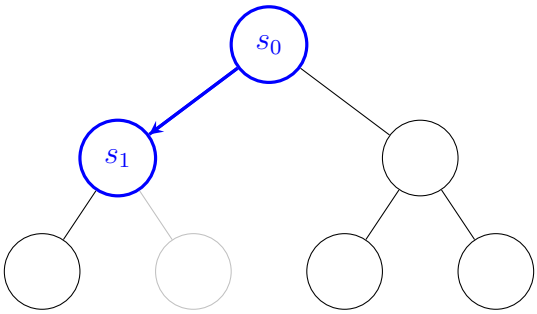


Figure 2.3: Selection of the node s_1 from the root node s_0 according to the priority of s_0 children.

After being selected all child nodes of s_n are added to the tree, and each node is given an initial excitement for selection [17]. A simulated game with random moves is initiated from s_n until some terminal state, where the weighting of moves greatly affects the level of play [19], this can be seen in figure 2.4.

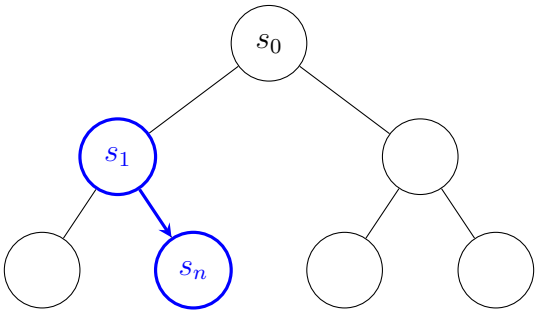


Figure 2.4: Expansion of the tree from the node s_1; exploring future moves before making one.

The simulation result is used to give a better value for the node; the child node's value affects the parent nodes by being propagated up the tree updating $s_n \Rightarrow s_{n-1} \Rightarrow ...s_1 \Rightarrow s_0$, seen in figure 2.5 [17]. The visit count and simulation results are used in adjusting the search priority of a node.

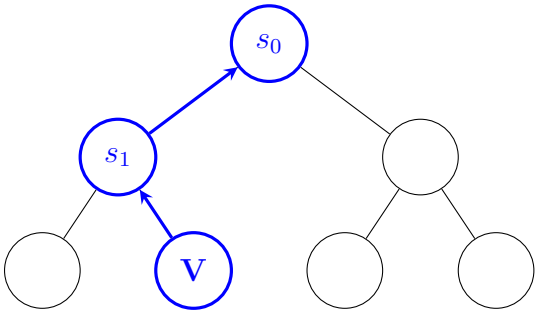


Figure 2.5: The simulation value is propagated up the tree, updating its parents.

These four steps: selection, expansion, simulation, and backpropagation, are repeated for a set number of iterations or within a certain time limit [17]. Finally, MCTS selects the most promising action to make from the tree. One method for choosing an action is to select the child of the root node with the highest value and the highest visit count.

2.4 Neural Network

Neural networks, a form of machine learning program, emulate human decision-making by employing mathematical models inspired by biological processes [20]. In the pursuit of mimicking biological decision-making, a neural network is built upon nodes, called neurons, and edges between them, called weights. These neurons are organised in layers that can detect features from inputs and produce a desired output.

The architecture of a neural network consists of an input layer, any number of hidden layers, and an output layer [20]. The input layer is responsible for converting the input fed to the network to a desired shape. Hidden layers allow for more complex reasoning at the cost of more computational power. Therefore, the number of hidden layers varies greatly depending on the network's use case and the complexity of the problem. Finally, the output layer combines the network's findings into a desired output. The generalised architecture can be seen in figure 2.6, where neurons in the input layer are marked as red, neurons in hidden layers are marked in green, and the output neurons are blue.

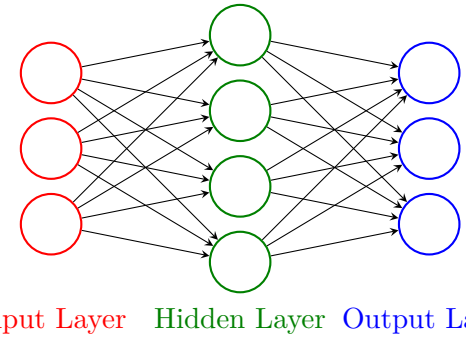


Figure 2.6: A visualisation of a neural network with an input layer in red, a hidden layer in green, and an output layer in blue.

To determine the importance of each neuron in the network, a specific value is associated with it. This is dependent on the values of neurons connected to it and their respective weights [20]. These characteristics define a feedforward network, as each neuron affects the ones connected to it in the next layer. A neuron's value y is given by equation 2.3, where x_i is the value of the i:th connected neuron, w_i is the connected weight between them, b is a bias and n is the number of connections to the neuron. The function f is the neuron's activation function, creating the conditions for when the neuron should pass its output to the next layer [20], presented further in section 2.4.1.

$$y = f(\sum_{i=1}^{n} w_i * x_i + b) \tag{2.3}$$

When feeding an input to the network, each neuron in the input layer gets activated with a value dependent on the network's input. These in turn trigger neurons in the next layer until the output layer is activated [20], leading to the network producing an output for a given input.

2.4.1 Activation Functions

The output of a neuron is given by the conditions set for the activation function [21]. Therefore, activation functions are vital in improving the network's ability to mathematically approximate the general trend of a dataset. There are multiple activation functions, each adapting the neuron's value in a specific way, with examples being the rectified linear unit (ReLU) and the Sigmoid activation function. This allows the network to learn complex patterns instead of relying on linear relationships, which are called linearities [22]. Avoiding linearities is important when approximating trends on non-linear data.

The ReLU activation function is one of the most popular activations for deep-layered architectures [23]. The activation limits the range of the output by changing negative values to zero, presented in equation 2.4.

$$ReLU(x) = max(0, x) \tag{2.4}$$

The Sigmoid function limits the range of the output from $(-\infty, \infty)$ to $(0, 1)$ [22]; this is presented in equation 2.5.

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

2.4.2 Training a Neural Network

Training a neural network is what improves its performance, through fine-tuning the network's output [24]. During initialisation, the network's parameters: weights and biases, are set according to an initialisation function [25]. Throughout training, weights and biases are continuously tuned to make the network learn what output should correspond to a given input. To achieve this, the network uses a loss function and backpropagation to adjust the network's parameters and decrease the error [20].

The purpose of the loss function is to compute the difference between the expected outcome and the actual outcome. Examples of loss functions are mean-squared error (MSE) and cross-entropy loss. The MSE E between the prediction x_i and the actual value z_i for a set of n predictions is given by equation 2.6.

$$E(x, z) = \frac{1}{n} \sum_{i=1}^{n} (x_i - z_i)^2 \tag{2.6}$$

Cross-entropy loss is a commonly used loss function when training deep neural networks [26]. In equation 2.7 the loss H is calculated between the predicted probability distribution q and the true distribution p.

$$H(p, q) = -\sum_{i=1}^{n} (p_i \times log(q_i)) \tag{2.7}$$

Loss functions are used in backpropagation, today's most common way of training a neural network with several hidden layers [27]. It utilises the network's loss in order to decide how the network should be updated to increase its performance. This can be seen in figure 2.7, where the error is used to update the parameters.

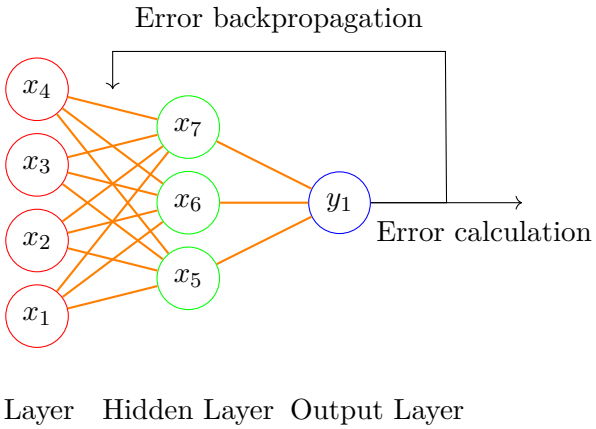


Figure 2.7: Network architecture showing the error propagating backwards top update weights and biases.

The algorithm works by propagating the error backwards through the network where it, for each neuron, computes a gradient based on its connected neurons [27]. The gradient then represents the direction and magnitude in which weights and biases are adjusted to reduce the total error of the network. Equation 2.8 highlights the formula for calculating a gradient. Herein, δ_n is the previous layer's error signal, originating from the output layer as $outcome - predicted$. Throughout the network, error signals of inner layers are computed via the chain rule of the previous layers' error signals. Then, $w_{n,j}$ is the weight between the n:th neuron in the previous layer and the current layer's neuron j. Finally, y_j is the neuron's value, as per equation 2.3. Therefore, the error propagates backwards towards the input layer, updating all parameters in the network.

$$g_t = \frac{\partial E}{\partial w_{i,j}} = y_j \times (\sum_n \delta_n \times w_{n,j}) \times y'_j \tag{2.8}$$

A common technique to speed up and improve the training of deep neural networks is to apply batch normalisation [28]. This normalises the information passed from the previous layer's activation function to a mean value of zero and standard deviation of 1, $X \sim \mathcal{N}(0, 1)$. This reduces training time, reduces the model's error and allows for greater learning rates of the optimiser.

2.4.3 Optimisers

Computing how much each weight and bias should change given their gradient is done by the network's optimiser. There are numerous optimisers, each changing the network's parameters their way according to the gradient. With this said, one of the most recognised optimisers is stochastic gradient descent (SGD).

SGD is a version of gradient descent preferable when working with large amounts of data due to its point sampling. Instead of calculating gradients for each data

sample, a random data point is used to calculate the gradient for all other data points [29]. Therefore, it provides efficiency, doing so however introduces noise to the system, as the gradients are only approximations of the exact ones.

The gradient descent algorithm works by iteratively taking steps of size Δx_t in the opposite direction of the gradient, as per equation 2.9. The distance of each step is specified by the optimiser's learning rate η [30] in the direction of the gradient g_t, presented in section 2.4.4.

$$\Delta x_t = -\eta g_t \tag{2.9}$$

Therefore, it gets closer to optimising the neural network's weights and biases. This can be seen in figure 2.8 where the parameters are updated each step to gradually converge the network's performance towards the global minimum.

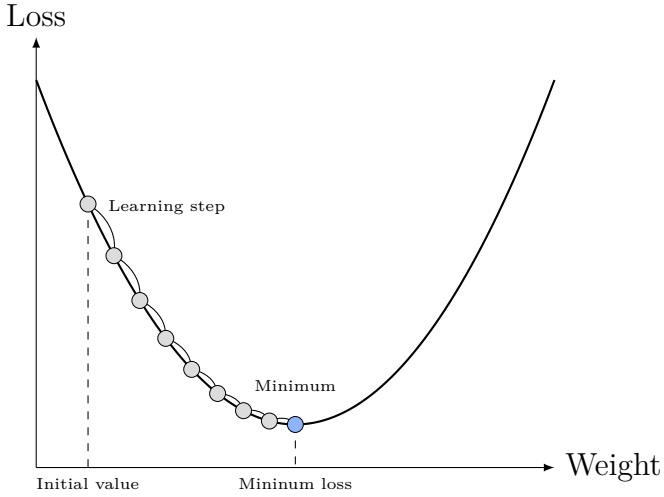


Figure 2.8: The figure shows incremental steps to minimise the loss and converge towards a minimum.

Another commonly used optimiser is Adam; an adaptive estimation algorithm [31]. It has several advantages, mainly that it on average converges faster than SGD. This is a result of different parameters to Adam calculating individual learning rates from the first and second moments of the gradients. This means it incorporates older gradient calculations to increase the performance. However, studies have shown that the Adam algorithm can lead to a bad generalisation for deep neural networks [32]. For insight into the mathematics of Adam, see: *Adam: A Method for Stochastic Optimization*, D. P.Kingma & J. Lei Ba, 2015 [31].

2.4.4 Hyperparameters

Hyperparameters are parameters that, together with the optimiser, alter the training process of the network to increase the network's performance [33]. There are several

hyperparameters including the number of epochs, batch size, and the optimiser's learning rate.

Epochs are used when training a network, specifying how many times the entirety of the dataset is trained on. 10 epochs result in each sample of the dataset being used 10 times to update the model's parameters throughout the training process [34].

Batches divide the total dataset into smaller datasets within an epoch. For each batch, the calculation of the network's error and change in parameters are updated after all samples in a batch have been trained on, instead of being updated after each sample [34]. Therefore, the number of times the network has to update its parameters decreases, resulting in reduced training time [35].

The learning rate specifies how much the parameters should be updated by the optimiser for each time step. The range for the learning rate is [0,1] [36]. Specifying a learning rate close to 1 increases the length of the network's steps per equation 2.9. However, a high learning rate risks the network overstepping the global minimum, leading to a never-improving neural network. This can be seen in figure 2.9, where the network's steps are too large for the parameters to converge on the global minimum.

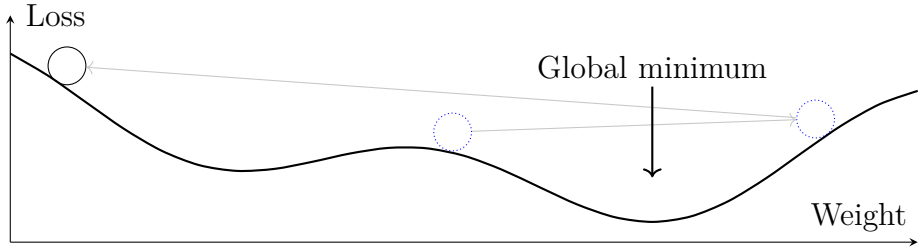


Figure 2.9: Visualises an example of a learning rate too high, causing the network to skip over the global minimum.

At the same time, a low learning rate can result in the network mistaking a local minimum for the global minimum and therefore not converging towards optimal weights and biases, seen in figure 2.10. A low learning rate could however converge towards the global optimum, but doing so would take a long time due to the small steps size [30].

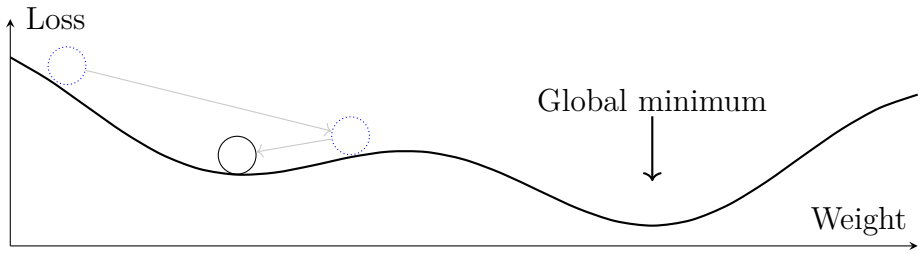
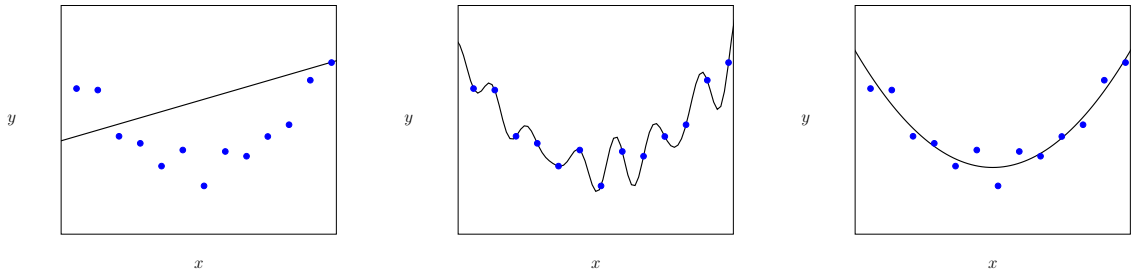


Figure 2.10: Visualises an example of the network finding a local minimum instead of a global minimum as a result of a learning rate that is too small.

