FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Neural Networks in Chess Game
Student:	Jakub Zeman
Supervisor:	Ing. Mgr. Ladislava Smítková Janků, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

Neural Networks in Chess AI
1) Write a survey of existing approaches to two-player board games (Go, chess) based on neural networks.
2) Prepare training data from automatically generated chessboards. Use existing chess game programs or databases.
3) Study the connection of MiniMax method with Neural Network output for chess game.
4) Propose the method for identification of ResNet architecture for the identification of the best move of the stockfish engine and the chess board evaluation at a given depth.
5) Compare AI control mechanism based on reinforcement learning and supervised learning
6) Perform and evaluate experiments and discuss the results.

Literature:
[1] Oshri Barak and Nishith Khandwala, Predicting Moves in Chess using Convolutional Neural Networks, Stanford University, 2015.
http://cs231n.stanford.edu/reports/2015/pdfs/ConvChess.pdf
[2] Lai, M. Giraffe: Using Deep Reinforcement Learning to Play Chess, Ph.D. Thesis
https://arxiv.org/abs/1509.01549v1
[3] M. Samadi, Z. Azimifar and M. Z. Jahromi, "Learning: An Effective Approach in Endgame Chess Board Evaluation," Sixth International Conference on Machine Learning and Applications (ICMLA 2007), Cincinnati, OH, USA, 2007, pp. 464-469, doi: 10.1109/ICMLA.2007.48

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 9 February 2023 in Prague.

https://ieeexplore.ieee.org/document/4457273
[4] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016).
https://www.nature.com/articles/nature16961
[5] Silver, D., Schrittwieser, J., Simonyan, K. et al. Mastering the game of Go without human knowledge. Nature 550, 354–359 (2017).
[6] H. Panchal, S. Mishra and V. Shrivastava, "Chess Moves Prediction using Deep Learning Neural Networks," 2021 International Conference on Advances in Computing and Communications (ICACC), Kochi, Kakkanad, India, 2021, pp. 1-6

Bachelor's thesis

NEURAL NETWORKS IN CHESS GAME

Jakub Zeman

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Mgr. Ladislava Smítková Janků, Ph.D.
May 11, 2023

Citation of this thesis: Zeman Jakub. *Neural Networks in Chess Game*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

Firstly, I would like to thank my supervisor Mrs. Ladislava Smítková Janků, for her supervision throughout this project. Furthermore, I am grateful to a dear friend of mine and an excellent chess player, Jiří Rýdl, for his expert analysis and comments on the chess boards positions used in this thesis. His input has significantly enriched the quality and readability of the work presented here. Lastly, I would like to express my heartfelt appreciation to my mother and father for their generous support in covering the cost of higher electricity bills. Without that electricity, all these computations wouldn't be possible.

Declaration

In Prague on May 11, 2023 ....

Abstract

This thesis aims to investigate the application of neural networks in a chess engine. Neural networks are trained using supervised and reinforcement learning. The supervised learning part of the thesis involves automatic dataset generation from the Stockfish chess engine. This part also includes an analysis of the relationship between the MiniMax algorithm and the output of a neural network's single forward pass, as well as the connection between the depth of the generated dataset and the use of the appropriate type of ResNet architecture. Proposes metrics of board complexity and reveals limitations of convolutional neural networks to find and utilize information hidden in the game tree. In addition, the thesis compares multiple approaches to reinforcement learning of the chess engine, proposes novel method to reinforcement learning, which achieved better results than standard method in two practical experiments.

Keywords neural networks, chess, deep learning, supervised learning, reinforcement learning, MiniMax algorithm, state space search

Abstrakt

Tato práce se zaměřuje na využití neuronových sítí v šachovém enginu. Neuronové sítě jsou trénovány jak pomocí supervizovaného, tak pomocí posilovaného učení. Pro supervizovanou část práce byl navržen algoritmus automatického generování dat ze šachového enginu Stockfish. Tato část také obsahuje analýzu vztahu mezi MiniMax algoritmem a výstupem dopředného průchodu neuronovou sítí, dále vztah mezi hloubkou generovaných dat a použitím vhodného typu architektury ResNet. Navrhuje metriku komplexity šachovnice a odhaluje limity konvolučních neuronových sítí najít a využít informaci skrytou v herním stromě. V druhé části práce obsahuje srovnání více přístupů k posilovanému učení šachového enginu, navrhuje novou metodu k tréninku, která dosahuje lepších výsledků než standartní metoda ve dvou praktických experimentech.

Klíčová slova neuronové sítě, šachy, hluboké učení, supervizované učení, posilované učení, MiniMax algoritmus, prohledávání stavového prostoru

List of abbreviations

MSE	mean squared error
RMSE	root mean squared error
CE	cross entropy
CNN	convolutional neural network
MLE	maximum likelihood estimation
TanH	hyperbolic tangens
ReLU	rectified linear unit
QSM	quick search space algorithm for mate seeking

Chapter 1

Introduction

The game of chess is one of the oldest domains of artificial intelligence research. Due to its perfect information nature, where all game information is available to both players and high complexity of the game tree, it provides an ideal platform to test the capabilities of artificial intelligence systems in state space search.

In 1951, Claude Shannon developed the first computer chess engine. In 1996, the Deep Blue team developed the first chess engine capable of defeating reigning world champion Garry Kasparov [1]. Since then, many chess engines have been created, and the competition between them is still ongoing.

The earliest chess engines were expertly designed and relied on a lot of heuristics. In order to achieve superhuman performance, hundreds millions of positions had to be searched and evaluated. However, availability of neural networks has made the pruning of state-space more effective, enabling chess engines to become increasingly powerful without the need for searching such high numbers of positions.

The study of artificial intelligence capabilities extends beyond the game of chess. While it may seem like a simple game with no relevance to other domains, the state-space search problem it presents is a common challenge in many other areas. This problem involves predicting which parts of the state space are relevant and evaluating non-terminal states, a fundamental challenge in many other domains. Therefore, studying the state-space search problem in the context of chess can have far-reaching implications.

Taking a data science approach to chess, stepping outside of human-made heuristics and expertly designed chess engines, offers further opportunities for generality. Neural networks-based chess engines can be trained in a supervised manner, where labeled datasets are used for learning, or in a reinforcement learning manner, where the agent learns from its own experience in the environment. The latter approach has become increasingly popular, for instance programs with similar architecture to the chess engine AlphaZero can be generalized to discover new algorithms for matrix multiplication.[1]

That is why I believe that studying artificial intelligence, supervised and reinforcement learning in the context of searching the state space can be essential for many future applications, even outside the field of board games.

[1]`https://www.nature.com/articles/s41586-022-05172-4`

Chapter 2
Goals

The main goal of this thesis is to investigate the game of chess through a data-science perspective with a specific focus on capabilities of convolutional neural networks for board position evaluation and move prediction. Furthermore their deployment in a game engine as well as various approaches to training. In particular, the sub-tasks that should be met are stated as follows:

2.1 Training data

Prepare training data for the supervised learning method from automatically generated chess board positions.

There is a lot of training data already available from human players. However, it is possible to generate a balanced and high-quality dataset using automated methods with open-source software. One of the goals of this thesis is to propose a method for automatic chess board position generation with possibility to create well defined set with specified parameters, position evaluation and probabilities of the next moves.

2.2 MinMax algorithm connection

Study the connection of MiniMax method with Neural Network output for chess game.

Main objectives of neural networks in a chess engine are board position evaluation and predicting the next move. These tasks are usually accomplished by simulating certain scenarios and switching perspectives in the game tree i.e. typically using MiniMax algorithm. One of the goals of this thesis is to determine if neural networks are capable of simulating the MiniMax algorithm and evaluating board positions and moves based on possible future gameplay in a single forward pass.

2.3 Resnet type and depth of search

Propose a method for identification of ResNet architecture for the identification of the best move of the Stockfish engine and the chess board evaluation at a given depth.

Although the MiniMax algorithm is not directly present in the neural network, fitting position evaluations of MiniMax run to higher depths requires approximating a more complex function than lower depths. Thus, a more complex model may potentially be able to fit higher depths evaluations. One of the goals of this thesis is to study this connection.

2.4 Supervised and reinforcement learning

Compare AI control mechanism based on reinforcement learning and supervised learning.

Neural network-based chess engines can be trained in a supervised manner or without any supervision using reinforcement learning. One advantage of supervised learning is that it needs much less data to train, but it is not usually able to achieve such performance as reinforcement learning methods. Differentiating between these two principles is a key part of proposing a reinforcement method that can learn without the limitations of human supervision. One of the goals of this thesis is to study both principles in the context of the game of chess. First, this topic can be analyzed mainly in the theoretical part, secondly can be further supported with practical experiments.

Chapter 3

Theoretical Background

This chapter provides the essential theoretical foundations required for this thesis. Section 3.1 describes algorithmic approaches to playing the game of chess, section 3.2 then describes the theory behind neural networks. Additionally, the 3.3 section covers the basic principles of reinforcement learning.

3.1 Computer approach to the game of chess

3.1.1 Game tree

An algorithmic approach to playing the game of chess typically requires the creation of a game tree. This game tree is a tree-like graph structure where the nodes represent the chess board positions, and the edges represent the available moves from a particular position. [2]

3.1.1.1 Complexity of the tree

Different games can have game trees of different complexity. This complexity can be measured by the scaling factor, which corresponds to the ratio of nodes between layers. The game of chess has a game tree of moderate complexity, with scaling factor estimated to be between 35-40. Comparatively simpler games such as checkers have scaling factor around 5-6, while more complex games such as Go have scaling factor estimated to be around 250-300. [2]

3.1.2 MiniMax algorithm

There are multiple approaches to utilizing the game tree to solve the game of chess. One such algorithm that uses the game tree is the MiniMax algorithm. This algorithm expands the game tree, evaluates terminal nodes, and backpropagates the results, taking the maximum value for the player on turn and the minimum value for the opponent. The purpose of this algorithm is to find a move that has the biggest benefit for the player on turn, while assuming that the opponent will play optimally. [2]

3.1.2.1 Heuristics functions

MiniMax algorithm approach is suitable for games with small game tree complexity. However, it is impossible to construct a game tree so large for the game of chess. Therefore, the tree must be cut at a given depth, and non-terminal nodes must be evaluated using a heuristic evaluation function that estimates the outcome of the game from non-terminal nodes.

There are also other heuristic functions, such as heuristics for ordering moves by strength, which can significantly reduce the scaling factor of the tree while used with alpha-beta pruning. [2]

3.1.3 Monte carlo tree search

Another common approach to solving board games is using the Monte Carlo tree search algorithm. This algorithm expands the game tree and simulates possible gameplay, the results are then backpropagated using statistical information about the outcome of simulated games. As more simulations are executed, optimal move selection becomes more precise. In contrast to the MiniMax algorithm, which is very effective for games that allow for the creation of precise heuristics for position evaluation and have a low scaling factor of the game tree, the MCTS algorithm has its benefits in more complex games. This is because the best move is not selected based on the extreme values in evaluation, but rather based on the average of multiple simulations. [3][4][5][6]

3.2 Neural networks

Neural networks are one of the most complex machine learning models with similar structure and learning principle to logistic regression. The main building blocks of neural networks are neurons, which are structured in multiple layers to create a graph structure. From a graph perspective, the nodes are called neurons, and the edges are called weights. The value of each neuron is computed as the outcome of activation function when given the input of the sum of all ancestral neurons multiplied by the weights on the connecting edges. There are various activation functions available, with one standard function, which limits the output to the range of 0 to 1, being the *sigmoid* function:

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{3.1}$$

Another common activation function, which limits the output to the range of -1 to 1, is the $TanH$ function:

$$TanH(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.2}$$

A frequently used activation function is also the *ReLU* function:

$$ReLU(x) = \max\{0, x\} \tag{3.3}$$

In the case of a multi-class label classification problem, the SoftMax function is being used. The SoftMax function is defined as follows:

$$SoftMax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.4}$$

where x_i is the input for the $i-th$ class, and the denominator is the sum of the exponential values of all classes iterated by the index j. The output of $softmax$ function approximates a probability distribution that sum to 1. [7]

3.2.1 Universal approximation theorem

Capabilities of neural networks lies in Universal approximation theorem, which states that a neural network with a single hidden layer containing a sufficient number of neurons can approximate any continuous function to an arbitrary degree of accuracy. [8]

3.2.2 Loss functions

Neural networks are trained in classification or regression tasks to minimize given loss functions on a given dataset. The most common loss functions for regression tasks are MSE or RMSE. Both functions are very similar, however RMSE yields the output in the same unit as in the data. The formulas for these are:

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_i (y_i - \hat{y}_i)^2 \tag{3.5}$$

$$RMSE(y, \hat{y}) = \sqrt{\frac{1}{n} \sum_i (y_i - \hat{y}_i)^2} \tag{3.6}$$

where y_i is the true value and $\hat{y}_i$ is the predicted value for the $i - th$ sample, and n is the total number of samples.

The most common loss function for classification problems is cross-entropy loss. The formula for cross-entropy loss is:

$$CE(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i) \tag{3.7}$$

where y_i is a true class label for the $i - th$ sample, and $\hat{y}_i$ is predicted probability for the $i - th$ sample. [7]

3.2.3 Gradient descend

Basic learning algorithm of neural networks is gradient descend. It iteratively takes steps in the direction of the steepest descend of gradient. Gradient is computed as a vector of the partial derivatives of a function with respect to its input variables, particularly for neural networks, the loss function with respect to the model parameters. [7]

3.2.4 Vanishing gradient

Deep neural networks introduced some difficulties with gradient descend algorithm. As the gradient flows through the network, in each following layer, the gradient is multiplied by value of the derivation of an activation function. For instance, the maximum derivation value of *sigmoid* function is 0.25 at the point of 0, so in $n - th$ layer the gradient only from activation function is maximum of 0.25^n. When the gradient is too small, it does not allow for the model parameters to be trained optimally, thus it is called vanishing gradient problem.

Similar problem to vanishing gradient problem is dying ReLU problem, where positive values have derivative of 1 and negative values 0, so in every following layer with negative input to ReLU, the value is multiplied by 0, thus resulting in zero gradient. [9]

3.2.5 Residual neural networks

To bypass the vanishing gradient problem in deep neural networks, residual neural networks introduce a skip connection feature. In classical neural network architecture, input is fed forward through the network layer by layer. The skip connection allows for a copy of the input to take the path directly to the next layer, while the other part skips one or multiple layers unchanged. This allows the neural network to learn incremental changes without losing previous information. Residual neural networks are capable of increasing their accuracy even with high number of layers, since the gradient can flow through the network by skip connections.

The most famous representatives are ResNet18, ResNet50, ResNet101, and ResNet152, where the number after ResNet indicates the number of layers. [9]

3.2.6 Convolution and GPU utilization

To accelerate the pattern recognition element in neural networks, trainable convolutional filters were added to improve the model's capabilities. Convolution is a mathematical operation that involves moving a sliding window, called kernel, through an array. The output is the sum of all inputs multiplied by the weights in the kernel.