2.4.5 Underfitting and Overfitting

When building and training a neural network a few common errors need to be avoided. Underfitting is an error where the neural network is incapable of finding trends while training on a portion of the entire dataset, referred to as the training dataset [37]. Additionally, this results in the network performing poorly on the test dataset, containing data that the network has not seen before. There are numerous reasons for underfitting such as the network being too simple in its design, too limited in the amount of training data, or not being trained enough.

Overfitting is an error on the opposite end. The neural network finds too much information in the training datasets which ends up hurting testing accuracy [38]. Often this error occurs due to training dataset containing outliers which do not need to be rationalised into the neural network's logic. This type of error can occur as a result of an overly complex network or training the network for too long. An overfitted network is characterised by high accuracy on the training dataset but fails to generalise the knowledge towards the testing dataset. This can be safeguarded against by having a large, general and varied training dataset. A visualisation of underfitting, overfitting, and perfect fitting can be seen in figure 2.11.



(a) Example of underfitting.

(b) Example of overfitting.

(c) Example of perfect fitting.

Figure 2.11: Visualisation of a common neural network challenge where the network underfits, overfits, and fits perfectly on the same dataset.

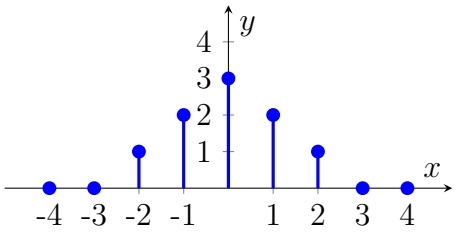
2.5 Convolutional Neural Network

A convolutional neural network (CNN) is a deep learning feedforward neural network [39]. The network consists of multiple layers, using convolutional layers to extract and transform features of the input into a useful output.

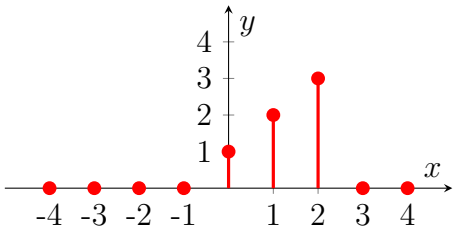
2.5.1 Convolution in Mathematics

Discrete convolution is a mathematical operation on two functions. It is used in signal processing, image processing, and when multiplying two polynomials together [40]. Discrete time convolution can be described as mirroring one array around the y-axis and then sliding it over the other array, one step at a time [41].

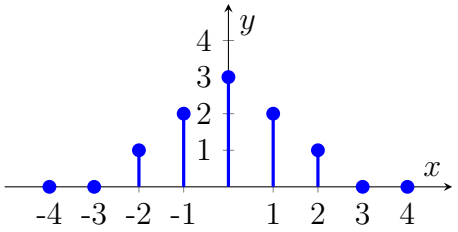
An example of a discrete convolution can be seen in figure 2.12. At each step, every overlapping value is multiplied and summed up. The result is the convolution value at that step. Since the two functions can be described with two arrays in one dimension, the operation is called one-dimensional convolution. The formula for calculating the one-dimensional discrete convolution $A * B$ of two discrete functions $A\colon \mathbb{Z} \to \mathbb{N}$ and $B\colon \mathbb{Z} \to \mathbb{N}$ can be seen in equation 2.10.



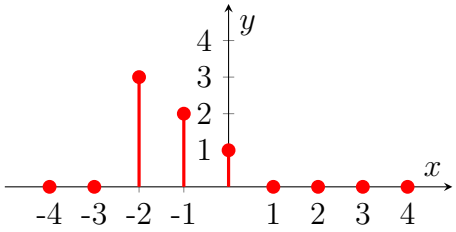
(a) The discrete function $A\colon \mathbb{Z} \to \mathbb{N}$.



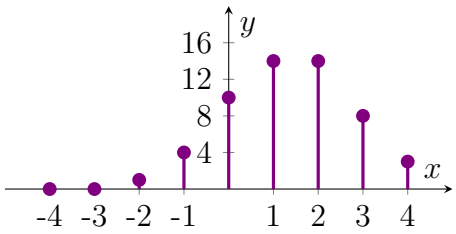
(b) The discrete function $B\colon \mathbb{Z} \to \mathbb{N}$.



(c) Function A is not mirrored before combined with *d*.



(d) Function B mirrored on the y-axis before combined with *c*.



(e) Result of convolution between figure *c* and *d*.

Figure 2.12: Example of discrete convolution between functions A and B. Function B is mirrored on the y-axis and then slid over function A, with the result shown in *e*.

$$A[k] * B[k] = \sum_{m=-\infty}^{\infty} A[m]B[m-k] \tag{2.10}$$

2.5.2 Two-dimensional Convolution

To recognise patterns and extract features in two-dimensional objects a convolution matrix, also known as a kernel, is used instead of a single array [42]. The number of features in an input is represented by the depth, often referred to as the number

of channels [43]. For example, for an RGB image, the number of channels is three as it consists of three-coloured features. Depending on the shape and weight of the kernel, it can extract different features. For example, a common way to implement a kernel used for edge detection can be seen in equation 2.11.

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & +8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{2.11}$$

An example of using the outline kernel from equation 2.11 can be seen in figure 2.13. Since the kernel sums up to 0, it results in a black-and-white image that showcases sharp changes in brightness, important for edge detection. This simplifies the image, removing irrelevant information while maintaining any important information.

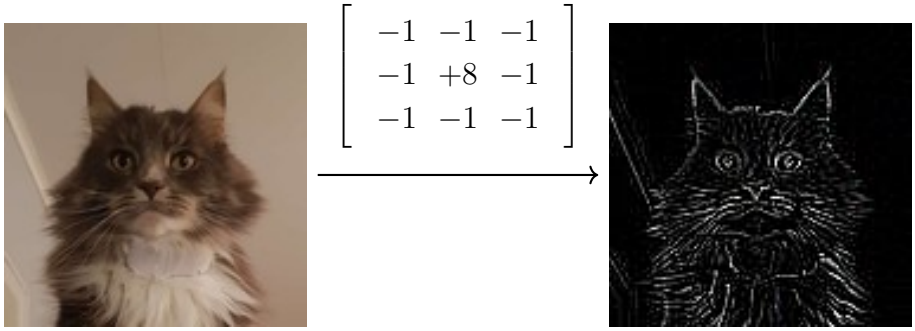


Figure 2.13: Example of a convolution kernel applied on a 128×128 image of a cat, visualising edge detection done by the convolution.

In two dimensions the kernel is not only able to slide across the pixel map but also able to use larger strides along it. The stride can be configured for specific tasks, moving in both x and y-axis as desired. [43]

Performing convolution on an input shrinks its size for each convolutional pass [43]. To keep the same shape in the output as in the input, padding is necessary. Padding refers to the practice of adding extra elements to the pixel map so that the convolution does not decrease the size. A common padding technique is to add zeros all around the boundary of the pixel map. This serves two purposes: to maintain the correct shape and to ensure that the added elements do not contribute to the result of the convolution.

2.5.3 Architectural Advantages

CNNs are used in many fields but are especially useful for images [44, 39]. Since images are typically a grid structure for all pixels, CNNs excel at image recognition and detection. The idea behind CNNs is based on visual perception.

A CNN offers several advantages over a fully connected network, primarily through a reduction in the total number of parameters [39]. The reduction is achieved through

weight sharing and using fewer connections between two layers. Weight sharing is when the convolutional kernel remains fixed throughout the stride across the input. This further reduces the number of parameters since, for all neurons, the weights are the same [45]. In a fully connected network, every neuron in the first layer is connected to each neuron in the second layer. However, in CNNs there are fewer connections between layers; each neuron in the second layer is only connected to the neurons in the first layer used in the convolution. This drastically reduces the number of parameters [39]. For example, consider a 6x6 image and a 3x3 kernel. The convolutional layer would use $3 \times 3 + 1 = 10$ parameters, while a fully connected layer would use $37 \times 36 = 1332$ parameters. Consequently, a CNN could encode many more features with the same number of parameters compared to a fully connected network [44].

2.6 Vanishing Gradient Problem

Vanishing gradients are a well-known problem when constructing deep neural networks using gradient optimisation [46]. Constructing competent neural networks benefits from using non-linear activation functions, as non-linearities allow the network to learn complex relationships between features [22]. However, these non-linear activation functions also cause the vanishing gradient problem.

Vanishing gradients occur when the gradient is backpropagated through the network's activation functions [22]. At each layer, originating from the output, the gradient is adjusted according to the derivative of the layer's activation. This results in the gradient gradually converging towards zero. Therefore, early layers in deep-layered architectures are prone to parameters not updating as a result of small gradients. Ultimately, this causes a degradation in the network's performance as its parameters do not converge towards a minimum loss.

An example of an activation function causing vanishing gradients is ReLU through its adjustment on negative weights to zero [46]. Therefore, the gradient also becomes zero, resulting in weights not updating properly. In turn, this leads to a network producing the same output, regardless of input.

To combat issues with vanishing gradients, multiple solutions have been proposed [46]. One solution is the construction of Residual neural networks to allow for free-flowing information throughout the network, introduced in the following section.

2.7 Residual Neural Networks

Constructing deep convolutional neural networks has proven to be an integral part of accurately classifying images [47]. Top competitors in the ImageNet competition, an online contest for neural network image recognition, all used networks with deep architectures of at least sixteen hidden layers. Constructing a network with more hidden layers, to increase its accuracy, has however proven to cause problems with

vanishing gradients. To avoid this problem, and still allow for a deep layered architecture, a residual neural network (ResNet) is built upon residual blocks with skip connections between them.

Utilising residual blocks to further increase the amounts of layers has proven effective as the ResNet architecture won several categories in the 2015 ImageNet competition [47]. To increase the number of layers without causing the gradient to vanish at backpropagation, each block contains a body of layers and a skip connection around them; that is, the input passed to each block is forwarded and added to its output without being changed by the block's body.

The skip connection allows the network to increase its information flow, eliminating the vanishing gradient problem [47]. As a result, each block in the network can increase the number of complicated detectable patterns whilst not yielding higher errors than a shallower model.

Figure 2.14 visualises the architecture of a single residual block. In the figure, x represents the skip connection and $f(x)$ is the residual function between the block's input and output [47], representing the difference between the input and output of a residual block.

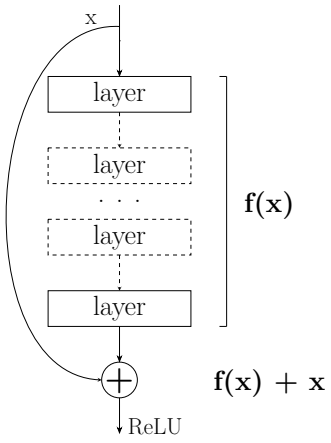


Figure 2.14: Visualisation of the architecture of a residual block in which the input x is forwarded around all layers to be added to the $f(x)$ representing the changes made by the layers.

2.8 Dense Neural Networks

Similar to ResNets, a Dense neural network (DenseNet) uses skip connections to allow for a greater flow of information through the network [48]. Therefore, DenseNets are preferable when constructing deep-layered architectures to learn complex patterns as they mitigate the vanishing gradient problem. At the same time, several other benefits of using DenseNets have also been acknowledged such as strengthening feature propagation between layers and reducing the network's parameter count by

feature reuse. Implementing DenseNets has proven effective as it has outperformed other regularly used architectures in image recognition tasks as a result of its dense blocks.

The dense block architecture differs from others through direct connections between each hidden layer in a block, allowing the network to increase its flow of information [48]. This is achieved by continuously concatenating the previous layers to the next one, therefore increasing the number of channels and features as the block progresses. The architecture of a single dense block can be seen in figure 2.15 where the input to a layer, H_n, is the combined output, x_n, of the block's previous layers.

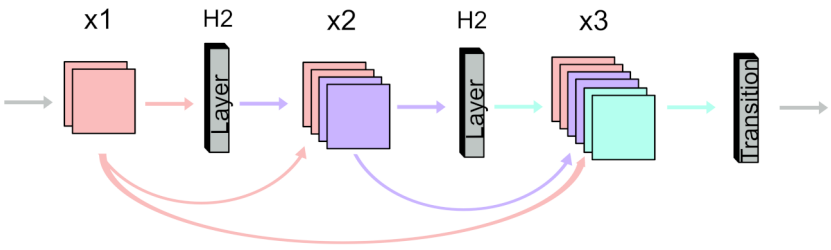


Figure 2.15: Visualisation of an implementation of a DenseNet, showing connections between each block and their skip connections.

As a result of concatenating each layer's output, dense blocks allow for greater feature reuse compared to ResNets [48]. This is due to the features learned at each layer being applicable and accessible for the upcoming layers. This phenomenon is the reason for the high parameter efficiency of DenseNets whilst still performing tasks accurately.

2.9 Combining Game Tree Search and Neural Networks to create Chess Engines

Due to the way that neural networks are designed to mimic human decision-making, they have become desirable to use when creating Chess engines. The reason for this is that the neural network can learn undiscovered strategies due to its conceptually limitless environment, constrained only by the rules of Chess itself. This can be compared to traditional engines that rely on thousands of rules and heuristics made by humans [7].

Neural network-based Chess engines like AlphaZero are implemented by using the network to guide a search algorithm in order to find promising moves [8]. AlphaZero specifically uses the MCTS algorithm as it is a probabilistic decision-making algorithm, which as described in section 2.3 tries to approximate the optimal move. This is useful when combined with the network since there is no need to do exhaustive searches to evaluate each position.

The neural network used in AlphaZero is structured to output a value that predicts the outcome of the game and a policy which is a probability distribution that shows how promising every possible move is [8]. This is done by creating a main "body" in the first part of the hidden layer architecture. The body then splits up into two different "heads", the policy head and the value head, where each head has a different architecture. An overview of this can be seen in figure 2.16. When the network is initialised the predictions will be completely random. The way that AlphaZero improves the network is by generating self-play games with the method described in the previous paragraph. The network is then trained on the values from the MCTS algorithm, i.e. the actual win percentage in a given position and the calculated probability of selecting a move. This training results in a network that predicts the values of an expanded search tree. AlphaZero trained the network for 700,000 steps, where each step was a batch of 4096 Chess positions; in total this equated to about 44 million Chess games. The training took approximately 9 hours using 5,000 first-generation tensor processing units (TPU) when generating the games with MCTS and 64 second-generation TPUs to train the network.

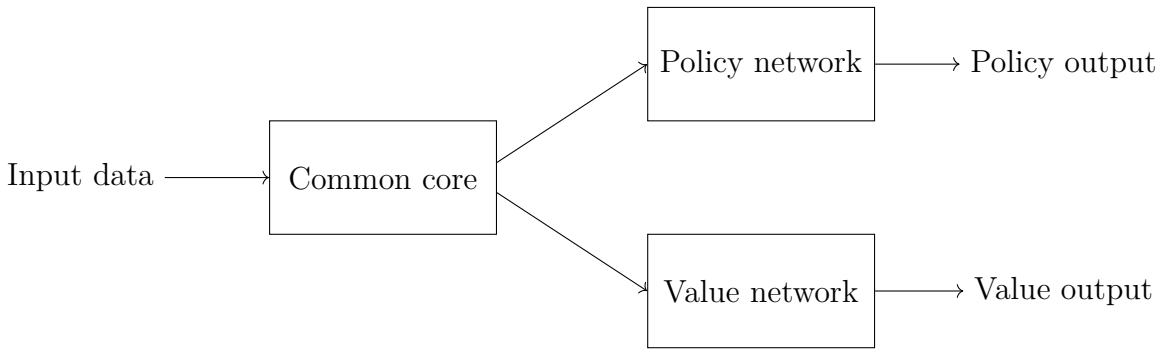


Figure 2.16: Example of how AlphaZero split the network into two parts to output a value and policy vector for each input [8].

3

Design of the Neural Network Architecture

This chapter presents the design choices made for the construction of the neural network. Architectural implementations are described throughout sections 3.1–3.4. Further, methods of training the network and tuning its parameters to improve performance are presented in section 3.5 along with methods of evaluating the overall performance in section 3.6.

3.1 Input to the Neural Network

Considering the focus on Antichess there was a need to plan how the neural network receives the chessboard as input. The main problem with this was to decide how to shape the input dimensions, i.e. how many dimensions are needed to represent the board and the pieces on it. There was also a need to differentiate the colour and type of the pieces, as this would help the network create a better intuition depending on the position of the pieces. The final decision after research on the shape of the input was to use the same shape as AlphaZero, since it seemed logical and has worked in the context of chess.

The input layer is therefore designed to accept inputs of the shape: $12 \times 8 \times 8$ oriented from the current player's perspective. The network expects the input to contain 12 channels of eight-by-eight board states with each board state representing a black-and-white image of a unique combination of the six types of pieces and the two colours. This can be seen in figure 3.1 where black squares are occupied and white squares are empty. The reason for representing the board as a black-and-white image was its utilisation in the effective AlphaZero [8].

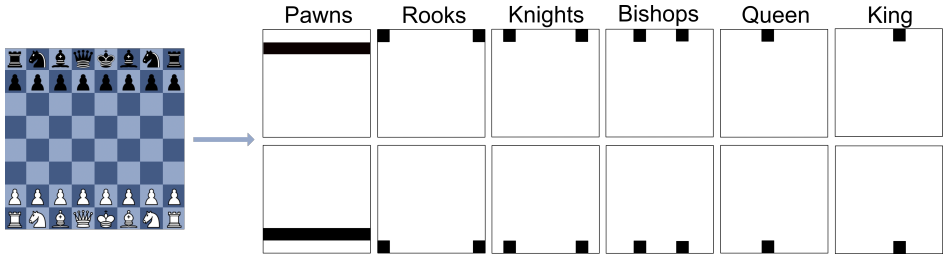


Figure 3.1: Visualisation of how the board is represented as 12 unique images in the input, with black representing occupied squares and white representing empty squares.

3.2 Shared Value and Policy Network

After an input of the dimensions $12 \times 8 \times 8$ is passed to the network, a shared architecture for the value and policy output is responsible for identifying initial patterns. The reason for a shared architecture is to improve the efficiency of the network and subsequently allow for a deeper neural network to be constructed. Without a joint architecture, both the value and policy would need to have their own deep architecture for pattern recognition. Having two separate architectures would increase the time of both training and self-play as pattern recognition has to be done twice. Instead, with the networks being designed to simultaneously find patterns used to output a value and policy, time can be spent on training a deeper architecture leading to a more capable network.

The plan for the shared architecture was initially to build a residual neural network as used in AlphaZero's implementation. However, further research led to a dense neural network being considered as well, because of the shown potential in image recognition tasks [48]. The final decision was that both architectures would be built from scratch in order to find the one that suited the project the best. Building both architectures from scratch was hypothesised to increase the flexibility of the networks, allowing for parameters to be tuned more easily. Assumptions were that this would ultimately increase the number of architectural comparisons, leading to a better-performing network in the end.

With the input being comparatively similar to an image, the core architecture and combination of layers took inspiration from, and utilised common foundations of, convolutional neural networks presented in section 2.5. Therefore, regardless of the type of network being implemented, they all achieve their pattern recognition capability through the use of convolutional layers.

At first, input to the network is passed through a convolutional layer. The convolution applies a 3×3 kernel with stride of 1 and padding to preserve the shape of the input whilst detecting features. The number of output channels from the convolutional layer varies depending on the network architecture: for ResNets the convolution outputs 256 channels in accordance with AlphaZero's implementation,

and for a DenseNet this is instead set to 32 output channels to limit the number of concatenated features in a dense block. After the convolutional layer, batch normalisation is applied before passing through ReLU.

After the input has been passed through convolution, batch normalisation, and ReLU, it is forwarded to either residual or dense blocks depending on the network architecture.

3.2.1 Implementation of Residual Neural Network

The residual neural network is implemented as a part of the shared value and policy architecture to further pattern recognition by passing the input through a predefined number of residual blocks.

Each residual block is built with convolution, batch normalisation, and ReLU layers. When the block receives an input-tensor it modifies a copy of the tensor so that the original tensor can be used as the block's identity. The copied tensor is modified throughout the residual function by being forwarded through a convolutional layer followed by batch normalisation and ReLU. Afterwards, it passes through another convolutional layer followed by batch normalisation. Each convolutional layer applies a 3×3 kernel along with stride of 1, padding to preserve shape, and 256 output channels in accordance with AlphaZero's implementation.

After passing through convolutions and retaining its shape the output of the residual function is added to the stored identity via element-wise addition of the two matrices. To ensure that no linearities are present afterwards, the output is forwarded to another ReLU as per the ResNet implementation of *He et al.* [47]. The output from the final ReLU in each block is then used as the input to the next residual block. A visual representation of the implemented ResNet architecture can be seen in figure 3.2.

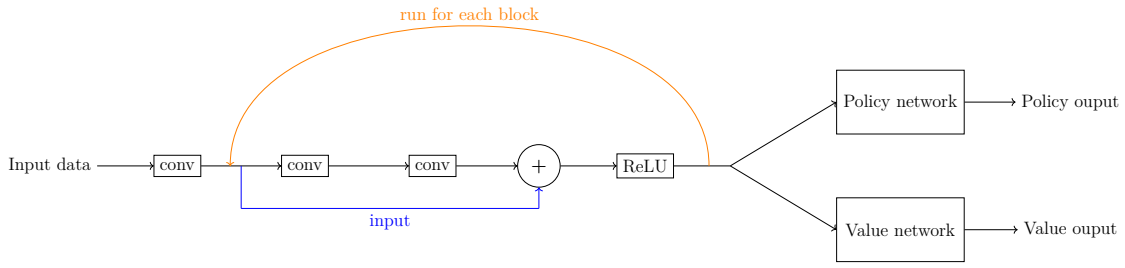


Figure 3.2: Visualisation of the implemented ResNet architecture presented in section 2.7. Each layer presented as *conv* represents the combination of a 2d convolutional layer, a batch-normalisation layer, and a ReLU layer.

3.2.2 Implementation of Dense Neural Network

Similar to the ResNet, the DenseNet is implemented as the main part of pattern recognition for the shared architecture of both the value and policy networks. This allows for a deeper and more complex recognition of patterns without suffering from vanishing gradients. Additionally, due to its high parameter efficiency, implementing a DenseNet was hypothesised to decrease the time of training whilst still performing equally accurately as a ResNet [48].

The implemented DenseNet consists of four blocks, each with four smaller sets of layers, along with a transition layer between each block per Huang and Liu's implementation [48]. Each of the sets in a block consists of two batch normalisations, two ReLU, and two convolutional layers. The first convolution applies a 1×1 kernel together with stride of 1, padding to preserve shape, and an output scaled k times the number of input channels. Huang and Liu proposed a scaling factor of $k = 32$ between the first and second convolutional layer [48], however due to computational limitations the implementation used a scaling factor of $k = 4$. During experiments, using a scaling factor of $k = 32$ resulted in a tenfold increase in training time when training on 1704 board positions. The second convolutional layer uses a kernel of size 3×3, stride of 1, padding to preserve shape, and 32 output channels. After passing through a set of layers, the output is concatenated with the output of previous sets and used as input to the next set of layers. When the input to the block has passed through the block's four sets of layers, it is forwarded to a transition layer.

The transition layer between each block was implemented as a single convolutional layer with: kernel 3×3, stride 1, padding same, 160 input channels, and 32 output channels. Additionally, batch normalisation and ReLU are added. After these layers, the output is forwarded to the next block as opposed to Huang and Liu's implementation in which the output reduces dimensions for each block by passing through an additional layer [48]. The reason for this is the smaller dimensions of a chess board. The implemented DenseNet architecture is visualised in figure 3.3.

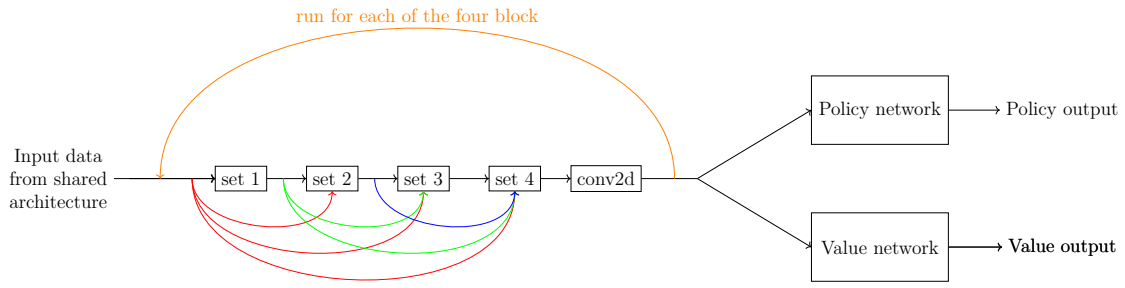


Figure 3.3: Visualisation of the implemented DenseNet architecture presented in section 2.8. Each layer presented as *set* represents the combination of two blocks consisting of a batch normalisation, ReLU, and 2d convolution layer.

3.3 Output from the Neural Network

The next step was to plan what the network should output. Our research led to the decision that the network should output a policy vector and a value output. This is formulated in equation 3.1 where the probability of winning, v, and the priority for each move, $\mathbf{p}$, is returned from the network f given a board state s [8].

$$(v, \mathbf{p}) = f(s) \tag{3.1}$$

The policy vector contains a distribution of 4216 move priorities used to guide the MCTS in Section 2.3, where a position in the vector can be decoded as a move in Antichess. Equation 3.2 shows the total amount of moves, accounting for every pair of squares being a possibly legal move and the promotions; this overcounts the possible moves.

$$64^2 + 5 \times 3 \times 8 = 4216 \tag{3.2}$$

The policy vector is encoded to represent moves from the current player's perspective instead of both player's perspectives. The reason for this was that it would shorten the vector as promotions only had to be represented in one direction. The value output shows the probability of winning in the current position. Values from the output is interpreted as following: values close to 0 would suggest that the position is a losing one, values close to 0.5 would be a drawing position and lastly values close to 1 would indicate a winning position.

3.4 Value and Policy Networks

Because the network was planned to have two different outputs, there was a need to split up the network into two heads, where one head was responsible for predicting the value and the other head the policy. Both heads are constructed as two independent convolutional neural networks. Transformation of the data is what distinguishes the two networks from each other. The value network transforms the data into a single float value, while the policy network transforms the data into a vector.

3.4.1 Value Network

The design of the value network is to specialise and convert the transformed input from the joint architecture into a single output representing the probability of winning at a specific board state. To achieve this, the network utilises a combination of convolutional, normalization, and activation layers.

The first layer is a two-dimensional convolution with a kernel size of 1×1, which

reduces the 12 channels down to one. Following the convolutional layer, batch normalisation is applied to stabilise and accelerate the training process. After that layer, ReLU activation is used to introduce non-linearity into the network, allowing it to capture complex relationships within the data. Subsequently, the output is flattened from a two-dimensional matrix into a one-dimensional array. This step is essential for feeding it into the last layer, which is a linear layer which outputs a single value: the evaluated probability of winning at the current board state. Figure 3.4 provides an overview of the policy network.

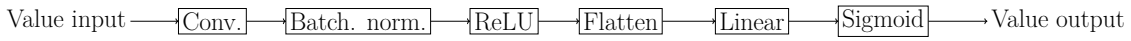


Figure 3.4: Visual representation of each layer the input to the value network is passed through before producing an output.

3.4.2 Policy Network

The policy network was created to output a distribution of values ranging from 0 to 1, which represent how promising each move is. Similar to the value network, it uses a convolutional layer followed by batch normalisation and ReLU layers. The convolutional layer applies a two-dimensional convolution with a kernel size of 3×3 and stride of 1. This layer reduces the channels from 256 down to 66.

Each channel consists of a 8×8 matrix where each slot represents the value of how promising a move is. With 66 channels there is a total of $66 \times 8 \times 8 = 4224$ possible moves to be encoded. The batch normalisation and ReLU are applied to speed up training and allow learning of complex non-linear relationships in the data respectively. Output from the ReLU must be further manipulated to represent all possible moves. The matrix is flattened into a single array of length 4224. However, since there are only 4216 moves, the array is shortened to include only the first 4216 elements, representing all possible moves a player can make. An overview of the policy network can be seen in figure 3.5.

Figure 3.5: Visual representation of each layer the input to the policy network is passed through before producing an output vector.

3.5 Training

The plan for the training process was to create a training function and a training loop that would enable the network to continuously create new data to train on. This process would provide the network with a large quantity of data in order to minimise the loss of the predictions.

3.5.1 Loss Function

Creating the training function required decisions on the implementation of loss calculation. It was decided that the most logical was to use the same loss function as AlphaZero, since it proved effective in training a neural network on Chess.

Mean squared error (MSE), introduced in section 2.4.2, is used to calculate the value loss, and cross-entropy loss is used to calculate the policy loss. MSE is a great loss function for the value output due to the loss representing the distance between the actual value and the network's prediction, as described in equation 2.6. Cross-entropy loss is suitable for the policy output as it calculates the difference between two sets of probabilities and returns it as the loss, as presented in section 2.4.2.

Due to the large difference in policy and value loss seen in figure 5.3, the losses are weighted with respect to each other. The reason for weighting the losses is to prohibit the network from prioritising improvements of the policy output over the value. If the losses were added together the policy loss would account for the majority of the total loss. Therefore, the network's total loss is given by equation 3.3 where α is used to weight the losses.

$$total\_loss = \alpha \times policy\_loss + value\_loss \tag{3.3}$$

Through experiments, the value for α was set to 0.2 as it resulted in the network slightly prioritising the policy. This is preferable as the policy is crucial in move selection, whilst still leading to the reduced value loss seen in figure 5.3a.

3.5.2 Hyperparameters

An important part of improving performance was choosing the best-performing optimiser and its learning rate. Two optimisers were therefore studied: SGD and Adam. For each of the optimisers, a learning rate of 0.1 was used as a default. The default was chosen from a range of [0.5, 0.0001]. The reason for this range was that it is beneficial to start with a larger learning rate so that the network will be able to find the global minimum instead of getting stuck in a local minimum.