During both the forward and backward pass in neural networks, many operations can be easily parallelized, allowing for efficient utilization of GPUs. Example of one such operation is above mentioned convolution. In fact, the entire forward and backward pass can be transformed into matrix multiplication, which is well-suited for GPU processing. [7]

3.2.7 Dual head networks

One useful advantage of neural networks is that a single model can be trained to perform multiple tasks while sharing common knowledge. These are called dual-head networks, and they are mostly used with convolutional layers. The convolutional layers, which are also known as the feature extraction part, are the same for both heads. The output is then split using fully connected layers, with each head being trained using a different loss function. [5][6]

3.3 Reinforcement learning

All machine learning models are trained to minimize the given loss funtion on a given data. Based on the way how the training data are being obtained, we can distinguish supervised and reinforcement learning. In supervised learning, the model is trained using a pre-labeled dataset. In contrast, reinforcement learning generates data on the fly by allowing the agent to interact with its environment and receive positive or negative feedback, often referred to as reward or punishment. The agent is allowed to make actions and increase the probability of making successful actions, while decreasing the probability of making unsuccessful actions. [3][7]

Chapter 4
Survey

This chapter provides a brief overview of other work related to this thesis. Section 4.1 differentiates between classical and neural network-based engine architectures. Sections 4.2, 4.3, and 4.3 briefly describe three deepmind projects - AlphaGo, AlphaGoZero, and AlphaZero, while Section 4.5 studies the supervised variant of AlphaZero for matching human moves called Maia. Additionally, in section 4.6, additional examples of solutions related to this thesis are described.

4.1 Approaches to chess AI

There are several typical approaches to creating strong and powerful chess engine. A basic AI principle that is shared by most of engines is state space search, one most common representative is MiniMax algorithm. In theory, MiniMax algorithm represents deterministic approach to play the game of chess by constructing a complete game tree, evaluating terminal nodes using rules about the end of the game and backpropagating the results to the root node. If there is a winning combination, the engine will choose it and win the game. This algorithm could be implemented using a few lines of code and would be the strongest engine possible.

However practically speaking, it is not that simple. The state space is just too huge. With current level of hardware it is impossible to find a winning combination and it is expected to stay so also in the future. Therefore, additional tools are required in order to create a competitive and strong chess engine. The main purpose of these tools are two-fold. Firstly to provide heuristics for a board evaluation in non-terminal nodes. And secondly for pruning of the state space. As these heuristics are different in all game engines, we can differentiate between multiple approaches to chess AI.

Even though almost all chess AI algorithms are based on state space search, the term state space search-based algorithm is understood to mean that the power of these algorithms is based on using brute force and high computational effort to search as many nodes as possible.

Another common approach is to use neural networks to replace the heuristics for pruning and non-leaf node evaluation and make a more accurate estimation of the next move and game position outcome. Although these engines use the principle of state space search, the constructed game tree is significantly smaller. However similar performance is achieved via using more accurate heuristics. In this thesis, these algorithms will be referred as algorithms based on neural networks to differentiate them from those that search significantly more nodes in the state space.

4.1.1 Classical approach - algorithms based on state space search

Those are classical algorithms that were first to outperform human masters. In 1996 DeepBlue by IBM was able to beat human grandmaster Garry Kasparov and it was huge step for chess AI [1]. In order to achieve superhuman performance, high amount of computational effort is needed, thus these engines got better with hardware upgrade over time.

Power of engines of this type is to search as many nodes as possible in shortest amount of time. In order to make a compatible engine based on this principle, high level knowledge of effective algorithm implementation is needed as well as the human-experts knowledge for heuristics. One strongest representative of this approach is Stockfish, which is considered as the strongest chess engine over multiple years. [10]

4.1.2 DeepBlue project

Following text provides a short description to DeepBlue engine architecture.

"Deep Blue was a chess-playing expert system run on a unique purpose-built IBM supercomputer. It was the first computer to win a game, and the first to win a match, against a reigning world champion under regular time controls...

Deep Blue's evaluation function was initially written in a generalized form, with many to-be-determined parameters (e.g., how important is a safe king position compared to a space advantage in the center, etc.)...

The system combines its searching ability of 200 million chess positions per second with summary information in the extended book to select opening moves." [1]

Although the DeepBlue may be the most famous computer chess program using classical principles, it is not the strongest game engine anymore.

4.1.3 Stockfish chess engine

4.1.3.1 Basic info

Stockfish is a free and open-source chess engine that implements the alpha-beta search algorithm. The engine is implemented very effectively in the C++ programming language and contains a lot of heuristics and algorithm optimizations. In order to use hardware most efficiently, Stockfish uses bitboards as a chess board representation, which have low memory usage and allow for quick piece movement using bitwise operations. Stockfish can use up to 1024 CPU threads and 32 TB of memory for transposition tables. It can be used in chess software or in other programming languages through the Universal Chess Interface. [10][11]

4.1.3.2 Competition results

"Stockfish has consistently ranked first or near the top of most chess-engine rating lists and, as of February 2023, is the strongest CPU chess engine in the world. Its estimated Elo rating is over 3500. It has won the Top Chess Engine Championship 13 times." [10]

4.1.3.3 Adding a neural network

In July 2020, Stockfish was enhanced with a neural network called NNUE. NNUE is a small and fast convolutional neural network used to evaluate the board position. Evaluating positions using NNUE decreased the number of searched nodes from 100 million positions per second to 50 million positions per second. Despite this fact, it increased the overall Elo rating of Stockfish by 100 points. Initially, Stockfish NNUE was only used for evaluating even board positions, but later it was fully accepted. Currently, it is possible to run Stockfish in configurations with or without NNUE. [11][12]

4.1.3.4 Summary

Although Stockfish started using neural networks recently, it is not the key-part of the engine, thus is still considered a classical-approach engine searching as many nodes as possible. Stockfish was the strongest chess engine even without neural networks and even today is still very powerful when used without them. Adding neural networks to Stockfish provided an additional boost in performance. However, it is not something that should be considered as a key-corner stone. Power of the Stockfish engine still lies in searching as many positions as possible. Stockfish can search about 100 millions positions in one second on one core of modern CPU.[10][11][12] For comparison my naive implementation of a chess engine of classical type in C++ programming language from BI-PA2 course searches about 1 million positions in second. Thus implementation of an engine of that type would require skills from different field and as such its focus would not be rooted primarily in data-science.

4.1.4 Algorithms based on neural networks

Engines of this type aim to achieve higher performance not by expanding as many nodes as possible, but by searching only those moves that are more likely to occur in real gameplay and by better estimating the outcome of the game from a given board position. For these engines, human-expert heuristics are replaced with neural networks.

Due to the fact that the forward pass of a neural network is highly computationally demanding, the number of nodes that can be searched is much lower than with hand-made heuristics. Despite these limitations the performance of these engines can be similar or even higher than classical algorithms based on state space search that are capable of searching an order of magnitude more board positions. These engines use neural networks primarily for board evaluation and pruning the game tree and can be trained using various approaches.

To create a competitive engine based on this principle, high-level knowledge from the machine learning domain is necessary. One of the strongest representatives of engines based on neural networks is AlphaZero [6], which was capable of surpassing the strongest engine at that time, Stockfish.

4.2 AlphaGo

4.2.1 Why to study AlphaGo

Even though this thesis is focused on the game of chess, AlphaGo is very important project to study, because it has been the first successful program for board games utilizing neural networks to outperform previous programs. It is an ancestor of AlphaGoZero program, which is ancestor of AlphaZero program. AlphaZero outperformed all previous chess programs that were using classical approach and laid a groundwork to neural networks based approach for chess engines. [4][5][6]

4.2.2 Introduction

The following text represents an introduction to the AlphaGo project:

"The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of stateof-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% *winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0."* [4]

4.2.3 Approach details

Monte Carlo tree search uses rollouts to estimate a node's value. These rollouts can be random, resulting in a vast search, or the breadth of the search may be reduced by sampling actions from a probability distribution over possible moves, which is determined by a policy function $p(a|s)$ that takes the current position s as input. In the past, these policies were trained on human data using a linear combination of input features. However, the success of AlphaGo lies in using deep convolutional neural networks to more accurately estimate the value of a node (value network) and the policy distribution (policy network). The final evaluation combines the rollouts with the output of the value network.

The input to the neural networks is not the raw encoded board, but includes some precalculated expertly designed features.

The neural networks are first trained on human games using supervised learning and then fine-tuned using reinforcement learning on a self-play, where the target variable Z is assigned a value of -1 in case of a loss and 1 in case of a win. Draw is not possible in the game of Go.

The AlphaGo approach uses quicker and less accurate policy networks to estimate the probabilities of the next moves, as well as slower but more accurate policies to train an optimal policy that is balanced between efficiency and accuracy. The authors faced the problem of how to balance the optimal policy network to quickly prune the state-space, but not miss any valuable options of gameplay and balance the exploration vs exploitation problem.

The final performance of AlphaGo was astonishing, and it outperformed all previous computer programs as well as human masters in the full-sized game of Go, a feat previously thought to be at least a decade away. [4]

4.3 AlphaGoZero

AlphaGoZero was the next project from Deep-mind team, program for playing Go without any supervision winning 100–0 against the previously strongest program and its ancestor - AlphaGo. [5]

4.3.1 Differences from AlphaGo

These are the several important differences from previous program AlphaGo:

"Firstly it is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it uses only the black and white stones from the board as input features. Third, it uses a single neural network, rather than separate policy and value networks. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts." [5]

4.3.2 Dual head ResNet

The authors of AlphaGoZero project experimented with multiple neural network architectures. The authors compared pure convolutional neural networks with residual neural networks as well as separate neural networks for policy and value to dual head network.

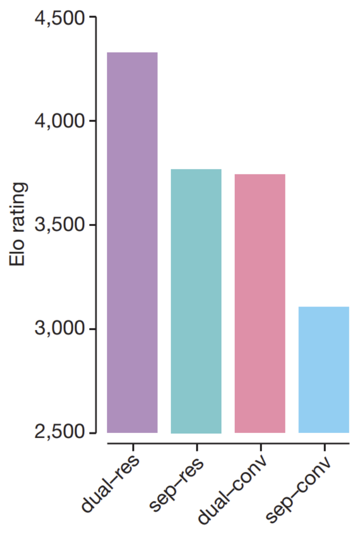


Figure 4.1 Performance of different net architectures [5]

As shown in the figure 4.1, best results were obtained using residual dual head network architecture. [5]

4.3.3 Elo rating of raw policy network

The authors compared multiple configuration of AlphaGoZero programs as well as other programs and one of configuration was single forward pass of policy network. Remarkably, even this single forward pass of neural network was able to play around Elo of 3000. [5]

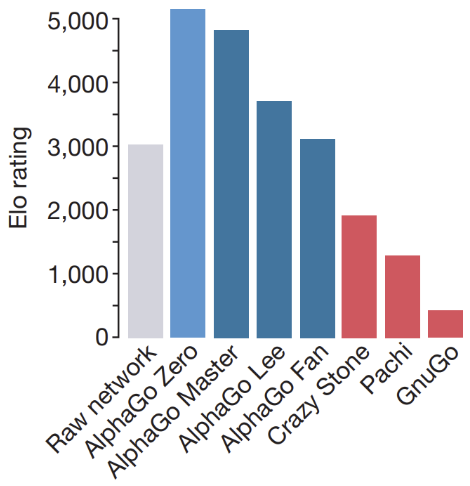


Figure 4.2 Performance of raw policy network [5]

4.3.4 Conclusion

Pure reinforcement learning approach requires more training time and achieves much better performance, compared to training on human expert data.

"AlphaGo Zero discovered a remarkable level of Go knowledge during its self-play training process. This included not only fundamental elements of human Go knowledge, but also non-standard strategies beyond the scope of traditional Go knowledge." [5]

Fact that AlphaGoZero outperformed previous version AlphaGo and was able to found new non-standard strategies implies that any kind of supervision may hinder final engine performance even though it has significantly faster learning process.

Another important aspect to take on to this thesis is that even raw output of policy network's forward pass without any search is capable of playing the game of Go at Elo of 3000. This fact is closely related to MiniMax algorithm and single forward pass problem. [5]

4.4 AlphaZero

4.4.1 Introduction

AlphaZero is DeepMind's third project focused on developing AI systems for two-player board games. This neural-network-based approach not only enabled generalization of heuristic functions in game engines, but it also demonstrated the ability to extend its decision-making process to three board games - Go, Chess, and Shogi. It was capable of defeating the world's top computer champions: Stockfish in chess, Elmo in shogi, and AlphaGoZero in Go. Remarkably, AlphaZero accomplished this with no prior knowledge of the games other than rules. Building on the architecture of AlphaGo and AlphaGoZero, AlphaZero is a neural network-based system that is trained entirely through self-play using reinforcement learning principles. [6]

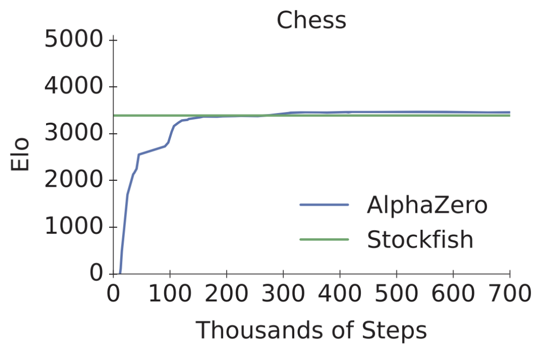


Figure 4.3 Training process of AlphaZero in chess [6]

4.4.2 Comparison with engines based on classical approach

AlphaZero replaces the handcrafted and domain-specific heuristic functions used in traditional engines with residual neural networks, and a general-purpose tree search algorithm.

"Instead of a handcrafted evaluation function and move-ordering heuristics, AlphaZero uses a deep neural network $(\boldsymbol{p}, v) = f_\theta(s)$ *with parameters* θ*. This neural network* $f_\theta(s)$ *takes the board position* s *as an input and outputs a vector of move probabilities* $\boldsymbol{p}$ *with components* $p_a = Pr(a|s)$ *for each action* a *and a scalar value* v *estimating the expected outcome* z *of the game from position* s*"* [6]

4.4.3 Technical details

AlphaZero utilizes a dual head residual network for two tasks - policy head for evaluation the chess board position, value head for predicting the following move - similar architecture to AlphaGoZero. While the policy head was trained using cross entropy loss, the value head was trained using mean squared error.

AlphaZero, like its ancestors, still relies on Monte Carlo tree search. However, since AlphaGoZero, it no longer uses rollouts [5].

Despite the belief that the classical alpha-beta search was superior to Monte Carlo tree search, approach using deep neural networks has brought previously unseen benefits.

Deep neural networks are computationally demanding, and Monte Carlo tree search allows for easier parallelization, making it more scalable.

Additionally, the classical alpha-beta approach is sensitive to extreme values, meaning a small mistake can break whole decision-making process. To avoid this, heuristic functions in MiniMax algorithm must be accurate and solid. Monte Carlo tree search, on the other hand, focuses more on mean values rather than minimum and maximum values, making it more suitable for vast tree expansion necessary to compete with Stockfish.