Another vital part concerning the learning rate is to consider the scenario where the global minimum is found, but the actual minimum is continuously overstepped because of a large learning rate. When this scenario occurs the optimal approach

is to decrease the learning rate. To decrease the learning rate during training, a learning rate scheduler was suggested. The scheduler would decrease the rate when a plateau was reached for an extended period of training. For this, the scheduler was implemented to decrease the learning rate by a factor of 10 if the total loss remained within a threshold of 0.02 for five consecutive epochs.

The last step was to decide the amount of epochs that the training function would loop through. This was a difficult element to decide on as every problem has different characteristics, which means that there cannot be a definite answer. A big reason why this is the case is because a low number of epochs can be perfect for one problem but lead to underfitting in another. Similarly, a large amount of epochs can be optimal but could lead to overfitting if the problem does not require it. Additionally, there is a risk of the network ceasing to learn i.e. the loss plateaus after a certain number of epochs. To mitigate this and avoid wasting time, an 'early stopper' was implemented. With all of the before mentioned in mind there was a need to do extensive testing to identify a reasonable amount of epochs to use. This was evaluated by testing different configurations of networks where the number of epochs would vary between 5 and 25. The networks would then be plotted on a graph showing the change of loss of the network to identify the different issues that could appear.

3.5.3 Validating Performance

The loss of the neural network is not the optimal indicator when it comes to judging the performance of the network. This is because the loss from the loss function exclusively shows the progress of the training and how well the network has adapted to the training data. If the data is well generalised and provides a wide range of information, the loss can be used to gauge the performance. However, this is seldom the case and therefore there is a need to validate the accuracy of the network.

The plan was to split up the training data into a training dataset and a testing dataset. In this project, the accuracy was shown by calculating the network's error using the loss function for predictions on the testing dataset; this is denoted as validation. The validation loss would help show if the performance was as expected in the scenario that the training loss reached a promising minimum. Our research suggested that the training dataset should be in the range of 60 – 80%, which therefore meant that the testing dataset optimally would be in the range of 20 – 40%. Thorough testing was required in order to decide which dataset size was best suited for this project. The decision would be made by analysing which dataset size led to the lowest loss together with the lowest validation loss; this while still maximising the training dataset size, since a large training dataset is the most desirable.

3.5.4 Training Loop

To train the neural network for an extended period of time, there was a need to create a training loop where data would be generated from the MCTS and then sent to the training function. The MCTS implementation would also use the network to select the moves, which is the heart of improving the network using self-play. This would be done through a command line interface that enables the communication between the MCTS algorithm and the network.

The idea was to await a batch of data from the MCTS and then use this data in the training function. The size of the data would vary depending on the hardware available and time at hand. This is what was defined as a generation, i.e., a full loop of generating data and training on the data. The vision was that the network would be instantiated with random weights, and because of that selecting moves randomly, and after a couple of generations showing an improved intuition when selecting moves. To be able to detect the improvement, change of loss and validation loss would be plotted on a graph. This would show the trends of the network, which could be used when analysing the results of the training. The desired result would be a downward trend where the loss and validation loss would be lower after each generation.

3.6 Comparing Network Architectures

The network has several configurable variables which change the architecture of the neural network. In pursuit of identifying the optimal architecture for the project, a comparative analysis of various network architectures was conducted. The methodology involved was a systematic alteration of variables, such as learning rate, amount of residual blocks, number of epochs before stopping, search iterations, and the number of games to be used in each generation. To ensure a rigorous examination, isolated variables were modified individually while keeping other parameters constant. This approach facilitated the observation of the relative effectiveness of different configurations in terms of time efficiency.

Ultimately, four architectures were tested: ResNets with 4, 8, and 16 residual blocks along with a DenseNet of 4 blocks with 4 sets in each. Each network was trained on 14 000 board positions, in batches of 256, generated by self-play. The networks trained for 15 epochs and 3 generations, therefore training 15 times on the same data before generating 14 000 new positions, a total of 3 times. To speed up self-play, a search iteration count of 64 was used. This resulted in the network evaluating 64 future positions to determine the best long-term move before making one.

4

Design of game tree search

In this chapter, design decisions are presented for the implementation of a game tree search that searches for promising moves to make in Antichess. The equation for how the game tree search prioritises moves is presented in section 4.1 along with the method of implementing Antichess in section 4.2.

4.1 Implementation of Game Tree Search

For the game tree search, multiple implementations of search algorithms were considered as alternatives to the MCTS implemented in AlphaZero; one of which is Monte Carlo graph search (MCGS) instead of the MCTS. Implementing a MCGS would reduce the memory required as repeating positions in MCTS could be shared across the subtrees by using a directed graph [49]. Ultimately, it was decided that using AlphaZero's standard implementation was optimal to start with due to MCGS bringing more complexity to the project and at best only improving the engine's memory usage. As a result, the game tree search is implemented according to MCTS presented in section 2.3.

The MCTS implementation uses the values yielded from the policy network to assign an initial weight to each move. The search priority for expanded nodes is calculated using equation 4.1, where n_i is the number of visits, v_i is the average result from the simulation, and w_i is the expected result as yielded by the value network. However, positions with a proven result receive a $\pm\infty$ score and a search priority of 0.

$$\text{priority} = \frac{v_i + w_i}{2{n_i}^2} \tag{4.1}$$

The selection of the child node to be explored is limited by the rules of Antichess. Therefore, the Antichess implementation prohibits the MCTS from making illegal moves.

4.2 Implementation of Antichess

Both the training of the neural network and the MCTS require the implementation of Antichess so that it can learn which moves to make correctly. Utilising a pre-existing Antichess implementation was considered, such as the Chess library `python-Chess`. Doing so would save time for implementation but at the cost of less flexibility. It was decided that implementing Antichess was worthwhile, as doing so allowed for an API tailored to the project's needs.

Additionally, Antichess was implemented along with a custom graphical user interface (GUI), which can be seen in figure 4.1. This GUI serves as an intuitive interface for users to interact with the Antichess game, providing visual feedback when playtesting against the engine, however it is not used whilst training the network.

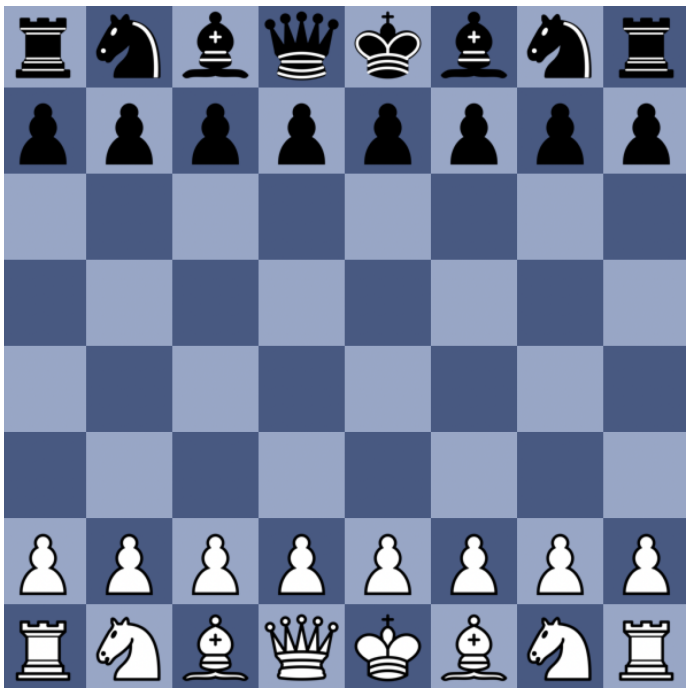


Figure 4.1: Shows a visualisation of the GUI that is used to play the game versus the engine. Pieces by Burnett, licensed under CC-BY-SA 3.0 [13].

5

Result

This chapter presents the final performance of the implemented engine in section 5.2 along with results from comparisons of neural networks described in section 3.6. Section 5.1 presents comparisons of optimiser and model architecture. All simulations and time comparisons are made using a computer of the specifications presented in table 5.1.

Table 5.1: CPU, GPU, and RAM specifications of the workstation used throughout training, self-play and comparisons.

Component	Model
CPU	AMD RYZEN 9 7950X
GPU	Nvidia RTX 4090
RAM	128 GB

The overall architecture of the final engine, evaluated in section, 5.2 is presented in figure 5.1. The figure highlights how the combination of an MCTS and Neural network is implemented, with the MCTS passing board states to the neural network in order to receive outputs guiding its search.

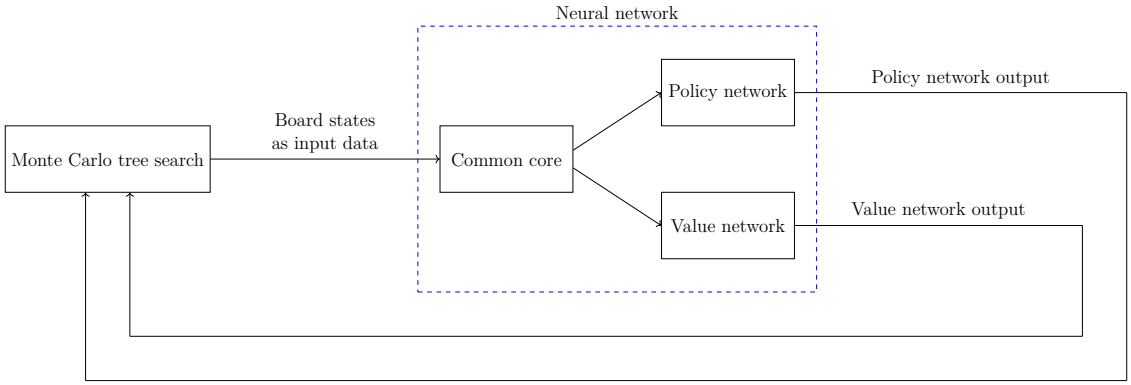


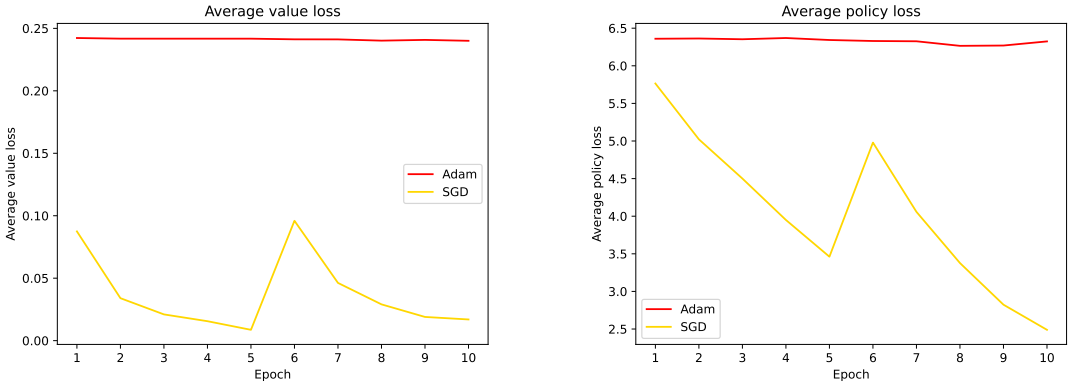
Figure 5.1: Overview of the full system where the Monte Carlo tree search sends board states to the network for evaluation. The network's output is then used in the Monte Carlo tree search to guide the exploration of nodes.

5.1 Evaluating Implementation Decisions

To achieve the best overall performance of the final engine, results from differences in the implementation of optimiser and network architecture are presented in the following sections.

5.1.1 Optimiser

Choosing the right optimiser is crucial for maximising model performance. A poor choice of optimiser can hinder training progress. After narrowing down our options to SGD and Adam, a simple experiment was conducted to determine which optimiser performed the best. The experiment was to train the model using the same parameters, once for SGD and once for Adam. The result of the experiment can be seen in figure 5.2 where it was clear that the Adam optimiser would not work at all. Therefore, the SGD optimiser is used in the neural network.



(a) Comparison between average epoch validation loss.

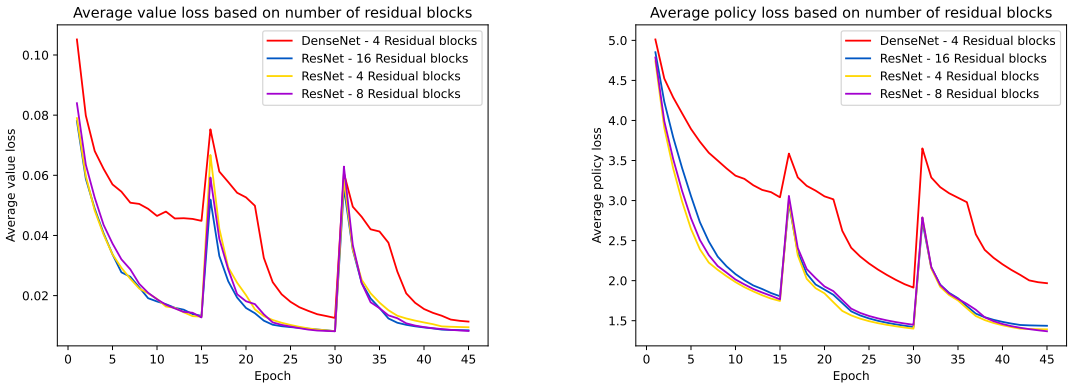
(b) Comparison between average epoch policy loss.

Figure 5.2: Comparison of optimisers: SGD in yellow and Adam in red on a Residual neural network with four residual blocks.

5.1.2 Network Architectures

Comparisons of several neural networks, with differences in architectural implementation, highlighted similarities in performance regardless of the model's complexity. All models achieved low loss for both the value and policy on the training set but failed to generalise all of the knowledge learned towards the validation set.

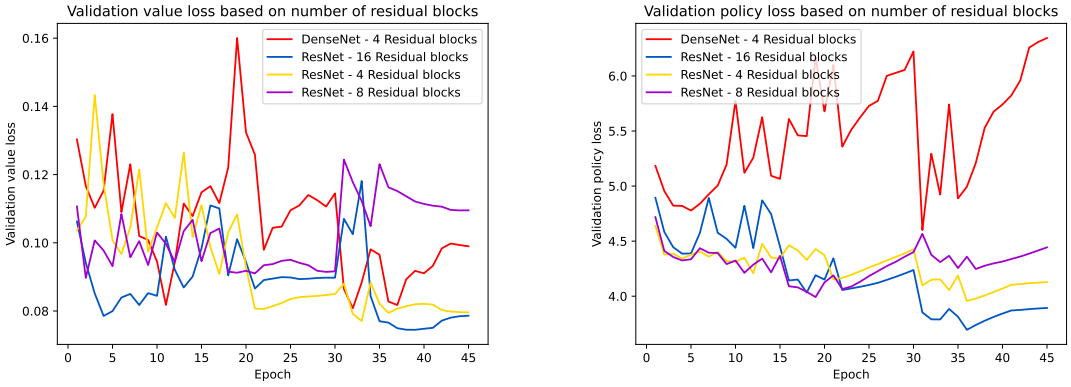
Figure 5.3 visualises each neural network continuously decreasing its value and policy loss throughout a generation of 15 epochs before new data is generated and trained on two more times.



(a) Training loss for the value output. **(b)** Training loss for the policy output.

Figure 5.3: Visualises the value and policy loss for each of the models. The loss is calculated from the training dataset.

Figure 5.4 visualises epoch losses for data that the network has never trained on. Therefore, it indicates the actual performance of the network, which for the majority of the network increases very slowly. Similar to the previous figure, new data is generated each 15th epoch.



(a) Validation loss for the value output. **(b)** Validation loss for the policy output.

Figure 5.4: Shows the validation loss for both the value and policy output for each of the four models. The loss is calculated from the testing dataset.

Due to differences in model complexity, the time it took to self-play and train the network for one generation varied greatly amongst the networks.

This is presented in table 5.2 where the name and number of blocks used are presented. Additionally, the time it took to self-play, with search iterations of $d = 64$ per move, and training of approximately 14 000 are presented together with the total time for one generation.

Table 5.2: Time difference for architectures, both for self-play and training

Model and number of blocks	Time of self-play	Time of training	Total time for one generation
ResNet 4	6 min & 50 sec	8 min & 24 sec	15 min & 14 sec
ResNet 8	11 min & 2 sec	13 min & 58 sec	25 min & 0 sec
ResNet 16	21 min & 13 sec	25 min & 33 sec	46 min & 46 sec
DenseNet 4	31 min & 34 sec	24 min & 39sec	56 min & 13 sec

With respect to the total time of self-play and training the networks together with the similarities in losses, the ResNet with 4 residual blocks was chosen as the network in which training should be continued.

5.2 Performance of the Engine

The final engine was trained for several generations, consisting of 14 000 positions each, with search iterations of 512 per move. Two versions of the neural network were used when evaluating the performance; one trained for 20 hours on approximately 196 000 board positions and the other trained for 9 hours on approximately 84 000 board positions.

To evaluate the engine, a round-robin tournament was ran with the engine at a few different iteration counts alongside a reference flat MCTS with the same settings. The engines will be ranked using their tournament score and Elo presented in section 2.2.

The result of a 672-game (192 per engine) tournament is presented in table 5.3. In the table, random denotes a player that makes moves randomly and flat refers to the sole use of probabilities from previous simulations of MCTS, without a neural network guiding its search. All the games for one double round-robin can be seen in appendix A.2, displayed as *Portable Game Notation*.

Table 5.3: The results of a 16 round double round-robin tournament between the different engine variants. The numbers in parenthesis show the iteration depth used in the MCTS.

Model	**Score**	**Wins**	**Draws**	**Losses**
ResNet 196k (2048)	146	144	4	44
Flat (2048)	138.5	135	7	50
ResNet 84k (2048)	137.5	136	3	53
ResNet 84k (512)	87.5	84	7	101
ResNet 196k (512)	82.5	79	7	106
Flat (512)	80	78	4	110
Random	0	0	0	192

The relative Elo of each player after the tournament is presented in table 5.4 with each player's starting Elo set to 800 before the tournament.

Table 5.4: Elo scores of the different engines, k=16, starting Elo=800. The numbers in parenthesis show the iteration depth used in the MCTS.

Model	**Elo**
ResNet 196k (2048)	1023
Flat (2048)	990
ResNet 84k (2048)	987
ResNet 84k (512)	771
Flat (512)	747
ResNet 196k (512)	737
Random	346

The best-performing engine, ResNet 196k, was benchmarked against Fairy-Stockfish approximating its overall skill at Antichess. It was able to beat level 2 but lost to level 3; see a game against level 2 in appendix A.1.

When compared to Flat, ResNet- 196k and 84k are much slower, running at roughly $\frac{1}{26.5} \approx 0.037$ iterations for each non-NN MCTS iteration. This is due to the slow runtime of running the network itself; the MCTS implementation is unchanged between the two.

6

Discussion

This chapter presents analysis and interpretations of the result. Section 6.1 discusses the neural network's effect on performance with regard to overfitting, data quality, model architectures, and training parameters. Furthermore, section 6.2 addresses limitations encountered throughout the project with future research opportunities presented in section 6.3. Finally, section 6.4 introduces the social and ethical implications of creating an Antichess engine.

6.1 Neural Network

Creating the neural network proved to be both challenging and time-consuming due to the complexity of the implementation. Even with reference points regarding implementation, such as the architecture of AlphaZero, there was a need to constantly evaluate different implementations with respect to the setup of layers.

6.1.1 Overfitting

Overfitting in a convolutional neural network is very common. From the results seen in figure 5.4 we speculated that our network has overfitted on the training data. Ideally, the validation loss should be continuously decreasing or plateau at a low loss. Instead, in our network, the validation loss is almost random as the graph is erratic and thus there is no clear decrease in the loss. The observation led to several efforts being made in order to mitigate overfitting.

We tried to decrease overfitting by penalising the network for large weights. To our surprise, it did not have any impact and did not reduce overfitting. Another way we tried to combat the issue was to increase the size of the dataset used in training. This helps the network generalise its findings instead of overfitting on the training data. But with a larger dataset, a new problem emerges: training time. Assessing how much additional training time is justifiable, relative to the performance gain it offers, is crucial. We did an experiment where we compared two models: one with a dataset of about 14 000 positions, and one with about 80 000 positions. To our disappointment, there was no improvement in the validation loss. Therefore, we decided that it was not worth it to increase the size of the dataset.

6.1.2 Data Quality

In our self-trained engine all data comes from the network playing games against itself. Each new training generation plays its own games, therefore generating unique data every time. This could result in large differences in the data quality between the generations. Additionally, this could affect the comparison of the performance of different models due to them having differences in data quality. Thus, it is not clear that the ResNet with four residual blocks was the most promising model, as it could have received better data quality.

6.1.3 Analysing Performance of the Dense Neural Network

Results from simulating multiple games with different architectures indicate that the DenseNet performs the worst, as opposed to Huang and Liu's results obtained from image recognition [48]. This could be attributed to several factors such as faulty implementation or poor applicability towards Chess engines.

Design choices for the DenseNet architecture were intentionally made differently compared to the original. Implementing changes to the original architecture to better suit the needs of an Antichess engine might have caused degradation in performance instead of the hypothesised improvements.

Additionally, a single DenseNet implementation might not have shown the full potential of the Dense architecture. Therefore, further tests with differences in depth for the DenseNet might have provided more insights into its underperformance when compared to the ResNet. However, due to constraints in time and computational availability, more tests were not conducted due to the implemented DenseNet underperforming in the initial test.

6.1.4 Analysing Performance of the Residual Neural Network

Self-play of the residual neural networks unexpectedly showed similar performance regardless of the model's complexity as seen in figure 5.3. It was hypothesised that the performance would increase with the complexity of the model, as it would allow more complex relationships to be learned. This could be attributed to all models being complex enough to represent the trends in the data. Therefore it would have been beneficial to start with a less complex model to explore if this was the case. Another reason for the similarity in performance could be the models overfitting, therefore not revealing the full potential of each model.

6.1.5 Using Existing Implementations

Using pre-trained implementations of both a ResNet and a DenseNet were considered as alternatives to building the entire network from scratch. However, using existing implementations would still require modifications to output a value and policy. Therefore, it was deemed that the time it took to implement the networks

from scratch would not be greater than the time to adapt the existing implementations. Additionally, the pre-trained implementations were trained on images of greater widths and heights than a chessboard; adapting the network would therefore require changes in the implementation despite using pre-built networks.

By implementing the networks from scratch, the flexibility of the network would increase and allow for parameters to be tuned more easily. This would ultimately increase the points of architectural comparison, leading to a better-performing network in the end.

6.1.6 Analysing Training

Initially we constantly trained different neural network architectures to test which network had the best performance. This would be done for 2-5 generations where the architecture with the most promising trends when minimising the loss would be evaluated more. We thought this approach was the best one in our case. However, after choosing the final architecture and training it for 14 generations, we realised that the network started performing significantly better. The engine started outperforming flat MCTS with search iterations of 2048, which showed that the guidance by the neural network helped improve the intuition of the game tree search. This showed that initial trends when starting to train a network architecture may be misleading, and that the best method may be to commit to an architecture for at least 10 generations of training.

Another observation was that the engine performed better when trained on MCTS with a higher iteration count. The problem with this is that the time per move increases, and with an iteration count of 2048 it takes approximately 10 seconds per move. This can be compared to an iteration count of 512 where each move takes roughly 3 seconds, which shows a linear increase in time. With the time and hardware available in this project it would be unreasonable to train with search iterations greater than 2048. In a different scenario, training with a deeper search could prove worthwhile and would probably increase the performance of the engine further.

6.2 Challenges and Limitations

Several challenges were faced during the project; some were solved with ease while others were more difficult. One of the biggest challenges was our decision to use the library TensorFlow instead of PyTorch, when originally designing the neural network. The reasons for choosing TensorFlow was: the broad use of it, the supposed advantages when creating CNN:s, and compatibility with tasks like image recognition. However, after using TensorFlow for some time we deemed that it was difficult to work with. A lot of problems had occurred that could directly be linked to TensorFlow, one of them being how to save and load trained models. This problem was the ultimate factor that made us switch from TensorFlow to PyTorch. The process of initially using TensorFlow, encountering and resolving the challenges, only to ul-

timately redo all our work with PyTorch instead, proved to be time-consuming. In PyTorch, the workflow was smoother, and the issues we faced were easier to solve. However, it is unclear whether this improvement stemmed from having already tackled the most complex problems, which could also explain our initial difficulties with TensorFlow. Moreover, it is uncertain whether we would have achieved a better result if we continued using TensorFlow, especially since there was a large increase in performance; the model's execution time got 2.5 times faster after the switch to PyTorch.

Another challenge we faced was that the process of designing and testing a neural network is time-consuming. The process primarily relies on trial and error. First, it is necessary to devise and implement your design. Second, the model has to be trained and tested in order to evaluate its performance. Afterwards, the results need to be interpreted and the previous steps repeated as it is an iterative process. However, each test could take multiple hours which leads to a lot of idle time as the project cannot progress until the results are obtained. Additionally, the training and testing was primarily done on a shared computer, resulting in scenarios where it was not possible to train and test as often as we wanted.

6.3 Future Work

Since the results of the project heavily depended on the specifics of the neural network implementation, we thought about how further studies surrounding the network architecture could help improve the performance of the engine. With more knowledge about dense neural networks, it could prove that this architecture can outperform the traditional use of residual neural networks. There may even be other convolutional network architectures that are more suitable for this project. Regarding our use of residual neural networks, additional knowledge would be needed to understand why the engine performed similarly with 4, 8, and 16 residual blocks.

In the end, the predictions of the neural networks were slow, which was a deciding factor as to why we would not test more complex networks further. More knowledge surrounding neural networks could have provided answers if this was because of poor implementation or if this was natural because of the concepts used in our implementation.

Further knowledge around the hyperparameters such as weight decay could have proven helpful when dealing with the overfitting problem. Extended experimentation on the other hyperparameters mentioned in 3.5.2 could also have proven necessary to combat the issues encountered. A specific example is the problem with our use of the Adam optimiser, which could have been a result of poor tuning when initialising the optimiser.

An interesting followup on this project would be to test different types of game tree search algorithms that could be combined with a neural network. This could provide more knowledge on the topic of self-playing chess engines and how one could make

them more efficient with less computational power and time.

The project's architecture of the neural network also holds potential for training on various other chess variants beyond Antichess, offering valuable insights into the performance of the networks across different rule sets and game dynamics. With appropriate adjustments, the network could also be adapted for training on entirely different games, such as Tic-Tac-Toe or Connect Four.

6.4 Social and Ethical Aspects

With the introduction of new technology, it is important to analyse the social and ethical implications beforehand. Technological advancements within Chess are no exception, as they have the potential to greatly affect the state of the sport.

6.4.1 Using Chess Engines to Gain Unfair Advantages

Chess is a game with high outward ethical and moral standards, The International Chess Federation's (FIDE) code of conduct disallows to "act in a manner likely to bring the sport into disrepute" [50]. FIDE, the regulatory body for international Chess [51], manages rules and regulations stating what you can and can not do in a game of Chess. Each player is expected to adhere to the rules and regulations stated by FIDE to contribute to a fair game of Chess.

One of the most important rules states that during a game of Chess, players are not to use any information or advice that could lead to advantageous positions [10]. This prohibits players from using Chess engines, as they give both information and advice regarding the current player's position.

In recent years, with the emergence of accessible strong Chess engines, there have been several occasions where Chess engines have been used to gain an unfair advantage against the opponent, even at professional Chess tournaments [52, 53]. This use of Chess engines contradicts the rules of Chess, as stated by the FIDE regulations [10].

Developing a Chess engine for Antichess could provide additional accessibility to tools that could violate FIDE fair play regulations. Using an engine to gain an advantage can negatively affect the ratings of players who do not cheat.

With invitations to tournaments often being based on rating, it is hence of great importance to have a high Chess rating. As a result, players using an engine could affect others' chances to earn money in these tournaments. While using an engine to cheat can be a large issue, the methodology to prevent that is beyond the scope of this report as it is too time-consuming due to its complexity.

6.4.2 Energy Consumption of Training Deep Neural Networks

AlphaZero, a self-trained engine for Chess, uses deep neural networks (DNN) to predict the best move, resulting in the greatest chance of winning [8]. However, the use of DNNs requires sophisticated and extensive training on both good and bad moves to accurately predict which moves to make. Training a DNN to be very accurate is both an energy-expensive task [54] and a costly task. In a time of increasing global warming, it is of utmost importance to use energy efficiently.

Throughout the development of the self-learning Chess engine, and specifically the evolution of the DNN, it is therefore important to both analyse and invoke measures to limit energy consumption. This could be done by continuously evaluating how different implementations of the DNN affect the energy consumption in relation to the DNNs accuracy in predicting the best move.

6.4.3 Increased Insights into Machine Learning and Neural Networks

AI, machine learning and neural networks are hot topics in today's society, with these methods yet to be fully understood and taken advantage of. With the emergence of effective machine learning algorithms, such as AlphaZero, the knowledge of how machine learning algorithms function and their applications has grown immensely.

Developing a Chess engine could further the knowledge within these fields as a result of increased research. While advancements in Chess engines might not directly impact other applications in society, breakthroughs and new techniques discovered in neural networks could be applicable outside the scope of Chess. Whether a neural network can discern which move to make has very little effect on society but in doing so it is increasing our ability to, and understanding of how to, construct and train a network.

Specifically, self-training is perhaps the most interesting as it reduces the need for a human to construct testing and training data for each possible situation. This is needed to create more general artificial intelligence where it isn't limited to one game or one specific situation.

7

Conclusion

A self trained Antichess engine was developed, composed of a Monte Carlo tree search (MCTS), guided by a neural network. The attempt at training a neural network showed improvements in performance. This resulted in an Antichess engine that utilises the neural network to successfully guide the MCTS, which outperforms one using only MCTS.

The performance of the engine is not ideal, and has many areas of improvement. While demonstrating promising results, the Chess engine operates notably slowly, especially on consumer-grade hardware. This renders any use of the neural network in the engine impractical due to the search being slowed to a crawl. While our network combined with MCTS is not competitive, it does show promise for the ability of a neural network to be utilised to refine searched positions of a MCTS in the domain of Antichess. We believe that the engine has not reached its full potential, which gives reason for further examination of different network architectures. Further studies surrounding the hyperparameters used in training could also provide ways to additionally optimise the network. Despite being slow, the engine demonstrates the ability to apply an AlphaZero style self-trained architecture to a Chess variant.