While Monte Carlo tree search may be a superior method for maximizing engine performance when using deep neural networks given sufficient computational resources, simpler methods closer to the MiniMax algorithm may be acceptable for smaller trees, which is the case used in this thesis. It is also closely aligned with the goals of studying MiniMax and neural network output connection. [6]

4.4.4 Performance and heuristics accuracy

AlphaZero exhibited a more human-like playing style, utilizing various strategies different from classical computer engines. Remarkably, AlphaZero expanded 1000 times fewer nodes in the game tree compared to classical algorithms, yet achieved similar performance. This suggests that deep neural networks can provide more accurate heuristic functions similarly to human masters, who can explore even fewer nodes than computer engines while achieving comparable performance. [6]

4.5 Maia chess

Maia chess is a project that is closely aligned with the goals of this thesis, therefore this section includes more detailed information paraphrased directly from Maia chess paper - [13]

4.5.1 Introduction

Maia chess is an open-source project creating a supervised machine learning chess engine. The main goal of Maia chess project is not to make the strongest engine possible, but to create an engine, that will imitate playing style of humans. Maia is a customized version of AlphaZero, which is trained on human games with the goal of playing the most human-like moves, instead of being trained on self-play games with the goal of playing the optimal moves. Training was done on a dataset of games obtained from the open-source online chess platform Li-chess. The goals are two-fold. Main goal is to predict the next move of a human opponent. Secondary goal is to predict a probability of a human player to make an significant mistake called blunder. In order to match higher amount of moves typical for humans, Maia chess does not use any tree search for the next move prediction and output is taken directly from a policy network, which better reflects human intuition and playing style. [13]

4.5.2 Maia specific architecture

The authors made further architectural changes to maximize predictive performance, including incorporating the past history of the game. In their approach, the open-source version of AlphaZero deep neural network framework called Leela Chess Zero was adopted and modified to serve their goals. [13]

4.5.3 Main goals of Maia project

Maia developers took a dual approach. First and foremost, they aimed to be able to predict the decisions people made during the course of a game. This stood in contrast with mainstream research in computer chess, where the goal was to algorithmically play moves that were most likely to lead to victory. Furthermore, following their motivation of producing AI systems that could align their behavior to humans at many different levels of skill, they aimed to be able to accurately predict moves made by players from a wide variety of skill levels. Secondly, they aimed to be able to predict when chess players would make a significant mistake. [13]

4.5.4 Why to study Maia

Maia Chess is an interesting project in connection to this thesis in multiple ways. First of all, it is a supervised chess AI method, which is very rare to find, because most of chess engines are focused towards maximal performance. For that purpose any kind of supervised learning is unsuitable, due to the facts that, human plays contain sub-optimal moves and a lot of mistakes and reinforcement learning may come up with brand new strategies and creates more optimal heuristics.

Firstly, this thesis is also focused on supervised learning in chess AI and thus it is important to study various approaches.

Secondly, it is a unique project in the context of chess AI because it does not utilize any tree search to match human-like moves. It brings interesting data about the performance of a pure model without any tree-search algorithm. Such data are very rare, since most engines are focused on maximizing performance and thus always utilize some kind of tree search. One of the primary objectives of this thesis is to examine the relationship between the MiniMax algorithm and output of a single forward pass. This has led me to study the Maia Chess project, which is closely aligned with my research goals.

4.5.5 How to restrict chess engine

Chess engines can be restricted in multiple ways to match a certain Elo. One way is to limit the computational time by lowering the depth of the game tree. Although overall Elo can be adjusted with this parameter, there was still missing a method to align the performance of engine to not only match an Elo of lower rated players, but also to match their playing style.

4.5.6 Previous solutions

There wasn't any strong predictor of human moves scalable for a given Elo range. One candidate was the Stockfish chess engine, but it doesn't predict human moves specifically in a given Elo range because the depth hyperparameter of the game tree doesn't help with scaling prediction for weaker/stronger players. Even for weak players, the best match is met with a high depth of the game tree. 4.4 [13]

Then the authors found that although Leela Chess Zero matched human moves with greater accuracy than depth-limited Stockfish, its accuracy was relatively constant across the range of player skill levels, implying that any one version of Leela wasn't specifically targeting and modeling a given skill level. Thus, using AlphaZero-style deep reinforcement learning did not match human moves specifically for a given Elo range either. 4.4 [13]

4.5.7 Greatest achievements of Maia project

The authors found out, first of all that Maia achieved much higher move prediction accuracy than either Stockfish or Leela. But Maia also displayed a type of behavior that was qualitatively different from these traditional chess engines: it had a natural parametrization under which it could be targeted to predict human moves at a particular skill level. Specifically, they showed how to train nine separate Maia models, one for each bin of games played by humans of a fixed (discretized) skill level, and they found that the move prediction performance was strikingly unimodal, with each Maia model peaking in performance near the rating bin it was trained on 4.4. This was fundamentally different from the parametrized behavior of either standard chess engines like Stockfish or neural network engines like Leela: by developing methods attuned to the task of modeling granular human decisions, they could achieve high levels of accuracy at this problem, and could target to a specific human skill level. [13]

4.5.8 Predicting mistakes

In their second main focus for the paper, the authors turned to predicting whether human players would make a significant mistake on the next move, often called a blunder. For this, the authors designed a custom deep residual neural network architecture and trained on the same data. The authors found that this network significantly outperformed competitive baselines at predicting whether humans would err. [13]

Taken together, their results suggested that there was substantial promise in designing artificial intelligence systems with human collaboration in mind by first accurately modeling granular human decision-making. [13]

4.5.9 Training data

Dataset from open-source online chess platform Li-chess contains 12 millions games. The raw data used were all downloaded from database.lichess.org, then converting the PGNs to files was computationally expensive. Processing took about 4 days on a 160 thread 80 core server and used 2.5 TB of memory, and converting to the final format for model another 3 days. First few moves of a game were excluded. All bullet and hyper-bullet games, because these data contained a lot of mistakes. [13]

4.5.10 Results

4.5.10.1 Move matching performance

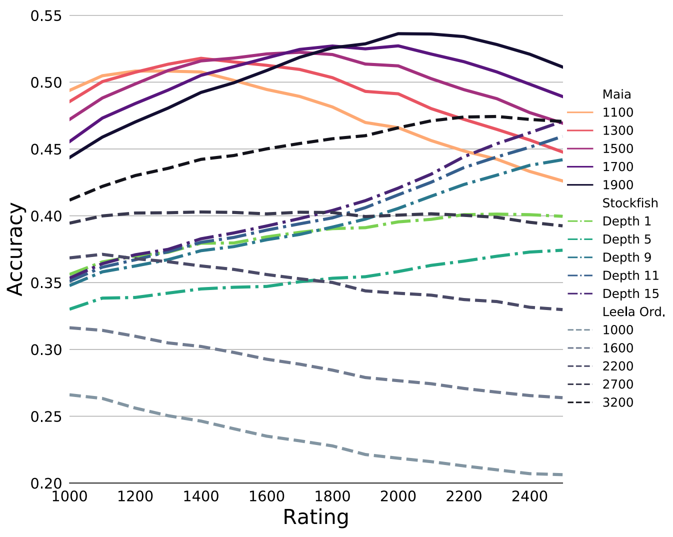


Figure 4.4 Move matching results [14]

In Figure 4.4, the authors present the performance of their Maia models on move-matching test sets. They observe that the Maia models achieve high accuracy levels, which are much better than the state-of-the-art chess engines. The lowest accuracy achieved by Maia is 46%, which is as good as the best performance achieved by any Stockfish or Leela model on any human skill level they tested. The highest accuracy achieved by Maia is over 52%. [13]

Another important finding is that the predictive accuracy of each Maia model is unimodal, with each model maximizing its performance on a test rating range close to the rating range it was trained on. This means that each Maia model captures how players at a specific skill level play and displays the skill-level-targeting behavior that the authors aim to achieve. The predictive performance of each model smoothly degrades as the test rating deviates further from the rating range it was trained on, which indicates that the model is specific to a particular skill level. [13]

To achieve this behavior, the authors made two key architectural decisions. First, they did not conduct any tree search to make a move prediction. Although Monte Carlo tree search was crucial to AlphaZero's strength and playing style, it tended to degrade move prediction performance in their setting. Here 4.5, the Authors demonstrated this by comparing base Maia with a version of Maia that did 10 rollouts of tree search exploration Second, they gave Maia the previous 12 ply (6 moves for each player) that were played leading up to the given position, which significantly improved their move-matching accuracy. [13]

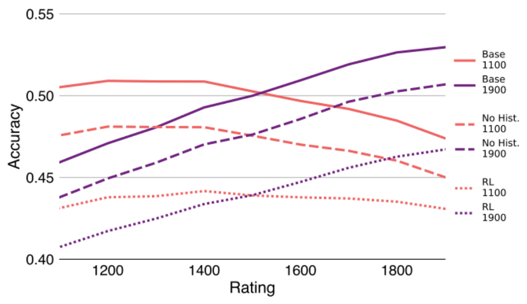


Figure 4.5 Move-matching performance of two Maia models, in either the base configuration, with no history provided, or with 10 rollouts of tree search performed. [13]

4.5.10.2 Engine performance in gameplay

The Maia team has made three of their bots available for free play on the Li-Chess platform. While the technical details of these bots have not been explicitly outlined in the accompanying paper, their performance is worth examining.

Firstly the bots have to utilize a database-driven approach for their opening moves, which allows for randomized gameplay and safeguards against commonly-used and deeply-analyzed opening strategies. Furthermore, it is important to note that the training data does not include the very first few moves of the game.

Secondly, there also has to be some solution for move randomization in the later stages of the game, single forward pass of neural network would always predict the same move, since it is deterministic process.

According to the paper, in order to achieve the highest matching accuracy of human moves, the tree search component of the algorithm was excluded, and move prediction relied solely on the policy network output. While from the paper it is not entirely clear whether the bots on Li-Chess utilize the same approach, let us assume that they do. Assuming this condition, the final performance of these models is quite impressive, but still not unbelievable. Based on comparison of an Elo rating achieved by AlphaGoZero with utilizing only the policy network output 4.2. It is worth noting that in the context of AlphaGoZero, the policy network performance was even better. However this may be due to combination of multiple facts. Firstly, AlphaGoZero was specifically trained to maximize performance. Secondly, the Elo for different games can have different scale. And lastly, games like Go may be better suited for an engine that relies simply on heuristics and intuition-driven approach rather than deep analysis.

There are some interesting observations about Elo ranges in the dataset and final performance of the model. Although the best model was trained on data from games played by 1900 Elo players, its performance is much worse than that. Currently, this bot on LiChess has an Elo of 1650 [15] in classical time play. It also should be noted that it would correspond to much lower Elo in real competition gameplay since ratings on LiChess are highly overrated compared to FIDE [16]. Another interesting observation is that the skill of players from the training data does not affect the engine's performance so much. The weakest bot trained on the 1100 Elo players' data has a final Elo of 1550 [17] compared to the 1900 training versus 1650 final score. The possible interpretation of this strange phenomenon can be found further in subsection 5.9.3.

This observation is particularly interesting in the context of this thesis, where the goal is to study the depth of the tree for the dataset generation and the ability of neural networks to fit that data. In this case, the strongest players represent a more complex evaluation and move prediction function for the neural network to approximate, corresponding to higher depths of the game tree. The fact that the performance is very similar implies that the neural network may not be capable of extracting this deep analysis, and the only benefit is that the strongest players utilize more clever heuristics represented by their intuition. [17] [15]

4.5.11 Impacts

Even though the main focus of the Maia project is to imitate human behavior in the game of chess and not to predict the move that most likely leads to a win, there are some key points that are crucial for this thesis.

First of all, the Maia approach proved that is is possible to match high percentage of human moves without any tree search using only one forward pass of a neural network. However, it can be expected that the resulting engine's performance would be severely limited. The resulting performance of the engine is on one hand really surprising for a single forward pass of a neural network, but is still quiet bellow a naive MiniMax algorithm performance. Considering that this project was carried out by a accredited team of computer scientists using a balanced and representative dataset containing hundreds of millions of positions, we can infer from the resulting performance that, for beginners, it could be a great learning tool, but a single neural network approach still won't create a chess engine for intermediate or advanced players. The MiniMax algorithm and single neural network forward pass relation is going to be studied further in this thesis and for that purpose Maia Chess has created a great information baseline.

Another important hypothesis is, that the depth of generated dataset won't affect final model so much in terms of performance as similar phenomenom happened in Maia chess project. Maybe because the ability of neural network to generalize simple cases just overtake the advantage of some more complex data, that are hart to fit by a neural network.

In terms of move-matching performance, a single neural network is capable of matching about 50% of moves made by human players. It is expected that a similar accuracy can be achieved with the output matching of other computer engines.

4.6 Other approaches

The mentioned approaches are not the only ways to implement an engine. There are many more approaches using neural networks that have some nuances from both the Stockfish and AlphaZero engines.

4.6.1 Convolutional neural networks as a supervised move prediction

An interesting supervised solution implemented in the standfor edu paper [18] is to use simpler network architectures. Here, the authors split the problem into multiple parts: predicting which piece should be moved and then its final destination.

The authors used convolutional neural networks, same as in AlphaZero, because convolution is best for pattern recognition, such as identifying king safety and pawn walls.

The model achieved an accuracy of 40 - 50% in predicting moves, depending on the piece type. [18]

4.6.2 Giraffe

Another interesting approach can be found in the Giraffe chess engine. The engine takes advantage of move probability evaluation, which limits computational time not by searching to a given depth of the tree, but by searching to a probability level threshold. This approach allowed the Giraffe engine to find more relevant nodes that would have been missed using the classical depth-limited approach. [19]

4.6.3 Solving endgames

Using neural networks as an evaluation function for alpha-beta search can also be beneficial in endgames, where an intuition-driven approach can often outperform handcrafted knowledge. The authors created a neural network to predict the number of moves leading to a win when playing optimally, and this was then used in alpha-beta search.

The proposed method improved state-of-the-art techniques in terms of time, space, and move complexity. [20]

Chapter 5

Supervised learning

This chapter is dedicated to the supervised goals of the thesis. The designed architecture employed to solve subsequent tasks is presented in section 5.1. The generation of the dataset is laid out in section 5.2. The relationship between the output of a single forward pass of the neural network and the result of the MiniMax algorithm is analyzed in the following sections. Particularly the consequences of the depth of the generated dataset on the accuracy of various ResNet architecture models in section 5.6. Additionally, section 5.7 proposes a board complexity metric, while section 5.8 conducts and evaluates experiments on datasets with different complexities generated using the proposed metric. Finally, in section 5.10, the deployment of trained models in a chess engine is investigated.

5.1 Architecture description

5.1.1 Description of the input

There are various approaches of how to encode chess board and pass it into a neural network. In this thesis following approach was taken.

Firstly, the network needs information about position of pieces. For that purpose, the input to the neural network consists of several layers. There exist 6 types of pieces for each player in the game of chess and positions of each kind of piece is represented in its own layer. Positions of pieces within a chess board is represented by 2D array of $8 * 8$ dimensions with value 1 for present piece and 0 if this type of piece is not present. To represent all these pieces positions, a total of 12 2D layers is required. The input then passes through multiple convolutional layers, thus it is important to keep original 2D structure, because the board position is then processed in the same manner as an image.

Secondly information about the protected squares as well as squares, that are being attacked is encoded in the input. The protected squares, in this particular usage, are defined as a set of final destinations of all potential pieces movements of the player on turn. Squares, that are being attacked, in this particular usage, are defined as a set of final destinations of all potential pieces movements of the opponent player, had it been on turn. This information is represented by 2 additional 2D layers. Such an approach may help the neural network to place pieces more cleverly so as evaluate the board position more accurately since these additional layers represents area being controlled.

The perspective is switched on the opponent's turn, eliminating the need for an additional feature to indicate the current player.

5.1.2 Model description

The solution employs a dual-head neural network of a ResNet type. The feature extraction part, which consists of convolutional layers, is shared for both heads, and the output is then split using fully connected layers into two heads with different activation functions, the policy network and the value network. The policy head predicts the probabilities of next moves, while the value head evaluates the board position.

The policy head was trained using cross-entropy loss, which is better suited for classification problems. The value head was trained using MSE, which is better suited for regression problems.

5.1.3 Value head output

The board position evaluation ranges from -1 to 1, where -1 indicates that the player on turn is most likely to lose, and 1 indicates that the player on turn is about to win. In order to limit the output of the last layer to the desired range, the $TanH$ activation function was used.

5.1.4 Policy head output

The output of the policy head is a vector of 4096 numbers, which sum to 1, achieved through the use of the SoftMax activation function. The first part of the index ($index//64$) represents the square the move was made from, and the second part ($index \bmod 64$) is the destination square. There are $8*8 = 64$ chess board squares. So the final vector consists of $64*64 = 4096$ [1] positions. For example, index $0 =$ A1A1, $1 =$ A1A2... Some indices do not map to a valid move. This is due to two reasons.

Firstly some moves cannot be made from every possible board position e.g. E2E2 or A1H7.

Secondly, some moves cannot be made from a particular board position. e.g. starting position and move A1A8.

5.1.4.1 Move validity

Classifying validity of a move is not a necessary task for a neural network to learn. Although the net is very quickly trained for that - after 1 hour of supervised learning the average value on a valid move is about 2000 times higher than on an invalid move. see 5.1 for more details.

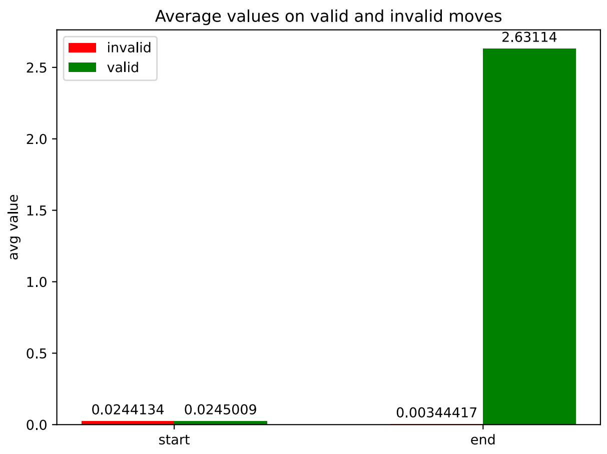


Figure 5.1 Comparison of average value on valid and invalid moves

[1] count of the source squares multiplied by count of the destination squares

But this task is simple and deterministic, so the net can be helped with that. After the model's forward pass, $SoftMax$ is applied to the vector consisting of only valid moves, which is the final taken approach.

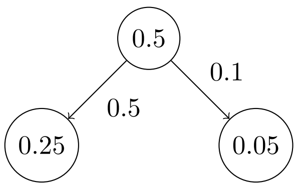
5.1.4.2 Summary

In summary, whole forward pass can be perceived as a task of classification of an 8 x 8 pixel image with 14 color profiles into 4096 categories, and an evaluation between -1 and 1.

5.1.5 Engine

This section describes how to utilize such neural network model in a chess engine for the best move selection. The engine constructs a game tree by gradual expansion of the defined amount of the most promising positions (nodes). The selection of the most promising nodes is the main task of policy network by assigning a probability value to each node.

In the first layer, the probability is direct output from the policy head, while in deeper layers, the probability of each node is multiplied by all its ancestors.[2]



At the beginning, when the model is not trained, the distribution of move probabilities is close to uniform distribution, resulting in all nodes being expanded equally layer by layer. So far this approach is similar to the use of a MiniMax algorithm without pruning. As the model becomes trained, it starts to recognize which positions are more likely to be played and prunes the others. As a consequence, with the given amount of expansion, the resulting tree is much deeper.

After given number of positions have been expanded, the leaf nodes are evaluated using the value network. Then the result is backpropagated to the root using the MiniMax algorithm.

5.2 Dataset generation

The dataset for supervised learning is generated using the Stockfish chess engine. To create the dataset, millions of different board positions must be obtained and target variables must be assigned to them. There are two main considerations in the generation of board positions: diversity and balance.

5.2.1 Board positions generation

These are important considerations when generating chess board positions.

The generation of chess board positions must adhere to two fundamental principles to ensure a comprehensive and representative dataset.

Firstly, the positions must exhibit diversity that accommodates all playing styles and different advantages and disadvantages (e.g. when white is ahead, or when black only has a knight and white still possesses two bishops).

[2]Binary heap is used to efficiently select the node with the highest likelihood.

Secondly the dataset should be balanced in terms of game stages and amount of advantage for winning player. The dataset must encompass all stages of the game, including positions from the opening, middlegame, and endgame and various winning probabilities represented in equal amounts. Positions must be representative of a real gameplay. Artificial situations, such as one player having their king on the enemy's home rank during the middlegame, should be avoided. Such absurd situation may be generated using random gameplay.

5.2.1.1 Diversity

The first rule of generating board positions can be satisfied through random move generation, but this approach is not ideal.

It results in an unbalanced dataset with a disproportionate number of endgames compared to openings and middlegames. This is caused by the fact, that the probability of being mated by a random move is just too small, thus the duration of such endgame is very prolonged with many unnecessary moves.

The second problem with using random move generation is that the resulting games are often uneven, with one player clearly winning and the other losing.

This leads to an unbalanced dataset, lacking positions with even chances of winning, and an imbalance in the distribution of game stages. As a result, this method does not provide optimal training data.

5.2.1.2 Balance

The second rule, which requires the positions in the dataset to be aligned to a real gameplay, can be met by having two strong chess engines, such as Stockfish, playing against each other. However, this approach may result in a lack of diversity in the generated positions - featuring very similar gamestyle. Thus, this method may not lead to optimal training.

5.2.1.3 Taken approach

The optimal approach to fulfill both rules of generating diverse and balanced chess board positions would be to combine the methods of random move generation and letting Stockfish play against itself. One approach is to let Stockfish play against itself with a suboptimal N-th move, where N is a hyperparameter that can be adjusted.

Another common approach is to first play a few moves from an opening database, which prioritizes more common openings over less common ones, and then transfer control to Stockfish. Although the latter method may provide a more balanced and better aligned dataset to a real-play, the former method is favored for its diversity and simplicity in implementation. This is the reason why this approach was finally adopted for purposes of this thesis.

5.2.2 Creating a target

Crucial task is proper definition of targets for each head of the neural network.

5.2.2.1 Vale head target

First objective is to evaluate the position in terms of its likelihood of winning or losing. This problem can be approached as a binary classification, via taking the advantage of the MLE principle, where the end result of a game played by Stockfish would determine the target variable as 0 for loss and 1 for win and mark this way all board positions of each player. This method requires a significant amount of training data split into huge batches to train the model effectively.

This approach is used in the reinforcement learning method, where the objective is to find an optimal function for the best performance of the engine without incorporating any heuristic information. In supervised learning, incorporating human-like heuristic information is acceptable. Hence, the definition of the target can be approached as a regression problem, where the goal is to approximate a ready-made heuristic function. This was the final approach taken, because one of the goals of this thesis is to study the connection between MiniMax algorithm and output of neural network, where depth of evaluation has to be directly present in the training data.

There are numerous heuristics that can be employed. One of the most commonly used is to assess the position based solely on a weighted material count. While this approach may yield some results, it fails to fully utilize the potential of deep neural network to approximate more complex functions and disregards the features extracted by convolutional layers. As such, a more sophisticated heuristic function, which takes into account factors such as positional advantage and king safety, is preferable.

The Stockfish chess engine includes the ability to estimate the likelihood of a win or loss based on the current position. The simplest solution, therefore, would be to utilize this ready-made functionality to attain the most accurate results possible. However, the use of this method brought certain difficulties. In a real chess game, position evaluation assumes that both players have played and will continue to play their best. This means that if one player has played his best and has lost two more pieces than his opponent, his chance of winning is almost negligible.

While the method of estimating the likelihood of a win or loss may provide the best approximation of reality, it is not the optimal target for training a model intended for use in a game engine. The game tree that traverses the state space is composed of a vast number of uneven board positions, and the model must be able to distinguish between positions that are bad and those that are even worse.

Although this approach may be useful for game analysis and estimation actual winning chances, models used in a game engines trained on data generated in this manner performed very poorly.

A possible interpretation of this fact is based on the following insight. Training the model on data generated using this method resulted in the ability to categorize positions as winning or losing, but the model was not be able to differentiate between such positions accurately. The goal is to train the model to accurately differentiate even between positions, where winning chances are almost sure. For example imagine one position with 5 pawns ahead as opposed to another position with 8 pawns ahead. The goal for the neural network should be to prefer the better position, despite the fact, that both chances of winning are already close to 100%.

When leaving this issue unsolved, the adverse effect is then further magnified. Imagine the game tree consisting of very large number of nodes with chances of winning evaluated in the range between 98 - 100%. Error rate of the trained neural network can be even 5%. Such situation can lead to almost random choice of preferable positions when MiniMax is selecting the correct branch in the tree.

Thus, better approach was taken. It can be taken advantage of the fact that Stockfish can also evaluate board position advantage in centipawn units. This metrics can be further scaled to satisfy desired range. In order for the model to distinguish between more or less winning/loosing positions, particularly skewed *TanH* function was utilized to limit the target values within the range of -1 and 1. Initially, the maximum potential advantage was computed representing a scenario where one player only has a king, while the other player possesses all of his pieces. This scenario yields to a value of 39 pawn-points advantage. Based on this, the hyperparameter SPREAD for the *TanH* function was set to 0.2, yielding the following expression converting pawn-unit evaluation into range -1,1:

$$\text{evaluation}(x) = \frac{1 - e^{-\text{SPREAD}\cdot x}}{1 + e^{-\text{SPREAD}\cdot x}}$$

This approach aims to achieve more accurate estimation as this transformation puts greater focus on balanced positions, rather than positions where one player has a clear advantage, since such positions offer limited strategic options. After employing this function, the differences in position evaluations using the $TanH$ function are as follows:

Losing a pawn	0.10
Losing a knight	0.29
Losing a rook	0.46
Losing a queen	0.72
Losing a queen and a knight	0.83
Losing a queen and a rook	0.89
Losing a queen, a rook, and a bishop	0.94
Losing a queen, a rook, a bishop, and three pawns	0.96

Table 5.1 Special TanH function for evaluation

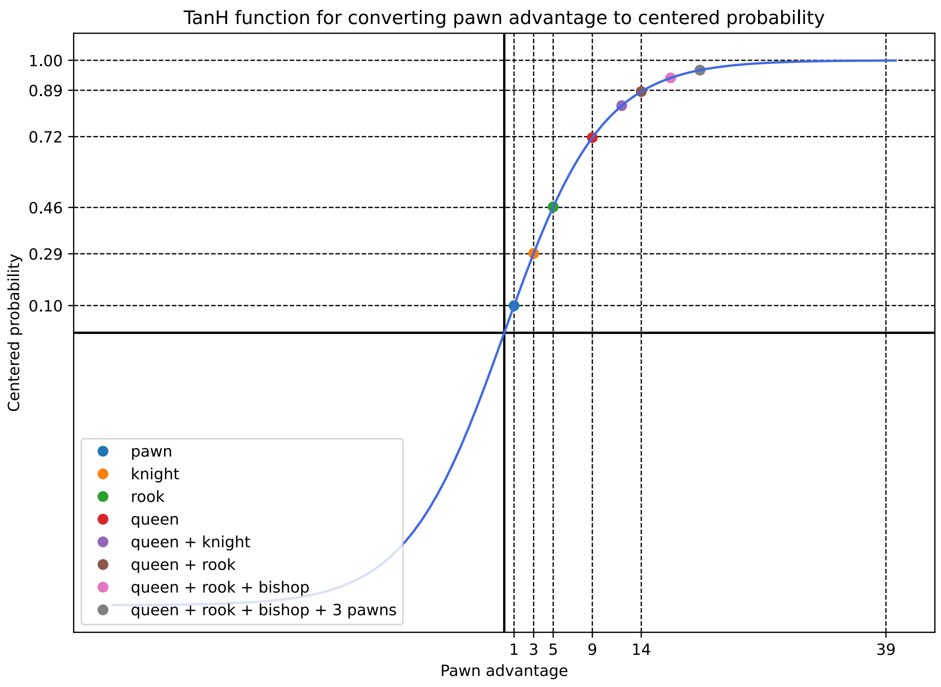


Figure 5.2 Special TanH function for board position evaluation

5.2.2.2 Policy head target

For move prediction, two different approaches for target definition were utilized.

The first strategy that was experimented with involves usage of Stockfish to suggest the single best move and to include it in the dataset with a probability of 100%. Although this method is straightforward and results in a fast generation of the dataset, it has some limitations. For example, consider the following scenario: 5.3

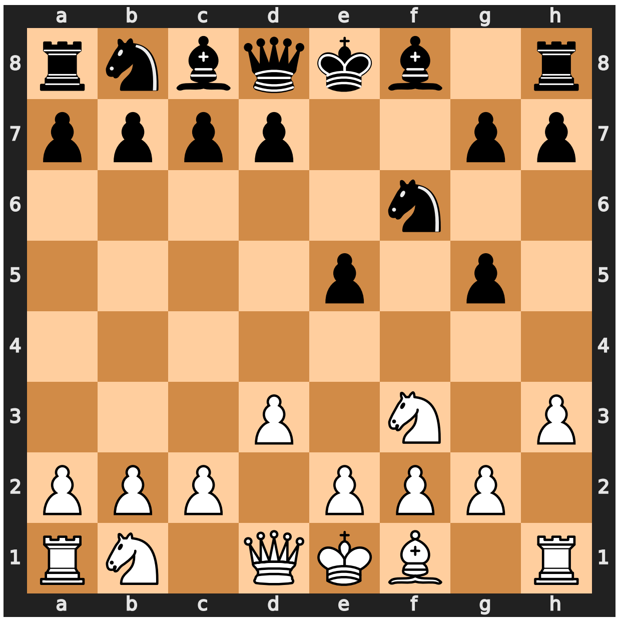


Figure 5.3 Example of two optimal moves

From this position, there are two optimal moves. Such scenario can lead to inconsistent [3] data within the dataset.

Further, in the evaluation of a board position that the model has already seen, the pruning process may be too accurate, resulting in neglect of other valuable moves. Although this is not a technical case of overfitting, as the loss on the training data can be similar to the loss on the test data, it is in a way similar issue and results in suboptimal behavior when deploying the model in a game engine, mostly on the opening boards.

The second approach takes an advantage of the fact that Stockfish can also provide position evaluation in centipawns. This output can be used to approximate the real probability distribution, where all valuable moves have some probability of being played, but better moves have higher probabilities and suboptimal moves have probabilities near to 0.

This approach evaluates the gain of every move in centipawns. 5.4b.

$$gain = \text{evaluation before move} - \text{evaluation after move} \tag{5.1}$$

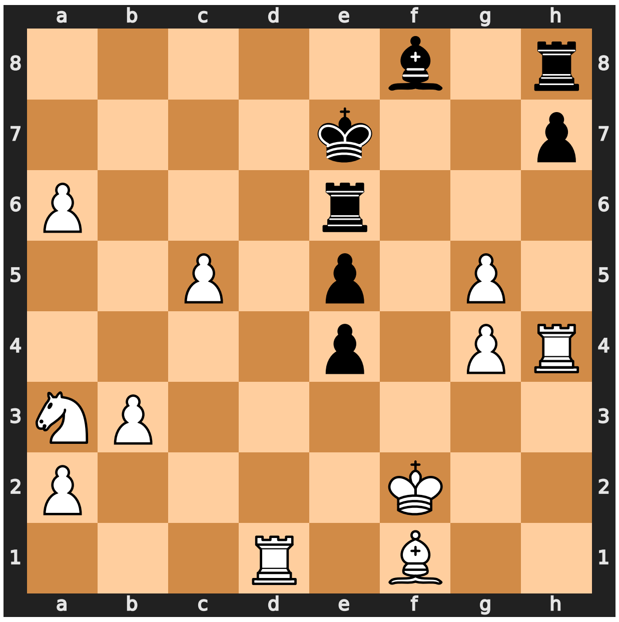
In the second step, 25% of weak outliers are removed as significantly sub-optimal. This percentage is based on practical observation that typically all the remaining moves are similarly strong among themselves when compared to the removed set. 5.5a.