Bibliography

[1] Salar Jung Museum, "A Game of Thrones - How Chess Conquered the World," Accessed: Jan. 24, 2024. [Online]. Available: https://artsandculture.google.com/story/a-game-of-thrones-how-chess-conquered-the-world-salar-jung-museum/fgUhNlxUQVZ2Kg?hl=en

[2] Wikipedia contributors, "List of chess variants — Wikipedia, the free encyclopedia," 2024, Accessed: Jan. 24, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_chess_variants&oldid=1196418043

[3] S. Simon, C. William, and G. Jan, "Enlightened automata," *The sciences in enlightened Europe*, vol. 105, pp. 126–165, 1999.

[4] M. Stezano, "In 1950, alan turing created a chess computer program that prefigured a.i." 2023, Accessed: Jan. 31, 2024. [Online]. Available: https://www.history.com/news/in-1950-alan-turing-created-a-chess-computer-program-that-prefigured-a-i

[5] F.-h. Hsu, T. S. Anantharaman, M. S. Campbell, and A. Nowatzyk, "Deep thought," in *Computers, Chess, and Cognition*, 1990, pp. 55–78, New York, NY: Springer. [Online]. Available: https://doi.org/10.1007/978-1-4613-9080-0_5

[6] T. Romstad, M. Costalba, J. Kiiski, G. Linscott, Y. Nasu, M. Isozaki, and H. N. *et al.*, "About," Accessed: Feb. 1, 2024. [Online]. Available: https://stockfishchess.org/about/

[7] D. Silver, T. Hubert, J. Schrittwieser, and D. Hassabis, Dec. 2018. [Online]. Available: https://deepmind.google/discover/blog/alphazero-shedding-new-light-on-chess-shogi-and-go/

[8] D. Silver *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017. [Online]. Available: https://arxiv.org/pdf/1712.01815.pdf

[9] G. Verney, *Chess Eccentricities*. Longmans, Green & Company, 1885.

[Online]. Available: https://books.google.se/books?id=2UoqAAAAYAAJ

[10] International Chess Federation (FIDE), "Fide handbook. e. miscellaneous / 01. laws of chess / fide laws of chess taking effect from 1 january 2023," 2022, Accessed: Jan. 26, 2024. [Online]. Available: https://handbook.fide.com/chapter/E012023

[11] Lichess, "Antichess • lose all your pieces (or get stalemated) to win the game." Accessed: Jan. 26, 2024. [Online]. Available: https://lichess.org/variant/antichess

[12] ——, "lichess.org open database," Accessed: May. 09, 2024. [Online]. Available: https://database.lichess.org

[13] C. M. Burnett, "User:cburnett/gfdl images/chess," 2023, accessed: May. 9th, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=User:Cburnett/GFDL_images/Chess&oldid=1178119050

[14] M. Watkins, "Losing chess: 1. e3 wins for white," *ICGA journal*, vol. 39, no. 2, pp. 123–125, 2017. [Online]. Available: https://magma.maths.usyd.edu.au/~watkins/LOSING_CHESS/LCsolved.pdf

[15] International Antichess Federation, "Unsolved openings," Accessed: Feb. 1, 2024. [Online]. Available: https://www.antichess.org/unsolved-openings

[16] A. E. Elo and S. Sloan, *The rating of chessplayers: Past and present*. Arco Publishing, 1979.

[17] C. B. Browne *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. [Online]. Available: https://doi.org/10.1109/TCIAIG.2012.2186810

[18] G. Chaslot, J. Uiterwijk, B. Bouzy, and H. Herik, "Monte-carlo strategies for computer go," *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, Jan. 2006. [Online]. Available: https://www.researchgate.net/publication/240124731_Monte-Carlo_Strategies_for_Computer_Go

[19] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 4, no. 1, 2008, pp. 216–217. [Online]. Available: https://doi.org/10.1609/aiide.v4i1.18700

[20] IBM, "What is a neural network?" Feb. 2024, Accessed: Feb. 8, 2024. [Online]. Available: https://www.ibm.com/topics/neural-networks

[21] T. Szandała, "Bio-inspired neurocomputing," *Studies in Computational*

Intelligence, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2010.09458

[22] Z. Hu, J. Zhang, and Y. Ge, "Handling vanishing gradient problem using artificial derivative," *IEEE Access*, vol. 9, pp. 22 371–22 377, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3054915

[23] J. H. sci, "Relu deep neural networks and linear finite elements," p. 502–527, Jun. 2020. [Online]. Available: https://doi.org/10.48550/arXiv.1807.03973

[24] IBM Data and AI Team, "Ai vs. machine learning vs. deep learning vs. neural networks: What's the difference?" Jul. 2023, Accessed: Feb. 8, 2024. [Online]. Available: https://www.ibm.com/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks/

[25] J. Ansel *et al*, "Pytorch documentation." [Online]. Available: https://pytorch.org/docs/stable/nn.init.html

[26] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/f2925f97bc13ad2852a7a551802feea0-Paper.pdf

[27] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, "Backpropagation and the brain," *Nature Reviews Neuroscience*, vol. 21, no. 6, pp. 335–346, 2020. [Online]. Available: https://doi.org/10.1038/s41583-020-0277-3

[28] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, "Understanding batch normalization," 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/36072923bfc3cf47745d704feb489480-Paper.pdf

[29] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, Y. Lechevallier and G. Saporta, Eds. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. [Online]. Available: https://doi.org/10.1007/978-3-7908-2604-3_16

[30] M. D. Zeiler, "Adadelta: An adaptive learning rate method," 2012. [Online]. Available: https://doi.org/10.48550/arXiv.1212.5701

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980/epoch

[32] A. Gupta, R. Ramanath, J. Shi, and S. S. Keerthi, "Adam vs. sgd: Closing

the generalization gap on image classification," in *OPT2021: 13th Annual Workshop on Optimization for Machine Learning*, 2021. [Online]. Available: https://www.opt-ml.org/papers/2021/paper53.pdf

[33] Google Cloud, "Overview of hyperparameter tuning," 2024, Accessed: May. 13, 2024. [Online]. Available: https://cloud.google.com/vertex-ai/docs/training/hyperparameter-tuning-overview

[34] J. Brownlee, "What is the difference between a batch and an epoch in a neural network," *Machine learning mastery*, vol. 20, 2018. [Online]. Available: https://deeplearning.lipingyang.org/wp-content/uploads/2018/07/What-is-the-Difference-Between-a-Batch-and-an-Epoch-in-a-Neural-Network_.pdf

[35] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1711.00489

[36] J. Raitoharju, "Chapter 3 - convolutional neural networks," pp. 35–69, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780323857871000087

[37] IBM, "What is underfitting?" May. 2024, Accessed: May. 9, 2024. [Online]. Available: https://www.ibm.com/topics/underfitting

[38] ——, "What is overfitting?" May. 2024, Accessed: May. 9, 2024. [Online]. Available: https://www.ibm.com/topics/overfitting

[39] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. [Online]. Available: https://doi.org/10.1109/TNNLS.2021.3084827

[40] Mathworks, "Convolution," accessed March 28th, 2024. [Online]. Available: https://se.mathworks.com/discovery/convolution.html

[41] T. Hamstreet, "Three methods to calculate discrete convolution," September 2023, SSY081; Transforms, signals and systems.

[42] Wikipedia contributors, "Kernel (image processing) — Wikipedia, the free encyclopedia," 2023, [Online; accessed 13-May-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&oldid=1180652863

[43] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*. Cambridge University Press, 2023. [Online]. Available: https://D2L.ai

[44] Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, Thomas B. Schön,

MACHINE LEARNING - A First Course for Engineers and Scientists. Cambridge University Press, 2022. [Online]. Available: https://smlbook.org

[45] IBM, "What are convolutional neural networks?" Accessed: April 10th, 2024. [Online]. Available: https://www.ibm.com/topics/convolutional-neural-networks

[46] L. Lu, "Dying relu and initialization: Theory and numerical examples," *Communications in Computational Physics*, vol. 28, no. 5, p. 1671–1706, jun 2020. [Online]. Available: https://doi.org/10.4208/cicp.OA-2020-0165

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," pp. 770–778, June 2016. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90

[48] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1608.06993

[49] J. Czech, P. Korus, and K. Kersting, "Monte-carlo graph search for alphazero," 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2012.11045

[50] International Chess Federation (FIDE), "Fide handbook. a. administrative subjects / 08. fide ethics / ethics and disciplinary code effective from 1 april 2022," 2022, Accessed: Jan. 31, 2024. [Online]. Available: https://handbook.fide.com/chapter/E012023

[51] ——, "About fide," 2022, Accessed: Jan. 26, 2024. [Online]. Available: https://www.fide.com/fide/about-fide

[52] Wikipedia contributors, "Cheating in chess — Wikipedia, the free encyclopedia," 2024, Accessed: Feb. 7, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cheating_in_chess&oldid=1201311571

[53] Guardian Sport, "Chess grandmaster admits to cheating with phone on toilet during tournament," *The Guardian*, Jul. 2019, Accessed: Feb. 8, 2024. [Online]. Available: https://www.theguardian.com/sport/2019/jul/13/igors-rausis-cheating-phone-tournament-scandal

[54] T.-J. Yang, Y.-H. Chen, J. Emer, and V. Sze, "A method to estimate the energy consumption of deep neural networks," in *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2017, pp. 1916–1920. [Online]. Available: https://doi.org/10.1109/ACSSC.2017.8335698

A

Appendix 1

A.1 ResNet 196k vs Fairy-Stockfish level 2

```
[White "Stockfish level 2"]
[Black "ResNet 196k(2048)"]
[Result "0-1"]
[Variant "Antichess"]
```

1. e3 g6 2. Ba6 bxa6 3. Qf3 e5 4. Qxf7 Kxf7 5. c4 Ba3 6. bxa3 Qe7 7. a4 Qa3 8. Nxa3 Ke8 9. a5 d5 10. cxd5 Nc6 11. dxc6 Bb7 12. cxb7 h6 13. bxa8=K c6 14. Kxa7 e4 15. Kxa6 Nf6 16. Nc2 Nh7 17. Ne2 Nf8 18. h4 g5 19. hxg5 hxg5 20. Rxh8 Kd7 21. Rxf8 Kc8 22. Rxc8 g4 23. Rxc6 g3 24. Nxg3# 0-1

A.2 PGN for two tournament rounds

```
[White "Flat(512)"]
[Black "ResNet 196k(2048)"]
```

1. b4 b6 2. a3 Bb7 3. Bb2 Bxg2 4. Bxg7 Bxg7 5. Bxg2 Bxa1 6. Bxa8 Nc6 7. Bxc6 dxc6 8. b5 Qxd2 9. Qxd2 cxb5 10. Qh6 Nxh6 11. Nc3 Bxc3 12. h3 Bxe1 13. e3 Bxf2 14. h4 Bxh4 15. Rxh4 c5 16. Rxh6 a5 17. Rxh7 Rxh7 18. a4 bxa4 19. Nh3 Rxh3 20. c4 Rxe3 1-0

```
[White "Flat(2048)"]
[Black "ResNet 196k(512)"]
```

1. g3 f5 2. b3 h5 3. Nh3 f4 4. Nxf4 b6 5. Nxh5 Rxh5 6. Bh3 Rxh3 7. Bb2 Rxg3 8. Bxg7 Rxg7 9. Rg1 Rxg1 10. Nc3 Rxe1 11. Qxe1 d5 12. Nxd5 Qxd5 13. h3 Qxb3 14. axb3 Bxh3 15. Rxa7 Rxa7 16. Qa1 Rxa1 17. b4 Kf7 18. d3 Bg2 19. b5 Ra3 20. c3 Rxc3 21. e3 Rxd3 22. e4 Bxe4 23. f3 Rxf3 1-0

```
[White "ResNet 84k(512)"]
[Black "ResNet 84k(2048)"]
```

1. f3 g5 2. e4 Bg7 3. b3 Bxa1 4. Qe2 Bb2 5. Bxb2 d5 6. exd5 Qxd5 7. Qxe7 Nxe7 8. Bxh8 Qxd2 9. Kxd2 b5 10. Bxb5 h5 11. Bxe8 Na6 12. Bxf7 Rb8 13. Bxh5 Rxb3 14. cxb3 Ng6 15. Bxg6 Bd7 16. Bf5 Bxf5 17. g3 Bxb1 18. Bc3 Bxa2 19. Ke3 Bxb3 20. Bg7 Bd5 21. Ke4 Bxe4 22. fxe4 c5 23. Bd4 cxd4 24. g4 Nb4 25. e5 Na6 26. e6 Nb8 27. h3 Nd7 28. exd7 a6 29. Ne2 a5 30. Nxd4 a4 31. h4 gxh4 32. Rxh4 a3 33. d8=K a2 34. Kd7 a1=K 35. Rh5 Ka2 36. Nb3 Kxb3 37. Rh8 Ka3 38. Rh6 Ka2 39. Kd8 Kb1 40. Rh8 Ka1 41. Re8 Kb2 42. Kd7 Kb3 43. Kc6 Ka4 44. Kb5 Kxb5 45. Re7 Ka4 46. Re2 Kb5 47. Re7 Kc4 48. Re8 Kb5 49. Re7 1/2-1/2

```
[White "Random"]
[Black "Flat(512)"]
```

1. f3 Nc6 2. a4 b5 3. axb5 a6 4. bxa6 Rxa6 5. Rxa6 Bxa6 6. b4 Bxe2 7. Bxe2 Nxb4 8. Ba6 Nxc2 9. Qxc2 Qb8 10. Qxc7 Qxc7 11. Nc3 Qxc3 12. dxc3 Kd8 13. Bd3 e6 14. Bxh7 Rxh7 15. Kf2 Rxh2 16. Rxh2 Bb4 17. cxb4 g5 18. Bxg5 f5 19. Bxd8 Ne7 20. Bxe7 d6 21. Bxd6 f4 22. Bxf4 e5 23. Bxe5 0-1

```
[White "ResNet 84k(512)"]
[Black "ResNet 196k(2048)"]
```

1. a3 c5 2. c4 b5 3. cxb5 a5 4. bxa6 Bxa6 5. Qc2 Bxe2 6. Qxh7 Rxh7 7. Bxe2 Rxa3 8. Rxa3 Rxh2 9. Rxh2 Qa5 10. Rxa5 g5 11. Rxc5 Na6 12. Bxa6 d5 13. Rxd5 Kd7 14. Rxd7 e6 15. Rxf7 g4 16. Rxf8 e5 17. Rxg8 g3 18. Rxg3 e4 19. Bf1 e3 20. Rxe3 0-1

```
[White "ResNet 84k(2048)"]
[Black "ResNet 196k(512)"]
```

1. d3 h5 2. g4 hxg4 3. Bh6 Nxh6 4. Bh3 gxh3 5. Nxh3 g5 6. Nxg5 Rh7 7. Nxf7 Nxf7 8. Rg1 Rxh2 9. Rg2 Rxg2 10. Qc1 Rxf2 11. Kxf2 d5 12. Qd2 e5 13. Ke3 Ke7 14. Kd4 exd4 15. Qf4 Qe8 16. Qxf7 Kxf7 17. Nd2 Qxe2 18. c3 Qxd3 19. cxd4 Qxd4 20. Nb3 Qxb2 21. Nc5 Qxa2 22. Rxa2 Bxc5 23. Rxa7 Bxa7 1-0

```
[White "Random"]
[Black "Flat(2048)"]
```

1. a4 Nc6 2. b3 b5 3. axb5 a5 4. Rxa5 Rxa5 5. bxc6 dxc6 6. e3 Qxd2 7. Bxd2 Be6 8. Bxa5 Bxb3 9. cxb3 e5 10. Bxc7 Bd6 11. Qxd6 g5 12. Qxc6 h5 13. Qxe8 e4 14. Qxf7 Nh6 15. Qxh5 Ng8 16. Qxh8 g4 17. Qxg8 g3 18. Qxg3 0-1

```
[White "ResNet 84k(512)"]
[Black "Flat(512)"]
```

1. b3 b6 2. g4 f5 3. gxf5 Na6 4. d4 Nb8 5. f3 g6 6. fxg6 hxg6 7. Bg5 Rxh2 8. Rxh2 Bb7 9. Bxe7 Kxe7 10. Rh4 Bxf3 11. exf3 Bh6 12. Rxh6 Nxh6 13. Bc4 b5 14. Bxb5 Ng4 15. fxg4 Ke6 16. Bxd7 Qxd7 17. Qc1 Qxd4 18. Nf3 Qxg4 19. Qd2 Qxf3 20.