The reduced data is then further transformed using MinMax scaler 5.5b resulting in nearly uniform distribution.

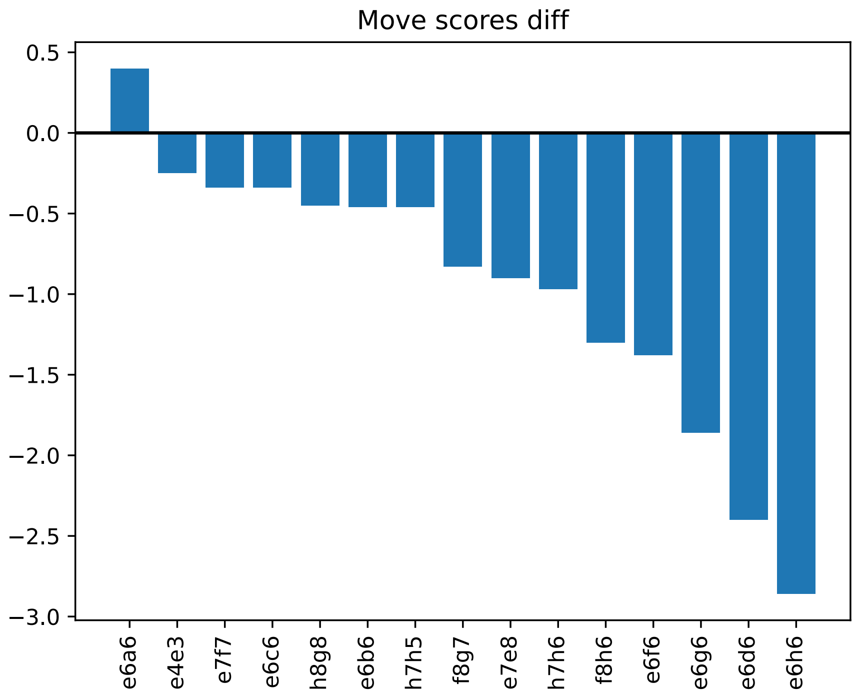
However, the goal was achieving of more exponential-like distribution. To accomplish this goal, each value was raised to the power of four 5.5c, thus creating a greater difference between high and low values.

Finally, the probabilities were then calculated by dividing each value by the sum of all values of all moves 5.6. This process generated target that closely approximate an exponential probability distribution, with a sum equal to 1. The resulting average distribution over all boards in the testing set is shown in the following figure: 5.7.

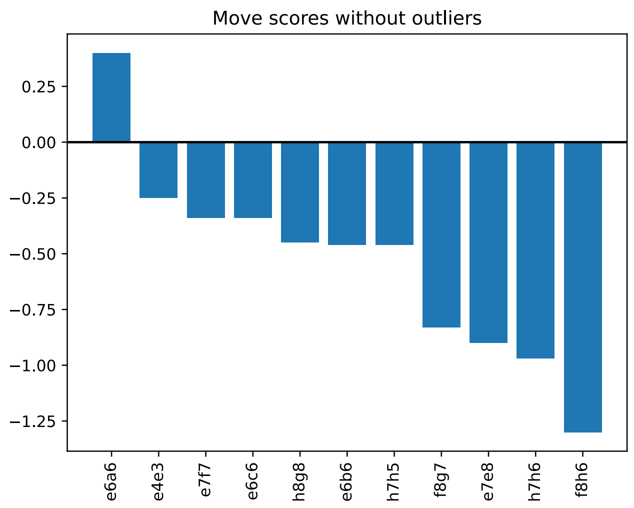
[3]identical inputs yielding different targets



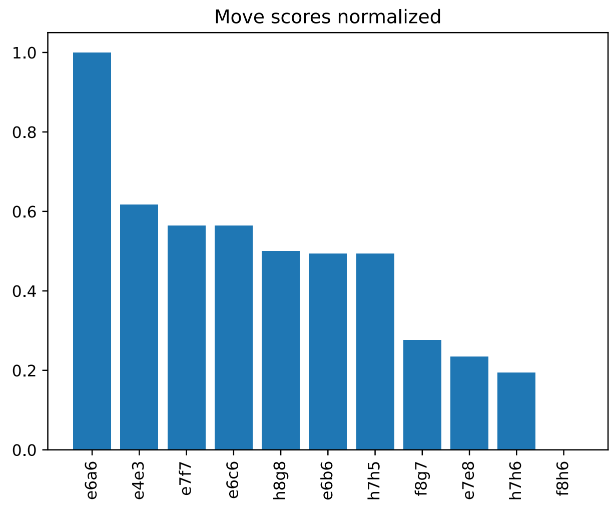
(a) Example board



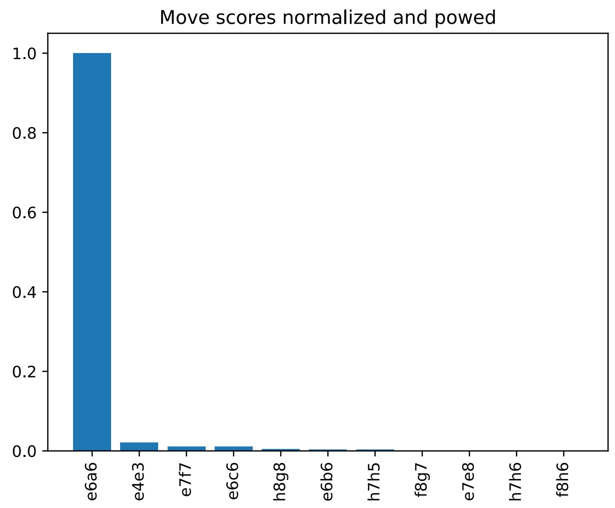
(b) Pawn-unit gain of all moves



(a) Pawn-unit gain of moves without sub-optimal outliers



(b) Normalized values



(c) Powed values

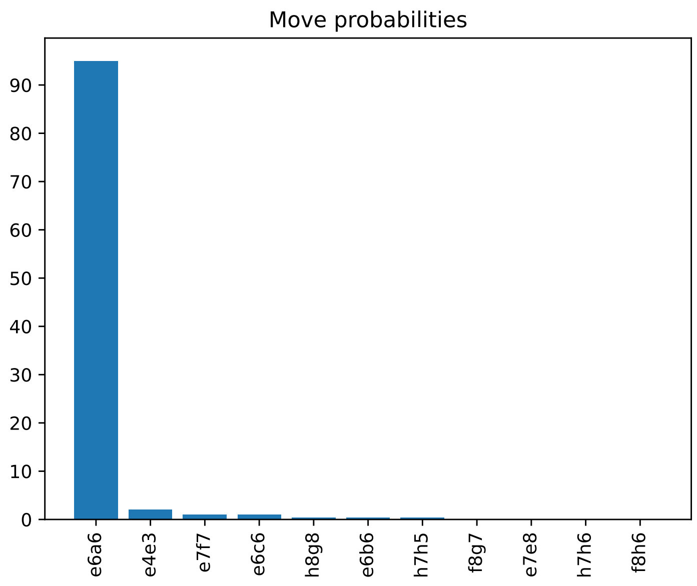


Figure 5.6 Final calculated move probabilities

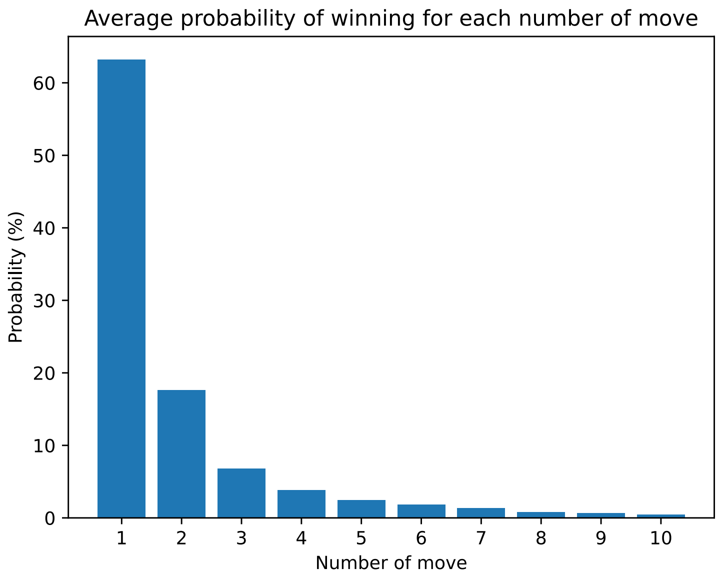


Figure 5.7 Average probabilities in the testing set

5.3 Training process

The supervised learning dataset consists of 10 million position, which were processed through a neural network in 10 thousand training steps. Each step consists of 40,960 board positions, which were fed forward to the neural network in 10 batches of 4,096 board positions. During each step, the training was conducted over three epochs, with the learning rate being updated using a geometric sequence with a hyperparameter $gamma = 0.9$. After each epoch, all of the 40,960 positions were randomly shuffled. After each step, half of these 40,960 positions were randomly swapped with random positions in the full dataset. The swapping of half of the training data addressed the issue of high errors on the testing data right after the first epoch of each next step. Finally, a small amount of L2 regularization was applied to prevent overfitting.

5.4 MiniMax algorithm and single neural forward pass - introduction

The primary objectives of neural networks in a chess engine are the evaluation of board positions and prediction of next move. However, simply counting and examining the positions of pieces on the board is inadequate for achieving accurate evaluations or selecting optimal moves. To accomplish these goals, it is necessary to simulate possible future gameplay, switching perspectives between both players, and imagining oneself in the shoes of the opposing player. In order to achieve a deeper evaluation of the likelihood of winning and the moves that are most likely to be selected, it is necessary to simulate future gameplay. Humans, for instance, switch perspectives and simulate gameplay mentally. Computers typically construct a game tree, evaluate it and backpropagate using the MiniMax algorithm, or employ strategies like Monte Carlo simulation with rollouts.

Following sections of the thesis aim to investigate whether neural networks can serve as a viable alternative to the MiniMax algorithm for evaluating board positions and predicting moves. Specifically, determine whether neural networks can achieve this objective of evaluation in consideration of simulating possible future gameplay in a single forward pass, without requiring a game tree.

5.5 Identifying challenging chess positions for CNNs to evaluate

CNNs are well-suited for pattern recognition and can replace the quick heuristics used in traditional chess engines. Using visible elements such as piece count and their distribution across the board, CNNs can extract and utilize features such as king safety, pawn structures, pawns near promotion, movable rooks, and blocked bishops. While these features are highly correlated with the likelihood of winning, evaluating a board position solely based on these features may not be sufficient compared to a deep evaluation using the game tree. While most of the positions tend to exhibit these features in a visible and clear manner, certain boards may appear similar at first glance but have significant differences in evaluation due to hidden information in the game tree.

5.5.1 Comparison to image recognition

In image recognition, similar pictures typically have similar outputs because all information is directly present in the image. However, in the evaluation of chess board positions, certain positions may hide information deeper in the game tree, making it less visible, thus harder to evaluate.

5.5.2 Example

Take this position as an example: 5.8

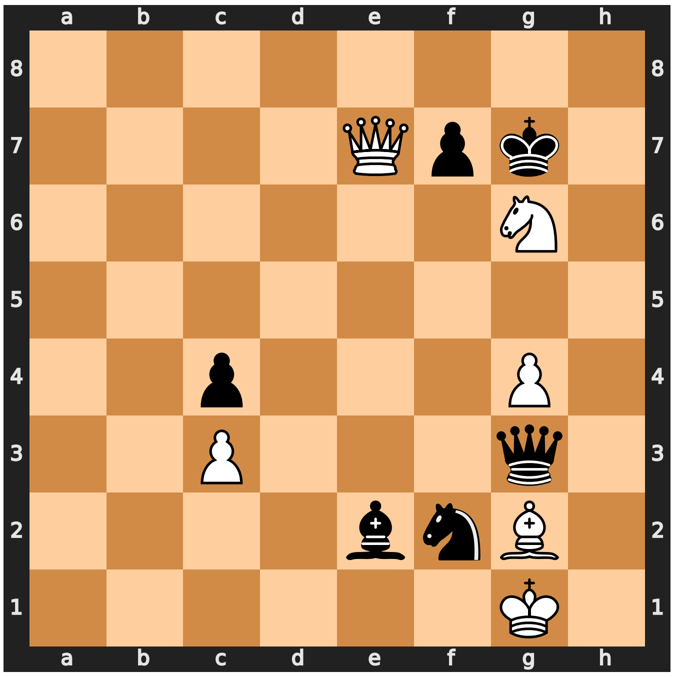
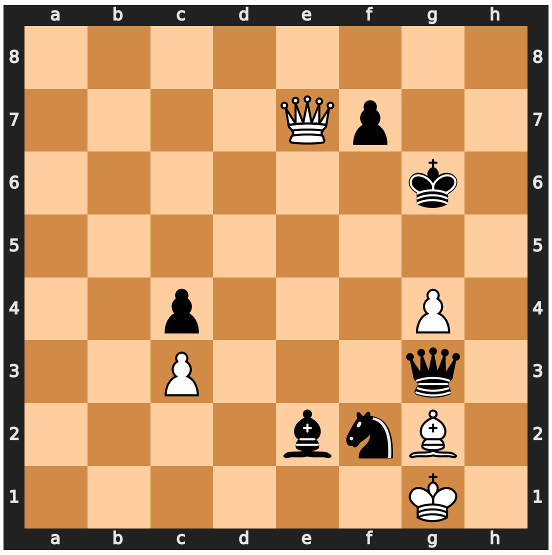


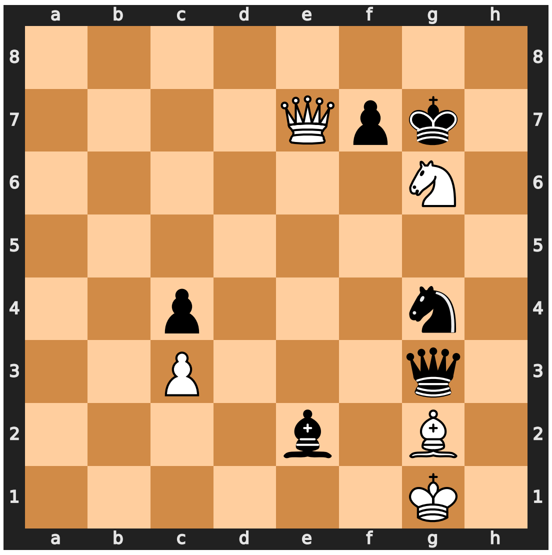
Figure 5.8 Challenging position (black on the move)

There are two possible outcomes of the game:

- Black can make Kxg6, has material advantage and ends with draw after Dg5+
- Alternatively, black can play Jxg4 and has a high chance of winning



(a) Draw



(b) Win

Figure 5.9 Possible outcomes

"This move allows for a quick draw as white has an unexpected motive, namely to force a stalemate by playing Dg5+ and the black king cannot escape the checks [a] *and Kxg5 is an immediate draw."* [21]

[a]The white queen switches among the squares h5, g5 and d8, and no matter where the black king goes, perpetual check occurs.

"After capturing the bishop and Kxg6, the position, although seemingly rather equal, is actually winning for black. The central reason is the difference between the position of the two kings. White does not have a perpetual check and the white bishop, in contrast to the black knight is rather passive and condemned to play the role of a pawn in front the king. With the queens on the board the difference between the functionality of the two minor pieces is decisive." [21]

Although these two positions can look very similar from the pixel-image perspective and possible extracted features [4], there is significant difference in likelihood of winning. The key difference is hidden in the game tree.

The aim of this section is to study such examples and determine whether neural networks can extract features and patterns to identify hidden information from the game tree and to use it for move prediction and board evaluation. [5]

[4]similar pieces count, similar positions, similar space controlled, no pieces is under attack
[5]It is actually very similar task.

5.6 Connection between ResNet type and depth of generated dataset

In case CNNs were able to approximate the result of MiniMax algorithm, then fitting evaluations yield by MiniMax run to higher depths would correspond to approximating more complex functions. As a result, there might be a connection between the depth of the generated dataset and the complexity of the model used.

In order to verify this hypothesis, several ResNet architectures were examined, in particular ResNet18, ResNet50 and ResNet152, with depths of 6 and 12 ply [6]. Two primary metrics were utilized to determine the model's ability to fit the data.

First metric - best move matching accuracy in (%) - further called as match best

Second metric - MSE of board evaluations.

Each type of ResNet was subjected to training through 10,000 training steps for both depths. Since there are frequently more equal moves, slight changes in the model's parameters can significantly impact matching the best moves in the testing set. Therefore, to improve readability, the results were averaged over 1000 training steps.

Finally, the results are as follows: 5.10

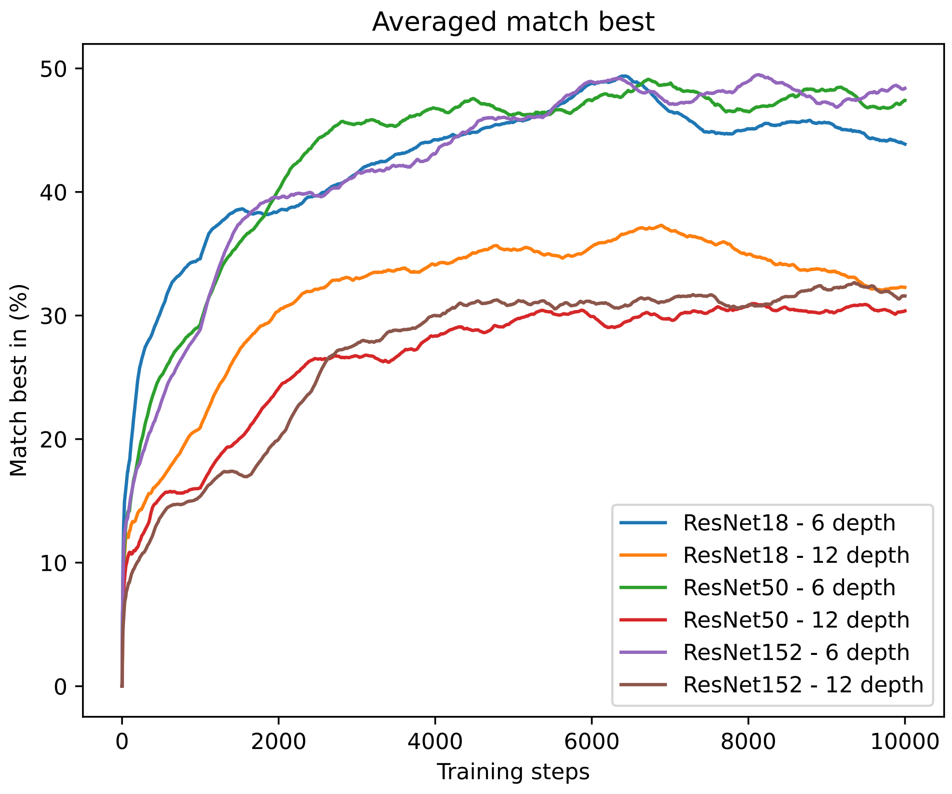


Figure 5.10 Move matching results

To begin with, the ResNet types show no significant differences from one another. On the same dataset, ResNet18 can fit the data similarly to ResNet152. The only difference is that ResNet18

[6]half moves

learns faster from the beginning. This is expected since ResNet18 is a much less complex model. However, there is a significant gap between the depths, regardless of the complexity of the model applied to them.

Depth 6 exhibits an average matching rate of around 50% over 1000 steps and a maximum of 60% in a single evaluation on non-averaged data. In contrast, depth 12 is only matched at around 30% on average over 1000 steps, with a maximum of 40% in a single evaluation on non-averaged data.

These are the results for board position evaluation: 5.11

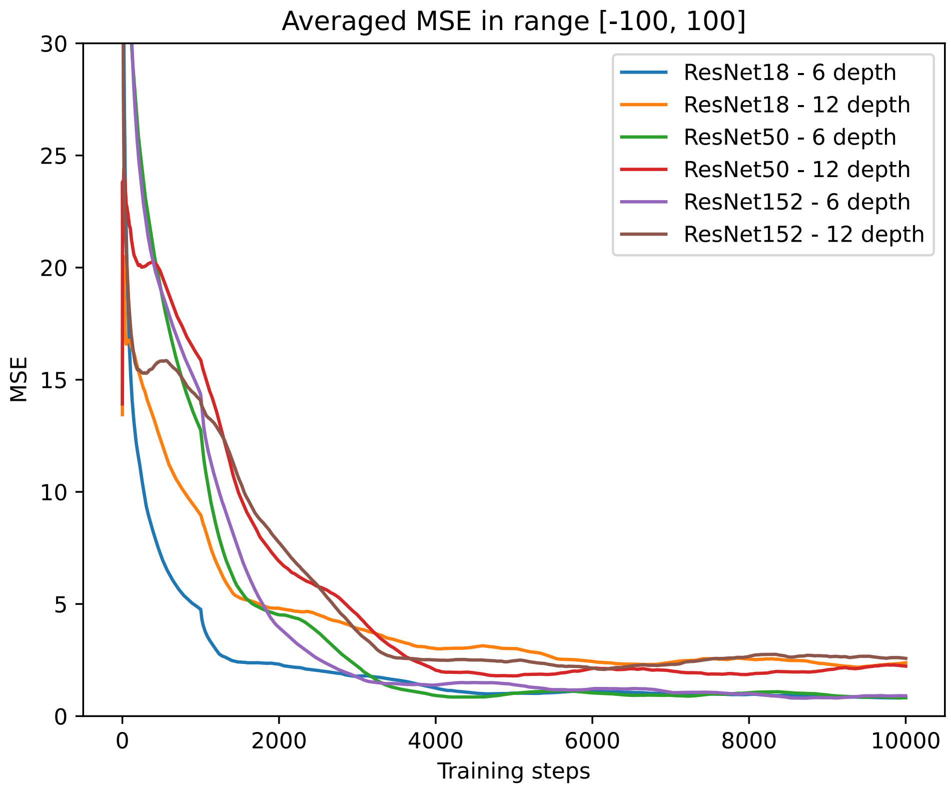


Figure 5.11 Position evaluation results

A similar pattern can be seen in the MSE measurement, where there is almost no difference between the models complexities, but a significant difference between the depths.

5.6.1 Depths differences

It should be noted that although there is a significant gap between the measurements of depths, it does not necessarily imply that CNNs can approximate the result of the MiniMax algorithm on best move selection. There are two potential reasons (or their combinations) for the observed results.

Firstly, CNNs can approximate the MiniMax algorithm to a certain extent; as all the positions evaluated at depth 6 became more complex with depth increased to 12, CNNs were still able to identify the optimal move at the depth of 6. However, at depth 12, the complexity of the game tree increases to a level where CNNs are no longer able to approximate the algorithm accurately.

Secondly, CNNs may not be able to approximate the MiniMax algorithm at all. Some simple visual patterns naturally corresponds to the best move selected by searching the game tree up to depth 6. However, as the depth of the game tree increases to 12, a new potential option of a future gameplay may be discovered in the game tree, causing the simple pattern to lose its original importance. Nevertheless, CNNs cannot extract this hidden pattern in the game tree, leading to a misclassification of the best move.

5.6.2 Testing less deeper trained ResNet on deeper data

To gain more insight into which explanation to support, very simple method for testing has been proposed. All the models trained on depth 6 were tested on test sets of depth 12. This was done to determine whether the model could extract and generalize complex information from the training data to outperform the models that lacked such information.

If deeper-trained models achieved higher performance than less-deep-trained models, it could suggest that CNNs have the ability to extract, generalize, and utilize information that is hidden in the game tree.

On the other hand, if the results revealed that there is no significant difference between models trained on the data with different depths, this may imply that CNNs could only understand positions where the best move matched some kind of simple patterns and do not benefit from deeper and less visible information in the training data. As a result this would imply that such information is too complex to generalize.

The outcome of this testing is as follows: 5.12

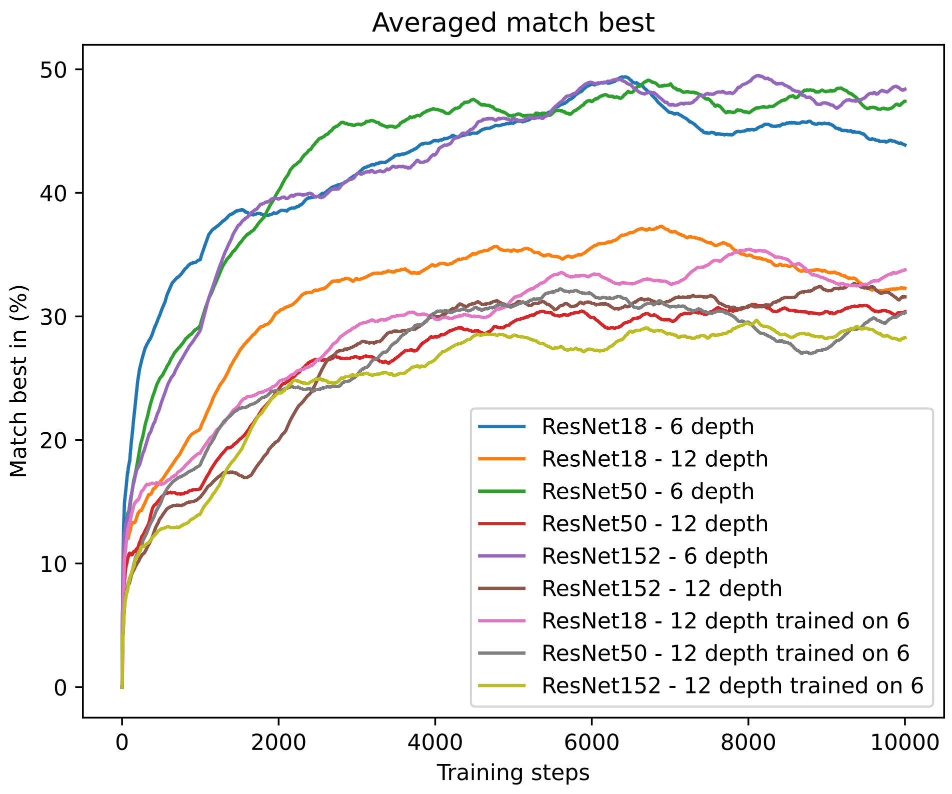


Figure 5.12 Testing less deeper trained ResNet on deeper data

The results revealed that incorporating additional depth in the training data did not improve the performance on the testing data. Interestingly, the best performance on the depth 12 test set was achieved by the ResNet18 model that was trained on the depth of 6. This finding contradicts the initial hypothesis, which suggests that neural networks could approximate the MiniMax algorithm, and that deeper evaluations would require more complex neural networks to achieve better performance.

5.7 Board complexity

The results obtained from the previous experiments revealed that the ability of convolutional neural networks to learn from higher-depth searches of the MiniMax algorithm was severly limited, as the error rates remained similar to the models trained on lower-depth searches. Nevertheless, in the best scenarios, there were test cases in which the best move was correctly predicted by the neural network for a given board position, even at a depth of 12 moves, in approximately 40%.

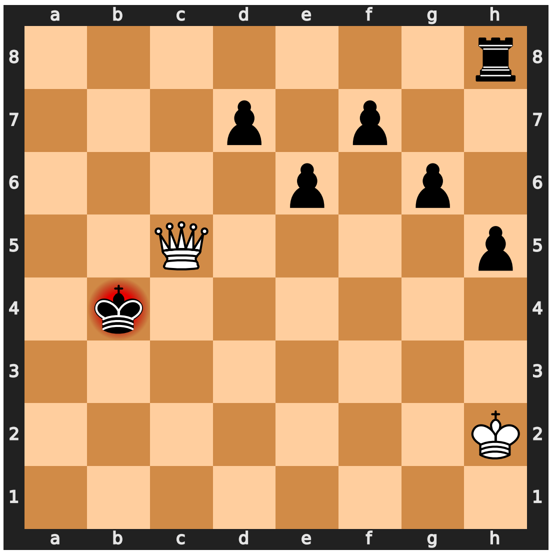
To gain a deeper understanding of the relationship between the MiniMax algorithm and the neural network's forward pass, it is important to assess cases that are easy for the neural network to identify as opposed to harder ones and investigate the mechanisms underlying how the neural network evaluates board positions and selects optimal moves. By doing so, it can be determined whether there are any special cases in which the neural network can accurately predict the MiniMax algorithm or whether this is fundamentally an impossible task.

5.7.1 Necessity of measuring a board complexity

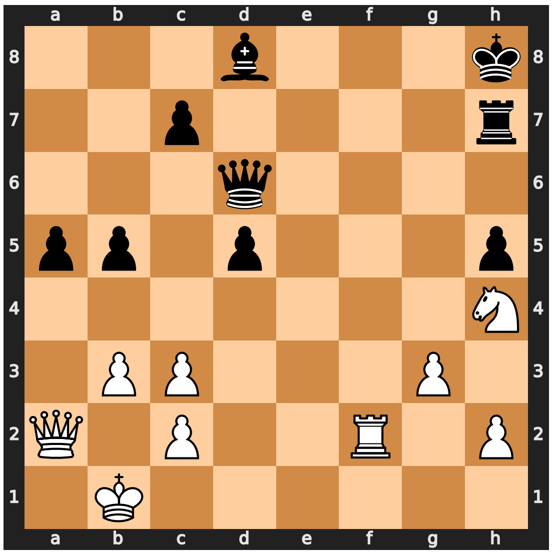
Achieving a 40% match to the Stockfish brute-force MiniMax algorithm implementation run to depth 12 may seem impressive at first glance. However the final performance in the game of chess is determined by the sequence of moves leading to a win or a loss. Therefore, assessing the ability of a subject to predict the next move on a single position cannot be done without the consideration of complexity of the position and the difficulty of understanding and evaluating it in order to select the next move. While some positions in a professional game may be predicted by a beginner, such as the second move of a trade, others may be so complex that even advanced players struggle to identify the following move. It is thus necessary to determine the proportion of the generated dataset that contains highly complex and difficult-to-understand board positions, as opposed to those that are relatively straightforward and less complex.