Qa5 Qxb3 21. Qxa7 Qxa2 22. Qxb8 Qxc2 23. Rxa8 Qxb1 24. Qxc7 Qxe1 25. Ra1 Qxa1 26. Qa5 Qxa5 1-0

```
[White "ResNet 196k(512)"]
[Black "ResNet 196k(2048)"]
```

1. b4 b5 2. d4 Bb7 3. g4 Bxh1 4. a4 bxa4 5. Rxa4 Na6 6. Rxa6 Bf3 7. Rxa7 Rxa7 8. exf3 Ra5 9. bxa5 Qc8 10. f4 Qa6 11. Bxa6 h5 12. gxh5 Rxh5 13. Qxh5 f5 14. Qxf5 c5 15. Qxd7 Kxd7 16. dxc5 Kc8 17. Bxc8 e6 18. Bxe6 Bxc5 19. Bxg8 Bxf2 20. Kxf2 g5 21. fxg5 0-1

```
[White "Random"]
[Black "ResNet 84k(512)"]
```

1. g3 f5 2. Nc3 e5 3. Nf3 c5 4. Nxe5 f4 5. gxf4 c4 6. Nxd7 Qxd7 7. Nb5 Qxd2 8. Nxa7 Qxf4 9. Bxf4 Rxa7 10. Bxb8 Rxa2 11. Rxa2 Bh3 12. Bxh3 Kd7 13. Bxd7 Ba3 14. bxa3 b6 15. Rg1 Nh6 16. Rxg7 Rxb8 17. Rxh7 c3 18. Rxh6 Ra8 19. Rxb6 Rxa3 20. Rxa3 0-1

```
[White "ResNet 84k(2048)"]
[Black "Flat(2048)"]
```

1. Na3 b5 2. Nxb5 Ba6 3. Nxc7 Qxc7 4. c4 Qxh2 5. Rxh2 Bxc4 6. Rxh7 Bxa2 7. Rxa2 Rxh7 8. Rxa7 Rxa7 9. Qa4 Rxa4 10. g4 Rxg4 11. Kd1 Rxg1 12. f4 Rxf1 13. e4 Rxf4 14. Ke1 Rxe4 15. d4 Rxe1 16. Be3 Rxe3 17. b3 Rxb3 18. d5 Ra3 19. d6 exd6 1-0

```
[White "ResNet 196k(512)"]
[Black "Flat(512)"]
```

1. Na3 b5 2. Nxb5 d5 3. Nxc7 Qxc7 4. e3 Qxc2 5. Qxc2 h5 6. Qxc8 a5 7. Qxb8 Rxb8 8. h4 Rxb2 9. Bxb2 g6 10. Bxh8 d4 11. exd4 g5 12. hxg5 a4 13. Rxh5 a3 14. g6 fxg6 15. Ke2 gxh5 16. Bg7 Bxg7 17. g4 Bxd4 18. gxh5 Bxf2 19. Kxf2 e6 20. Rd1 Nh6 21. d3 Kf7 22. Bg2 Kg6 23. hxg6 Nf7 24. gxf7 e5 25. f8=R e4 26. dxe4 0-1

```
[White "Random"]
[Black "ResNet 84k(2048)"]
```

1. e3 b5 2. Bxb5 g5 3. Bxd7 Qxd7 4. f3 Qxd2 5. Qxd2 Kd8 6. Qxd8 Bf5 7. Qxb8 Bxc2 8. Qxa8 Bxb1 9. Qxf8 Bxa2 10. Qxg8 Rxg8 11. Rxa2 f6 12. Rxa7 f5 13. Rxc7 g4 14. fxg4 fxg4 15. Rxe7 g3 16. Rxh7 gxh2 17. R7xh2 Rxg2 18. Rxg2 0-1

```
[White "ResNet 196k(512)"]
[Black "ResNet 84k(512)"]
```

1. c4 Nc6 2. b4 Nxb4 3. Qc2 Nxa2 4. Qxh7 Rxh7 5. Rxa2 Rxh2 6. Rxa7 Rxa7 7. Rxh2 Ra2 8. Bb2 Rxb2 9. e3 Rxd2 10. Kxd2 d6 11. Be2 Bh3 12. Rxh3 d5 13. cxd5 Qxd5 14. Rh5 Qxd2 15. Nxd2 b5 16. Rxb5 f5 17. Rxf5 Kd7 18. Rxf8 e5 19. Rxg8 e4 20. Rxg7 c6 21. Rxd7 c5 22. Nxe4 c4 23. Bxc4 0-1

```
[White "ResNet 196k(2048)"]
[Black "Flat(2048)"]
```

1. g4 h5 2. gxh5 Rxh5 3. d3 Rxh2 4. Rxh2 g5 5. Bxg5 Bh6 6. Bxe7 Nxe7 7. Rxh6 f6 8. Rxf6 Nf5 9. Rxf5 Qg5 10. Rxg5 b5 11. Rxb5 a6 12. Rxb8 Rxb8 13. a3 Rxb2 14. Bh3 Rxc2 15. Qxc2 d6 16. Bxc8 a5 17. Qxc7 a4 18. Qxd6 Kd7 19. Bxd7 0-1

```
[White "Random"]
[Black "ResNet 196k(512)"]
```

1. a3 Nc6 2. h4 g5 3. hxg5 Nb4 4. axb4 e6 5. Rxa7 Qxg5 6. Rxa8 Bxb4 7. Rxc8 Qxg2 8. Rxc7 Qxf2 9. Rxh7 Qxe1 10. Rxb7 Qxd2 11. Rxh8 Qxe2 12. Nxe2 Bc5 13. Rxd7 Kxd7 14. Qxd7 f5 15. Qxe6 Bd6 16. Qxd6 Ne7 17. Qxe7 f4 18. Nxf4 0-1

```
[White "ResNet 196k(2048)"]
[Black "ResNet 84k(2048)"]
```

1. g4 h5 2. gxh5 Rxh5 3. Bh3 Rxh3 4. Nxh3 g5 5. Nxg5 Bg7 6. Nxf7 Bxb2 7. Nxd8 Kxd8 8. Bxb2 e5 9. Bxe5 Ke8 10. Bxc7 d6 11. Bxb8 Rxb8 12. d4 Bg4 13. e3 Bxd1 14. Kxd1 Rc8 15. Nc3 Rxc3 16. Ke2 Rxe3 17. Kxe3 Nf6 18. Ke4 Nxe4 19. Rag1 Nxf2 20. Rg3 Nxh1 21. h4 Nxg3 22. h5 Nxh5 23. d5 Kf7 24. c3 Ke6 25. dxe6 Ng3 26. a3 Nh1 27. a4 b5 28. axb5 a5 29. bxa6 Ng3 30. c4 d5 31. cxd5 Ne4 32. a7 Ng5 33. d6 Nxe6 34. d7 Ng7 35. d8=B Nh5 36. Bf6 Nxf6 37. a8=N Ng4 38. Nc7 Nh2 39. Nd5 Nf3 40. Ne3 Ng5 41. Nc2 Nf7 42. Nd4 Nh8 43. Nc6 Nf7 44. Nd8 Nxd8 1-0

```
[White "Flat(512)"]
[Black "Flat(2048)"]
```

1. Na3 b5 2. Nxb5 Nc6 3. Nxc7 Qxc7 4. b4 Qxh2 5. Rxh2 Nxb4 6. Rxh7 Nxc2 7. Rxg7 Nxe1 8. Qxe1 Bxg7 9. a4 Bxa1 10. d4 Bxd4 11. Bb2 Bxf2 12. Bxh8 Bxg1 13. Qa5 Bd4 14. Qxa7 Bxh8 15. Qxd7 Rxa4 16. Qxe7 Kxe7 17. g4 Rxg4 18. e4 Rxe4 19. Ba6 Bxa6 1-0

```
[White "Random"]
[Black "ResNet 196k(2048)"]
```

1. c3 d5 2. a3 Bh3 3. gxh3 e6 4. Ra2 Bxa3 5. Rxa3 d4 6. Rxa7 dxc3 7. Rxb7 cxb2 8. Rxb2 Qxd2 9. Kxd2 Ra3 10. Nxa3 f5 11. Rxb8 f4 12. Rxe8 f3 13. exf3 e5 14. Rxe5 c5 15. Rxc5 g6 16. Ra5 g5 17. Rxg5 h6 18. Rxg8 Rxg8 19. Ke2 Rxg1 20. Rxg1 h5 21. Nb5 h4 22. Rg3 hxg3 23. hxg3 0-1

[White "Flat(512)"]
[Black "ResNet 84k(2048)"]

1. d3 b6 2. Nc3 b5 3. Nxb5 g5 4. Nxc7 Qxc7 5. Bxg5 Qxc2 6. Bxe7 Kxe7 7. Qxc2 d5 8. Qxc8 Kd6 9. Qxb8 Rxb8 10. Rb1 Rxb2 11. Rxb2 Kc5 12. d4 Kxd4 13. Rb4 Bxb4 14. g3 Bxe1 15. h3 Bxf2 16. Rh2 Bxg3 17. a3 Bxh2 18. Bg2 Bxg1 19. Bxd5 Kxd5 20. e4 Kxe4 21. a4 a5 22. h4 Kd3 23. h5 Bh2 24. h6 Nxh6 1-0

[White "Flat(2048)"]
[Black "ResNet 84k(512)"]

1. e4 Nc6 2. b4 Nxb4 3. Qf3 Nxa2 4. Qxf7 Kxf7 5. Rxa2 e6 6. Rxa7 Rxa7 7. Ba3 Bxa3 8. Nxa3 Rxa3 9. Ba6 bxa6 10. h3 Rxh3 11. Rxh3 Qg5 12. Rxh7 Qxd2 13. Kxd2 Rxh7 14. Ke3 Kg6 15. Kd3 Rh3 16. Nxh3 Kh7 17. Kc4 c5 18. Kxc5 g5 19. Nxg5 d6 20. Nxh7 dxc5 21. Nf6 Nxf6 22. g3 Nxe4 23. g4 Nxf2 24. c4 Nxg4 1-0

[White "ResNet 196k(2048)"]
[Black "Flat(512)"]

1. d3 c6 2. Bg5 e5 3. Bxd8 Kxd8 4. d4 exd4 5. Qxd4 g6 6. Qxd7 Kxd7 7. Nh3 Kd6 8. Rg1 Bxh3 9. gxh3 g5 10. Rxg5 Bg7 11. Rxg7 a6 12. Rxg8 Rxg8 13. Bg2 Rxg2 14. Kd1 Rxh2 15. h4 Rxf2 16. Kc1 Rxe2 17. Nd2 Rxd2 18. Kxd2 a5 19. Rd1 h5 20. b4 axb4 21. Ke2 Rxa2 22. Rxd6 Rxc2 23. Rxc6 Rxc6 24. Kd1 Rc1 25. Kxc1 f5 26. Kd2 b3 27. Kd3 b2 28. Ke4 fxe4 1-0

[White "ResNet 196k(512)"]
[Black "Flat(2048)"]

1. b4 c5 2. bxc5 Qb6 3. cxb6 axb6 4. g3 Rxa2 5. Rxa2 d5 6. c4 dxc4 7. Qb3 cxb3 8. e4 bxa2 9. Ne2 axb1=N 10. g4 Bxg4 11. Rg1 Nxd2 12. Bxd2 Bxe2 13. Bxe2 b5 14. Bxb5 Nh6 15. Rxg7 Bxg7 16. Bxh6 Bxh6 17. Bxe8 Rxe8 18. Kd2 Bxd2 19. h4 Bg5 20. hxg5 h6 21. gxh6 Rg8 22. e5 Rg7 23. hxg7 f5 24. exf6 exf6 25. g8=K Nc6 26. Kh7 Nd8 27. Kg8 Nf7 28. Kxf7 b5 29. Kxf6 b4 30. Ke7 b3 31. f4 b2 32. f5 b1=K 33. Ke6 Ka2 34. Ke7 Ka1 35. Kd6 Kb2 36. Kc7 Kb1 37. Kc6 Ka1 38. Kd6 Ka2 39. Kc6 Ka3 40. Kc7 Ka4 41. Kb7 Ka5 42. Kb6 Kxb6 43. f6 Kb7 44. f7 Kc7 45. f8=K Kc8 46. Kg7 Kd8 47. Kg6 Kd7 48. Kg5 Kd6 49. Kg6 Kd5 50. Kg5 Kc6 51. Kg4 Kb6 52. Kf4 Kb7 53. Kf5 Kb6 54. Ke6 Kb7 55. Kf5 Ka6 56. Ke4 Kb7 57. Kf4 Ka6 58. Kf3 Kb7 59. Kg4 Ka6 60. Kh5 Kb5 61. Kg4 Kb6 62. Kh5 Kc6 63. Kh4 Kb6 64. Kg5 Kb7 65. Kg4 Kc8 66. Kg3 Kb7 67. Kh3 Kb6 68. Kh4 Kb7 69. Kg5 Kb6 70. Kf5 Kc6 71. Kf6 Kb6 72. Ke5 Kb7 73. Kd4 Kc7 74. Ke4 Kd7 75. Kd4 Ke7 76. Kc4 Kd7 77. Kd4 Ke7 78. Ke4 Kf7 79. Kf3 Kf6 80. Ke3 Kg6 81. Kf3 Kh6 82. Ke3 Kh5 83. Ke2 Kh4 84. Kf1 Kg4 85. Kg1 Kh4 86. Kf1 Kg4 87. Kg1 Kh4 88. Kh1 Kg4 89. Kg1 1/2-1/2

[White "ResNet 84k(2048)"]
[Black "ResNet 84k(512)"]

1. h4 g5 2. hxg5 e6 3. Rxh7 Qxg5 4. Rxf7 Qxg2 5. Bxg2 Kxf7 6. Bxb7 Bxb7 7. e4 Bxe4 8. Qg4 Bxc2 9. Qxe6 Bxb1 10. Rxb1 dxe6 11. Nh3 Rxh3 12. d3 Rxd3 13. Bh6 Bxh6 14. Kd2 Bxd2 15. f4 Bxf4 16. Rg1 a6 17. Rxg8 Kxg8 18. a4 Rd7 19. b3 Kf7 20. a5 Be3 21. b4 Bf4 22. b5 axb5 23. a6 Rxa6 1-0

```
[White "Flat(512)"]
[Black "Random"]
```