5.7.2 Example

Let us consider the following two board positions 5.13a, 5.13b to examine the impact of complexity on the ability to understand and evaluate next move.



(a) Clear move (black on turn)



(b) Surprising move (white on turn)

Figure 5.13 Examples of different complexity boards

In the first position 5.13a, where black is on turn, the optimal move is easily recognizable by any player, regardless of their skill level or by any chess engine used, regardless of the depth of search. It is evident that black should capture the white queen with king.

Expert comment: *”If black decides not to capture the queen, white manages to take the rook on h8 (and, subsequently, all the pawns) using a series of checks. As an example, after Kb3 white pushes the black king down the board by playing Qb5+ and as black cannot enter a black square (white gives a check on the diagonal a1-h8), after Ka2 (or Kc2) Qc4+ Kb1 Qb3+ the rook is lost in 2 moves.”* [21]

In contrast, the second position 5.13b is more challenging. While a beginner may choose to capture the hanging pawn on a5 with his queen, a more sophisticated move, which requires deeper analysis, is to move the rook to f8 and deliver a check.

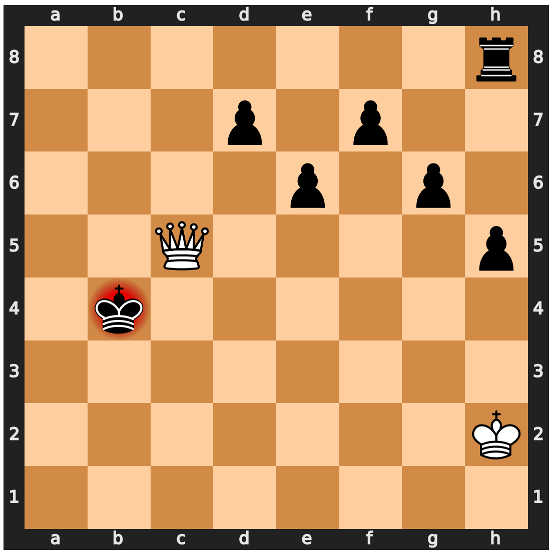
Expert comment: *”What follows is that white wins the black queen for the rook and the knight, which, in this position, results in a quick win. The two options for black are Qxf8 and Kg7, which white responds to with Jg6+ and Jf5+, respectively. Both options lead to the capture of the black queen in the next move.”* [21]

These examples illustrate how board complexity affects the ability to understand and evaluate a chess position, and highlights the importance of considering the complexity level of each position in assessing the ability to predict next move.

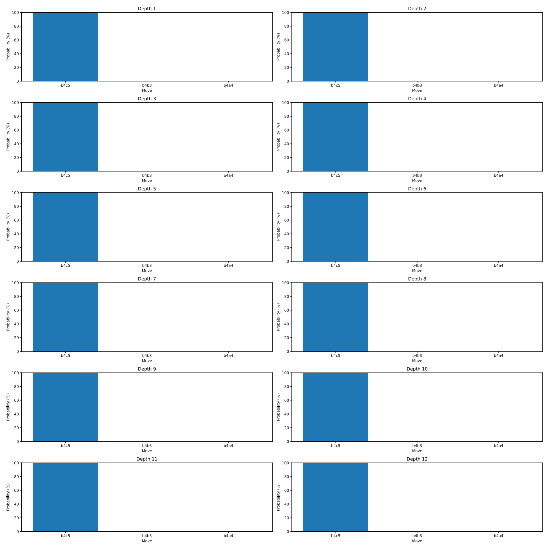
5.7.3 Board evaluation at a given depth

The question that arises is what distinguishes these two boards, and whether there is a way to algorithmically measure these expertly observed results. The main difference between the two positions is that in the first position, the benefit of the optimal move is immediately apparent, while in the second position, the advantage is hidden in the game tree.

To further examine this, we can analyze the probabilities of the next moves at various depths for both positions. Let us first consider the probabilities for the first position:



(a) Clear move - it is obvious that the following move will be Kxc5 reggardless of player's experience



(b) Histogram of evaluations according to particular depths - Kxc5 is still the only option with probability near 100%

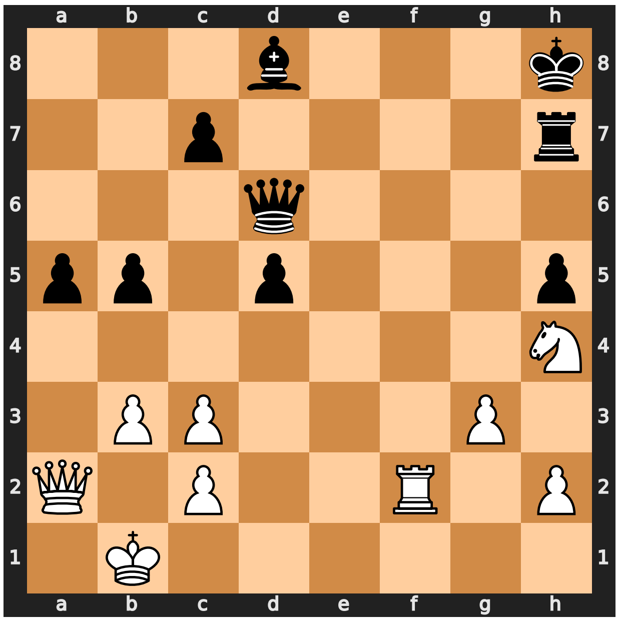
Figure 5.14 Examples of board complexity

As shown in the analysis, the probability of the optimal move "king takes queen" remains consistently high regardless of the depth of evaluation, indicating that the position is straightforward and easy to evaluate. Therefore, this position can be assigned a complexity value of 0.

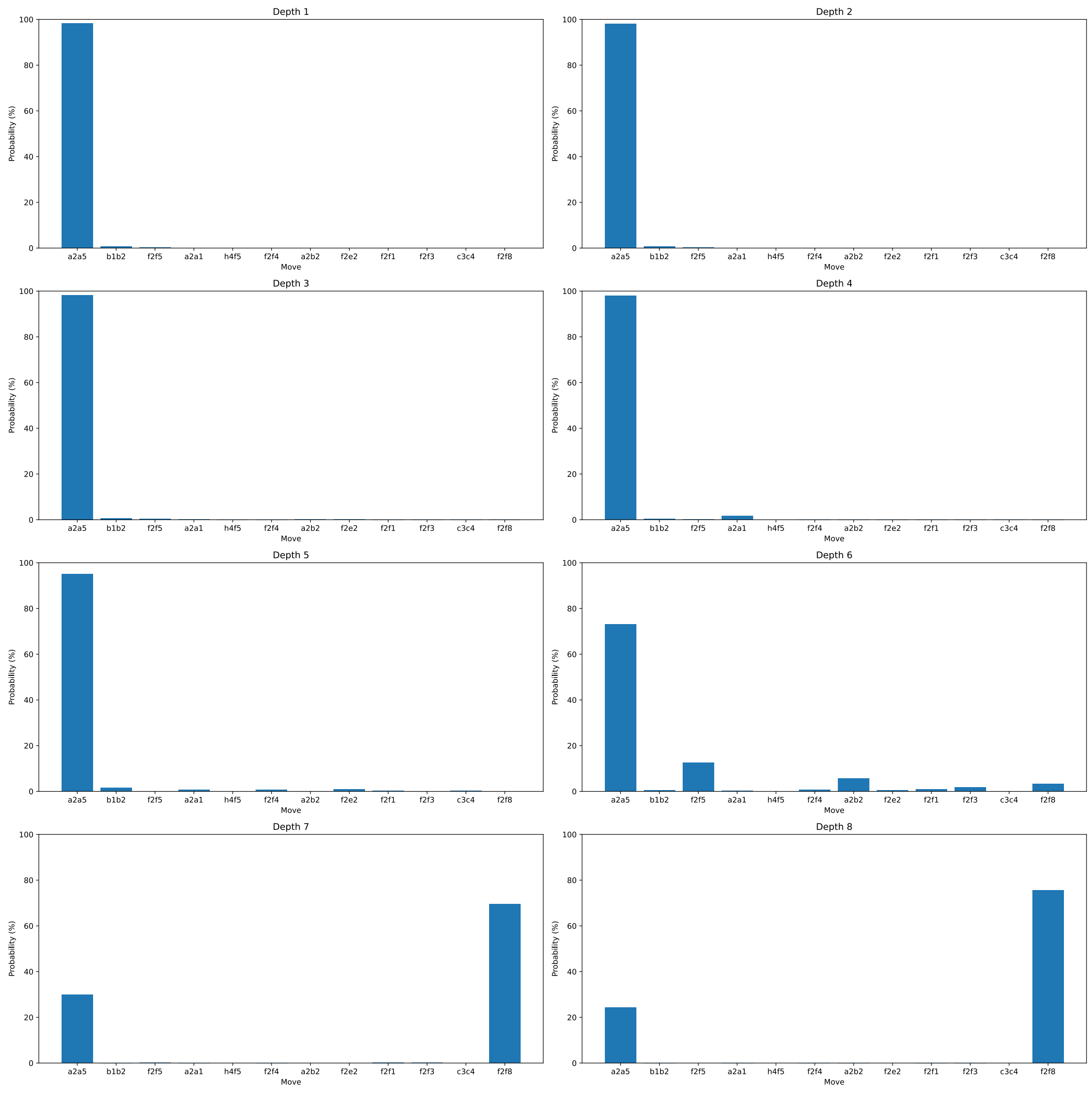
In contrast, the second position 5.15 has a more complex game tree, with the optimal move changing as the search depth increases. Taking the hanging pawn is the optimal move until the depth of 7, where f2f8+ suddenly arises as the best move from an almost negligible initial evaluation. This implies that some important information is hidden in the game tree, and to uncover it, a search up to depth 7 is required. Therefore, the complexity evaluation of this board would be assigned a value of 7.

5.7.4 Board complexity metrics

The proposed metric evaluates the board position by progressively searching deeper and deeper until it finds a move that was initially evaluated with a probability below 5%, but newly has the highest probability of being played, with a probability over 40%. In case no such example was found, the complexity evaluation for that board position is assigned as 0.



(a) Surprising move



(b) Histogram of evaluations according to particular depths - f2f8+ becomes important at the depth of 7+

Figure 5.15 Examples of board complexity

5.8 Experiments

To determine whether there are some special cases, where CNNs can approximate the MiniMax algorithm, a simple and straightforward experiment was suggested. The experiment involves comparing the match accuracy of two sets of board positions, one with a complexity level of 0 and the other with a complexity level of 6. The aim is to identify instances where the CNNs make errors on the simpler board positions, and to examine where the model is capable of predicting the optimal move at the more complex level. In order to establish a baseline for comparison, random predictions must be included, as the board positions in both sets can represent highly diverse data with varying numbers of possible moves.

5.8.1 Results

ResNet152, which exhibited the highest accuracy at the depth of 6 on the testing set during the training process, was selected for testing purposes. This model achieved an average accuracy of approximately 50% accuracy over 1000 training steps 5.10. To establish a baseline for comparison, random predictions were averaged over 100 testing iterations.

Results are as follows:

ResNet152		Random predictions	
Complexity	Accuracy	Complexity	Accuracy
0	44%	0	5.5%
6	11%	6	7.7%

Table 5.2 Comparison of ResNet152 and random predictions

The results are also shown in the figure 5.16

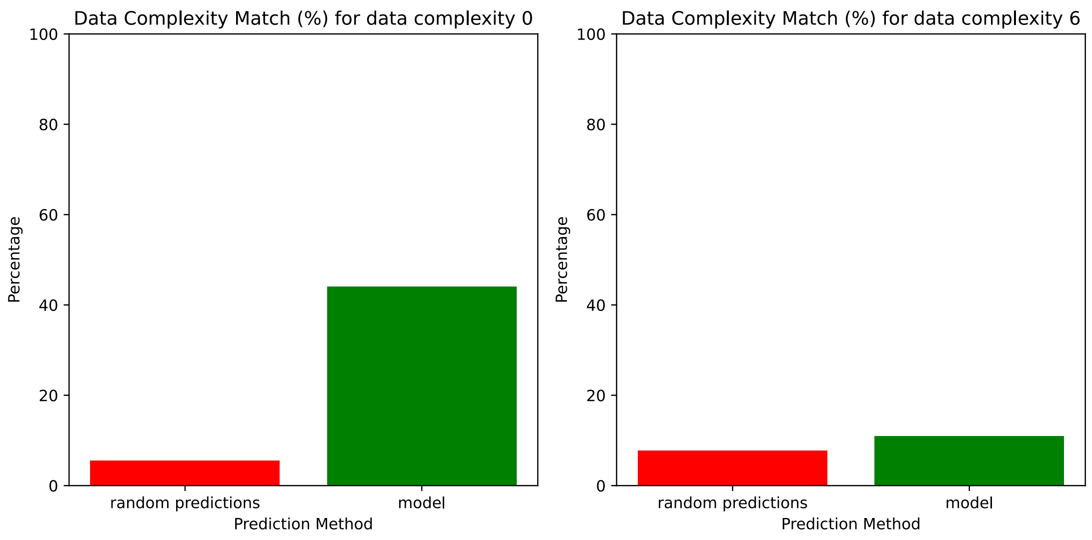


Figure 5.16 Graphical representation of results

As expected, the results obtained on board positions with a complexity level of 0 were similar to those achieved on the whole testing set regardless of the complexity split. However, it is important to examine some mismatched moves to see, where the model generally misclassify.

On the other hand, the results obtained on board positions with a complexity level of 6 were slightly better than random predictions. Therefore, it is essential to examine specific examples where the model was capable of accurately identifying the optimal move.

5.8.2 Mismatched moves in the low complexity test set

There are several reasons why CNNs may predict a different move than the optimal move suggested by Stockfish. One of the most significant reasons is that there are often situations where each piece is safe, so the selection of the next move depends on the individual strategic preferences. Another reason can be a situation where multiple moves are equally close to being optimal. It is important to note that a complexity level of 0 does not necessarily imply that there will be only one strong move available. It simply means that, at first glance, the optimal move is not unexpected.

An example of a balanced situation where a player can choose an individual strategy is the opening board, while an example of the second situation is as follows: 5.17

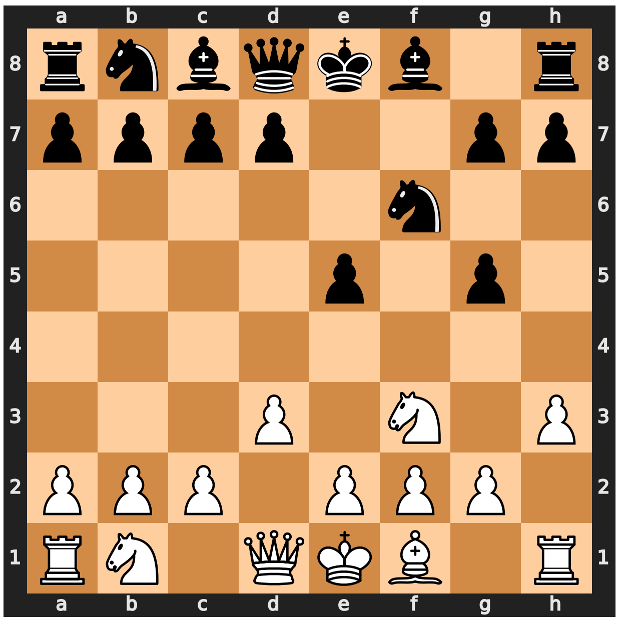


Figure 5.17 Example of two optimal moves

Expert comment: *"Although taking the pawn on e5 looks better from a strategic perspective (taking the center is a major factor in the decision making process), taking the one on g5 does not change the overall evaluation of the position - white is still much better."* [21]

The majority of examined misclassified situations exhibited this pattern, and further observations did not reveal any general types of mistakes made by the model.