1. e4 g6 2. b3 c6 3. Bc4 c5 4. Bxf7 Kxf7 5. Qg4 Ke8 6. Qxg6 hxg6 7. b4 Rxh2 8. bxc5 Rxh1 9. Kd1 Rxg1 10. d3 Rxd1 11. f4 Rxd3 12. cxd3 Qc7 13. e5 Qxe5 14. fxe5 Na6 15. g3 Nxc5 16. a4 Nxd3 17. g4 Nxc1 18. Ra2 Nxa2 19. e6 dxe6 20. Nc3 Nxc3 21. g5 Nxa4 1-0

```
[White "ResNet 196k(2048)"]
[Black "ResNet 84k(512)"]
```

1. g4 f5 2. gxf5 h5 3. Bg2 b6 4. Bxa8 g5 5. fxg6 e5 6. Bb7 Bxb7 7. b3 Bxh1 8. g7 Bxg7 9. d4 exd4 10. Qxd4 Bxd4 11. e4 Bxf2 12. Kxf2 Bxe4 13. Bh6 Bxc2 14. Nc3 Nxh6 15. Ne4 Bxb3 16. axb3 d6 17. Nxd6 cxd6 18. Rxa7 Rh7 19. Rxh7 d5 20. Rxh6 Qd7 21. Rxb6 Qa4 22. Rxb8 Qxb3 23. Rxe8 Qf3 24. Kxf3 h4 25. Kg3 hxg3 26. hxg3 d4 27. Rg8 d3 28. Ne2 dxe2 29. Rf8 e1=N 30. Rf3 Nxf3 31. g4 Nh2 32. g5 Nf3 33. g6 Nd2 34. g7 Nf1 35. g8=B Ng3 36. Bb3 Ne2 37. Bd1 Ng3 38. Bh5 Nxh5 1-0

```
[White "ResNet 196k(512)"]
[Black "ResNet 84k(2048)"]
```

1. c4 Nf6 2. g4 Nxg4 3. Bh3 Nxf2 4. Kxf2 d5 5. cxd5 Qxd5 6. Bxc8 Qxd2 7. Bxb7 Qxe2 8. Bxa8 Qxb2 9. Bxb2 g5 10. Bxh8 Nc6 11. Bxc6 g4 12. Qxg4 a5 13. Bxe8 h6 14. Bxf7 a4 15. Qxa4 h5 16. Bxh5 c6 17. Qxc6 e6 18. Qxe6 Bg7 19. Bxg7 0-1

```
[White "Flat(2048)"]
[Black "Random"]
```

1. Nc3 Nf6 2. Ne4 Nxe4 3. e3 Nxd2 4. Bxd2 e5 5. Ba6 Nxa6 6. b4 Nxb4 7. Bxb4 Bxb4 8. Qxd7 Bxd7 9. Kd2 Bxd2 10. Ne2 Bxe3 11. fxe3 Bc6 12. Rhb1 Bxg2 13. Rxb7 Bxb7 14. Re1 Rf8 15. Rd1 Qxd1 16. Ng1 Qxg1 17. e4 Qxh2 18. c4 Qxa2 19. c5 Bxe4 20. c6 Bxc6 1-0

```
[White "Flat(512)"]
[Black "ResNet 84k(512)"]
```

1. Na3 b5 2. Nxb5 e6 3. Nxa7 Rxa7 4. Nf3 Rxa2 5. Rxa2 Qe7 6. Nh4 Qxh4 7. Ra6 Qxh2 8. Rxh2 Bxa6 9. Rxh7 Bxe2 10. Qxe2 Rxh7 11. Qxe6 fxe6 12. Ba6 Nxa6 13. b3 Rh4 14. Ba3 Bxa3 15. d4 Rxd4 16. f4 Rxf4 17. Kf2 Rxf2 18. c4 Rxg2 19. c5 Bxc5 20. b4 Bxb4 1-0

```
[White "ResNet 196k(2048)"]
[Black "ResNet 196k(512)"]
```

1. e4 c5 2. Ba6 bxa6 3. b4 cxb4 4. Nc3 bxc3 5. dxc3 d5 6. Qxd5 Qxd5 7. exd5 Bh3 8. gxh3 Nc6 9. dxc6 Kd7 10. cxd7 Re8 11. dxe8=N g5 12. Bxg5 Nf6 13. Bxf6 exf6 14. Nxf6 Bc5 15. Nxh7 Bxf2 16. Kxf2 Rxh7 17. Nf3 Rxh3 18. Ng1 Rxh2 19. Rxh2 a5 20. Rh8 a6 21. Kf1 a4 22. Ke1 a3 23. Rb8 f6 24. Rb2 axb2 25. Nh3 bxa1=N 26. Ng5 Nxc2 27. Ke2 fxg5 28. Ke3 Nxe3 29. c4 Nxc4 30. a3 Nxa3 1-0

```
[White "ResNet 84k(512)"]
[Black "Random"]
```

1. a3 a6 2. d4 e6 3. Bh6 gxh6 4. Kd2 Bxa3 5. Rxa3 Ne7 6. Rxa6 Rxa6 7. Ke1 Rf8 8. Na3 Rxa3 9. bxa3 Ng8 10. h4 Qxh4 11. Rxh4 Na6 12. Rxh6 Nxh6 13. g4 Nxg4 14. Bg2 Nxf2 15. Bxb7 Bxb7 16. Kxf2 d6 17. d5 Bxd5 18. Qxd5 exd5 19. a4 Nb4 20. Ke3 Nxc2 21. Ke4 dxe4 22. a5 Kd8 23. Nh3 h6 24. Ng5 hxg5 25. a6 Kd7 26. a7 c6 27. a8=R Rxa8 28. e3 Nxe3 1-0

```
[White "Flat(2048)"]
[Black "ResNet 84k(2048)"]
```

1. Nh3 g5 2. Nxg5 Bh6 3. Nxf7 Bxd2 4. Qxd2 Kxf7 5. Qxd7 Qxd7 6. Kd1 Qxd1 7. g4 Qxe2 8. Bxe2 Bxg4 9. Bxg4 Nh6 10. Bxh6 Rc8 11. Bxc8 Kg7 12. Bxb7 Kxh6 13. Bxa8 Nc6 14. Bxc6 a6 15. Bb5 axb5 16. h3 Kg6 17. h4 Kg5 18. hxg5 b4 19. Rxh7 b3 20. Rxe7 bxc2 21. Rxc7 cxb1=B 22. Rxb1 0-1

```
[White "Flat(512)"]
[Black "ResNet 196k(512)"]
```

1. b4 c5 2. bxc5 Qb6 3. cxb6 axb6 4. e3 Rxa2 5. Rxa2 g5 6. Ra3 e6 7. Nf3 Bxa3 8. Nxg5 Bxc1 9. Nxe6 fxe6 10. Qxc1 d6 11. Ba6 Nxa6 12. Qb2 Nc5 13. Qxb6 Kf7 14. Qxb7 Nxb7 15. c3 Ne7 16. Kd1 Re8 17. Kc2 Nd5 18. Rc1 Nxc3 19. dxc3 e5 20. Na3 Kf8 21. Rb1 Bd7 22. Rxb7 Rb8 23. Rxd7 h6 24. Rxd6 Rc8 25. Rxh6 Rxc3 26. Kxc3 Ke8 27. Rh7 Kd7 28. Rxd7 e4 29. Rd1 0-1

```
[White "ResNet 84k(2048)"]
[Black "Random"]
```

1. e4 c5 2. Nc3 h5 3. Qxh5 Rxh5 4. Ba6 Nxa6 5. b4 Rxh2 6. bxc5 Rxg2 7. Rh3 Rxg1 8. Rd3 Rxe1 9. Rxd7 Kxd7 10. Nb5 Rxc1 11. Nxa7 Rxc2 12. Nxc8 Rxc8 13. f3 Rxd2 14. Rc1 Rxa2 15. Rg1 Rxc5 16. Rxg7 Bxg7 17. f4 e5 18. fxe5 Rxe5 1-0

```
[White "ResNet 84k(512)"]
[Black "ResNet 196k(512)"]
```

1. g4 h5 2. gxh5 Rxh5 3. Bh3 Rxh3 4. Nxh3 e6 5. b4 Bxb4 6. Na3 Bxd2 7. Qxd2

b5 8. Qxd7 Nxd7 9. Nxb5 Qg5 10. Nxc7 Qxc1 11. Nxe8 Qxe1 12. Nxg7 Qxf2 13. Nxe6 fxe6 14. Nxf2 a6 15. Rhg1 Ra7 16. Rxg8 Nf6 17. Rxc8 Ne4 18. Nxe4 Ra8 19. Rxa8 a5 20. Rxa5 e5 21. Rxe5 0-1

```
[White "Flat(2048)"]
[Black "ResNet 196k(2048)"]
```

1. b4 c5 2. bxc5 Qb6 3. cxb6 axb6 4. Ba3 Rxa3 5. Nxa3 Nh6 6. g3 b5 7. Nxb5 Rg8 8. Qb1 b6 9. g4 Nxg4 10. Qb3 Nxf2 11. Qxf7 Kxf7 12. Kxf2 d5 13. Nd6 exd6 14. e4 dxe4 15. Ba6 Bxa6 16. Rf1 Bxf1 17. Kxf1 e3 18. dxe3 Nc6 19. h3 Nd4 20. exd4 g5 21. d5 g4 22. hxg4 Rxg4 23. Rxh7 Rxg1 24. Rxf7 Rxf1 25. Rxf8 Rxf8 26. c3 Rf1 27. c4 Rb1 28. c5 dxc5 29. a3 Rb4 30. axb4 cxb4 31. d6 b3 32. d7 b2 33. d8=N b5 34. Nf7 b1=K 35. Ne5 b4 36. Nf3 Kb2 37. Ng1 Kc3 38. Nh3 Kd3 39. Ng1 Kc2 40. Nh3 Kd3 41. Ng1 Kc2 42. Nh3 b3 43. Ng1 b2 44. Nh3 Kd3 45. Ng5 Kc4 46. Nf3 Kd4 47. Nxd4 b1=N 48. Ne6 Nc3 49. Nd8 Ne4 50. Nc6 Nd2 51. Nd8 Nc4 52. Ne6 Ne5 53. Nc7 Ng4 54. Nb5 Ne3 55. Na7 Nc4 56. Nb5 Na3 57. Nxa3 0-1

```
[White "ResNet 196k(512)"]
[Black "Random"]
```

1. h4 g5 2. hxg5 e6 3. Rxh7 Rxh7 4. e3 Qxg5 5. Qe2 Qxg2 6. Bxg2 Rh5 7. Bxb7 Bxb7 8. Qxh5 Bc6 9. Qxf7 Kxf7 10. Nf3 Bxf3 11. Kd1 Bxd1 12. a4 Bxc2 13. Ra3 Bxa3 14. Nxa3 Bxa4 15. Nc2 Bxc2 16. e4 Bxe4 17. d4 Bb7 18. d5 exd5 19. Bd2 Kf6 20. Bg5 Kxg5 21. f4 Kxf4 22. b4 Nc6 23. b5 Rd8 24. bxc6 Bxc6 1-0

```
[White "ResNet 84k(2048)"]
[Black "ResNet 196k(2048)"]
```

1. h4 e5 2. d4 exd4 3. Qxd4 Qxh4 4. Qxd7 Bxd7 5. Rxh4 Ba3 6. Rxh7 Rxh7 7. bxa3 Rh3 8. Nxh3 Bxh3 9. gxh3 g6 10. Bh6 Nxh6 11. Nc3 Ng4 12. hxg4 b5 13. Nxb5 a5 14. Nxc7 g5 15. Nxe8 f6 16. Nxf6 Nd7 17. Nxd7 Rb8 18. Nxb8 a4 19. f4 gxf4 20. e4 fxe3 21. Kd2 exd2 22. Re1 dxe1=K 23. c3 Kxf1 24. c4 Ke2 25. g5 Ke3 26. g6 Ke4 27. Na6 Kd5 28. cxd5 0-1

```
[White "Flat(2048)"]
[Black "Flat(512)"]
```

1. g4 a6 2. f4 f5 3. gxf5 g6 4. fxg6 hxg6 5. f5 gxf5 6. e4 fxe4 7. Bxa6 Rxh2 8. Rxh2 Rxa6 9. Rh6 Rxa2 10. Rxa2 Bxh6 11. Ra6 bxa6 12. d3 exd3 13. Qxd3 Bxc1 14. Qxd7 Bxb2 15. Qxc7 Qxc7 16. c3 Bxc3 17. Nxc3 Qxc3 18. Nh3 Qxh3 19. Kf1 Qxf1 1-0

```
[White "ResNet 196k(2048)"]
[Black "Random"]
```

1. e3 Nh6 2. Nc3 Na6 3. Bxa6 bxa6 4. Nb5 axb5 5. Qg4 Nxg4 6. d3 Nxh2 7. Rxh2 d6 8. Rxh7 Rxh7 9. Nh3 Bxh3 10. gxh3 Rxh3 11. a4 Rxe3 12. axb5 Rxd3 13. Rxa7 Rxa7 14. cxd3 Ra4 15. b4 Rxb4 16. Bd2 Rxb5 17. Bf4 Rd5 18. Bxd6 exd6 19. d4 Rxd4 20. f4 Rxf4 21. Kf2 Rxf2 1-0

```
[White "ResNet 84k(2048)"]
[Black "Flat(512)"]
```

1. e3 c6 2. Ba6 bxa6 3. c4 g5 4. Qf3 c5 5. Qxf7 Kxf7 6. h4 gxh4 7. Rxh4 d5 8. cxd5 Qxd5 9. Rxh7 Qxg2 10. Rxh8 Qxf2 11. Rxg8 Kxg8 12. Kxf2 Bh3 13. Nxh3 Bg7 14. b4 Bxa1 15. bxc5 Bc3 16. dxc3 Kg7 17. Bb2 Kh8 18. e4 Kh7 19. Ng5 e6 20. Nxe6 Nd7 21. Nd8 Rxd8 22. Na3 Nxc5 23. Bc1 Nxe4 24. Ke3 Nxc3 25. Kd2 Nxa2 26. Nb5 Nxc1 27. Nxa7 Rxd2 28. Nb5 axb5 1-0

```
[White "ResNet 84k(512)"]
[Black "Flat(2048)"]
```

1. g4 b5 2. d3 a5 3. Be3 Ra7 4. Bxa7 e5 5. Bxb8 Ba3 6. Bxc7 Qxc7 7. Nxa3 Qxc2 8. Nxb5 Qxe2 9. Kxe2 d6 10. Nxd6 Bxg4 11. Nxf7 Bxe2 12. Nxe2 Kxf7 13. Ng1 Nh6 14. Qg4 Nxg4 15. b4 Nxf2 16. bxa5 Nxd3 17. Bxd3 Ra8 18. Bxh7 Rxa5 19. Rd1 Rxa2 20. Rd2 Rxd2 21. Bg6 Rxh2 22. Bxf7 Rxh1 23. Ba2 Rxg1 24. Bb1 Rxb1 1-0