5.8.3 Matched moves in the high complexity test set

More interesting observations has been made on the high complexity testing set. The results mostly indicated that using a sophisticated engine like Stockfish, which is designed to run into deeper levels of the game tree, is not optimal for implementing board complexity metrics demanding initial evaluation at the small depth of 1.

One such example is this 5.18 position where Stockfish, at the depth of 1, would suggest moving the king, while capturing the bishop was initially given a very small chance. However, the model still identifies capturing the bishop as the best move, which is the correct behavior.

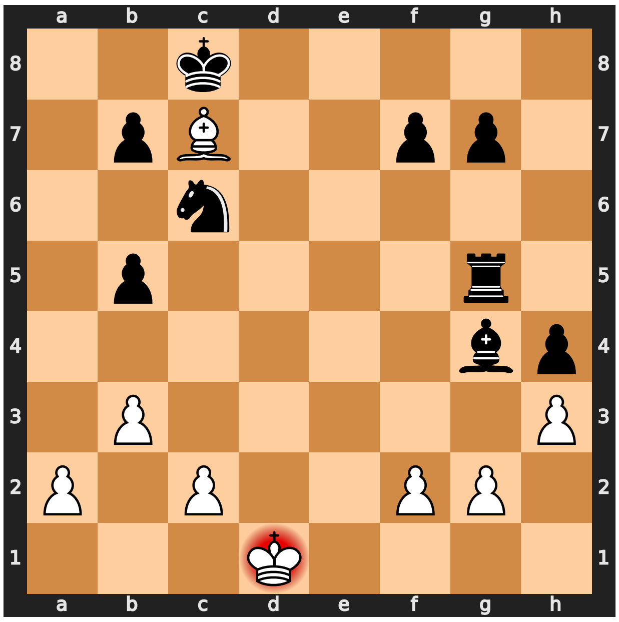


Figure 5.18 Example of Stockfish misclassification at the depth of 1

Due to the described limitations of proposed metrics using Stockfish, brief expert evaluation of the testing set and cleanup was performed. Simple cases, which were misclassified to a higher level, but upon expert evaluation exhibited complexity 0 were removed. Resulting testing set was then filled with other positions. Both final testing sets included 100 chess board positions.

Here is another example 5.19 that highlights the weaknesses of this specific board complexity implementation using Stockfish. In this particular case, black has clearly lost and has no chance of winning. A human player would definitely resign at this point, but in this specific case, it can be very challenging for a computer engine to make a move, because all moves are practically equal. However the engine still must decide and in this particular case Stockfish played c6. Surprisingly, model identified exactly the same move. As a possible explanation, there may be a possibility that the model was able to learn some default behavior of the engine in these specific situations.

Expert comment: *"Why would the engine suggest Kc6 in the depth of 6? The reason is that you can prolong getting mated - note that you only get mated close the border and not in the center - by blocking off the white king and forcing the white queen to lose a few moves to push the black king back to a corner."* [21]

It is apparent that to extract this strategy, the game tree of higher depth is required.

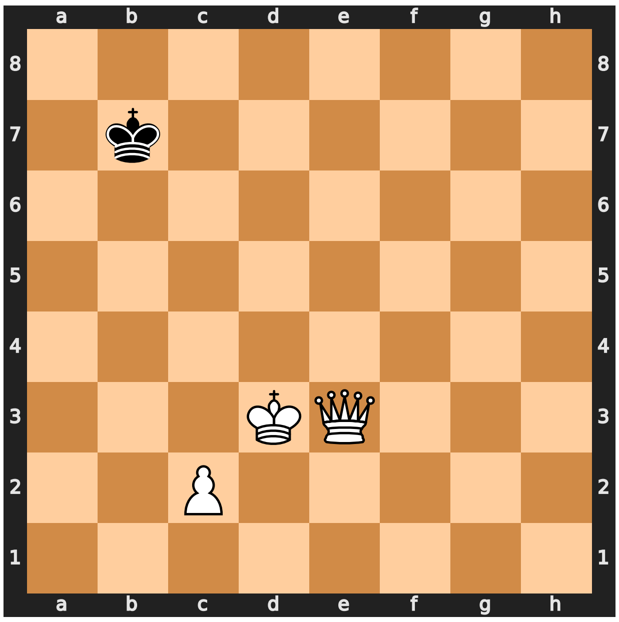


Figure 5.19 Example of weakness of implemented metrics

One of the most interesting situations 5.20 is where the model is able to recognize a pattern of the king moving towards its pawns in the endgame. However, in order for an engine to calculate the benefit of this move using a game tree, a deep game tree must be constructed due to the long distance to the pawns. This suggests that there are cases where there is a visible pattern that is connected with the optimal move evaluated in the deeper levels of the game tree. Other examples of such situations include giving a check, which is often a very powerful move, but sometimes it is only revealed after multiple moves.

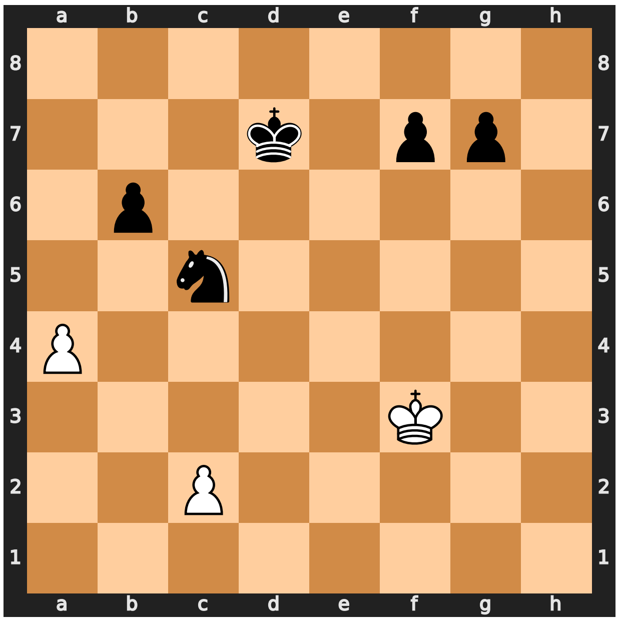


Figure 5.20 Visible pattern

Finally it could be possible that the network would match a small number of positions due to overfitting, resulting from the random generation of the testing and training sets without awareness of other set. However, due to huge vastness of chess state space, its probability is negligible.

All remaining positions were expertly evaluated, and none of them demonstrated the potential for convolutional neural networks to select optimal move based on information hidden in the game tree.

5.9 Conclusion

5.9.1 Result summary

Despite some limitations in the implementation of the complexity metric evaluator, the supplementary use of human supervision allowed the construction of testing sets at the required complexity. This revealed that, with the exception of certain special cases, convolutional neural networks are not able to extract and utilize the information hidden in the game tree. While CNNs are useful for replacing heuristics necessary for the MiniMax algorithm, they cannot substitute the algorithm itself.

The observation that neural networks are incapable of approximating the output of the MiniMax algorithm can be very helpful across various domains even outside the game of chess. This phenomenon not only provides greater insight into the mechanisms of convolutional neural networks within the context of chess gameplay, but also in relation to other state-space search problems.

5.9.2 Intuition behind the mechanism of neural network in the context of the game of chess

Although providing an exact proof of how convolutional neural networks decide may be challenging, because the underlying mechanisms can be trained on complex and hard-to-explain features, it is still beneficial to consider the theoretical foundations of the use of CNNs in image processing. In a chess engine, CNNs play the role of a human-like intuition that enables the rapid assessment of board positions based on visible patterns. For instance, a human master can quickly understand a given position with a brief glance, taking into account factors such as the number of pieces on the board, king safety, pawn structures, control of the center, whether the position is open or closed, and the availability of piece movements. In this sense, CNNs replicate and enhance this intuitive decision-making process.

However, while this information provides a useful starting point, it is insufficient for selecting an optimal move at a higher elo level without detailed information about potential future gameplay. The full decision-making process of an advanced player includes simulating possible outcomes of gameplay. In practice, advanced players are capable of searching and calculating up to depth 7 and around 100 positions. Due to the effective pruning of unnecessary calculations based on intuition, even such small amount of positions can result in moves extremely powerful, such that achieving the same level of accuracy in a brute-force computer engine would require searching through millions of positions.

5.9.3 Maia chess bots - performance explanation

With understanding capabilities of CNNs, possible explanation of similar performance of multiple Maia bots arises. Coming back to strange phenomenon described in part 4.5.10.2 that Maia models trained on Elo of 1100 has a similar final Elo in gameplay as model trained on 1900 Elo dataset, it is possible to use gained knowledge for possible interpretation.

The observation that the performance of the engine is not essentially affected by the Elo ratings in the dataset is indeed interesting. While it is expected that a dataset with more mistakes would lead to a lower Elo rating, final model does not seem to be affected by this fact so much.

To begin with, it is necessary to provide a discussion regarding the differences between novice and advanced chess players.

One of the most significant disparities lies in the strategies being used. Advanced players possess the ability to analyze the board position from multiple perspectives, taking into account various aspects such as the openness or closeness of the position, and the level of activity of each piece. They do not focus solely on pieces, that were moved in last few moves, but rather consider all pieces on the board.

Another distinguishing factor is the depth of search and the number of positions that advanced players can keep in mind. While beginner players can usually only think one move ahead, advanced players can search for possible gameplay further. They can anticipate the consequences of their moves and assess potential threats and opportunities that arise as the game progresses.

Lastly, an important aspect that sets advanced players apart from novice players is the frequency of mistakes. While it is common for beginner players to make blunders, advanced players, even if they occasionally make a mistake, are more precise and make blunders only rarely.

Now, with an understanding of the differences between beginner and advanced chess players and the capabilities of CNNs in the context of chess, it is possible to provide basic insight into the phenomenon of bots final ranking.

Firstly, what differentiates players with different Elo ratings the most is the frequency of mistakes. However this frequency is not reflected as prominently in a neural network. This may be due to the fact that in the dataset, most moves even made by 1100 Elo rated players are near-optimal. This allows the model to generalize basic strategies such as capturing and protecting pieces. Since mistakes represent almnost neglibigble percentage of the dataset, generalizing optimal strategies has higher positive impact to model final loss function score in comparison to learning making mistakes.

Secondly, the difference between beginners and advanced players lies in the depth of search and the ability to discover hidden opportunities deeper in the game tree, which is a task that CNNS are not capable of. Thus deeper search in the training data will not affect final performance regardless of the quality of information in the training data.

Lastly, the use of advanced strategies is another aspect that differentiates players of varying Elo ratings, which CNNs are capable of learning. However, since this aspect does not have as much of an impact on the resulting Elo in contrast to frequency of mistakes or blunders, the final Elo rating is only slightly affected.

It is important to note that while CNNs can learn certain common strategies, there are some strategies that are too complex to be fully utilized in a single forward pass. As a result, the maximal performance of the CNN may be limited within achieved range and may not be possible to push much further even with better data or more complex models. Moreover, it is important to remind that CNNs are not capable of deep analysis, which means that the gameplay may be comparable to using only heuristics without MiniMax. This corresponds to what could be described as an ”intuitive playing style”.

5.9.4 Further ideas utilizing gained knowledge

With this knowledge, new possibilities for utilizing this information to increase accuracy of matching human moves arise.

One approach involves directly inputting information about the complexity and MiniMax output into a neural network, since these additional features cannot be extracted by a convolutional neural network.

Another method involves training a separate model specifically designed to predict the ability of players at a given Elo rating to understand boards with a particular complexity. To implement this methods, small number of nodes can be expanded using a neural network model serving as a pruning and evaluation function. With nodes sorted by probabilities, parameter of amount of node expansion can be easily scaled to players within a specific Elo range and can be a trainable parameter of the system. This approach may be sufficient even for small node depths and could satisfy human players, who tend to expand and intuitively prune fewer positions. Therefore, if the engine is unable to find a move, it is unlikely that a human player would have been able to find it.

Although these are just hypotheses, the intuition behind them suggests that selectively incorporating tree search might positively impact overall matching performance. This would involve using tree search only in situations where the complexity is low enough for humans to understand but high enough for the model to predict. In order to approximate real people on a more advanced level, some kind of tree search can be utilized in cases where the neural network fails.

Even though described ideas are just speculations, they represent an interesting and promising problem for further investigation (outside the scope of this thesis), with potential benefits for the Maia project. Incorporating tree search selectively could help not only match player style, but also their Elo and ability to search n-amount of positions. By doing so, the Maia-similar bots could not only match similar playing styles, but also the performance as human players and thus address current issues, such as strange behavior of playing with brilliant intuition followed by very simple mistakes that are inadequate for players at their level.

5.9.5 Optimal training data

Another question that arises is related to the training data. If CNNs learn only heuristics, human games would likely be better suited for training than computer engines, since they represent more intuition-based gameplay as opposed to engines, where evaluations and moves selected can be very unintuitive even for humans due to information hidden deeply in the game tree.

However, it should still be possible to extract common strategies even from the algorithm-based approach, but setting bigger depths of the dataset might represent noise in the data, which would be deteriorating to the model's learning. Studying this connection could be a topic for another thesis, which would require further analysis.

5.9.6 Other models in other domains

Another interesting question that arises is how inability of CNNs to extract and utilize information hidden deeper in the game tree differs in other domains of state-space search. It is possible that there are domains where MiniMax is so straightforward that CNNs could approximate it. If not, it still may be possible for other models with different architectures to achieve it. Therefore, attempting to develop a model capable of efficiently approximating MiniMax from data and testing it across multiple domains could be a potential area for further research. This could serve as a topic for another thesis.

5.10 Performance of supervised engine

While the main objective of the thesis was not focused to train the most powerful engine possible, it is still important to discuss the achieved performance. Remarkably, the model surpassed expectations, particularly the policy head, which demonstrated outstanding results. However, the engine's overall performance, taking into account all factors, was lower than expected.

5.10.1 Accuracy of the model

As illustrated in the figures 5.10, the policy head average of 1000 training steps achieved approximately 50% matching accuracy of Stockfish. Without computing the average, maximum matcing accuracy was almost 60%. Although these results appeared promising, the average MSE for the value head was 0.1, which corresponds to a RMSE of 0.32 within the range of -1 to 1. These values are roughly equivalent to the advantage one knight. 5.2.

From the initial perspective, this level of accuracy would have been considered unacceptable. However it is expectable with the understanding of the relationship between the MiniMax algorithm and single forward pass of the neural network, which reveals that there is a considerable amount of data that the neural network cannot comprehend.

Despite the poor accuracy of the model on the testing set, the final performance of the engine is not that low. The average error is likely caused by positions where the neural network evaluates in a simplistic manner, such as by using simple piece count and visible features. However, the target contains information that is hidden in the game tree which CNN is unable to learn and which clearly manifested in described MSE. Still quality of resulting gameplay exceeds expectation. One possible explanation might be that the errors are mostly present in specific situations where Stockfish utilizes information hidden deeply in the game tree, but in vast majority of other cases the outputs are comparable and the output of more simpler evaluation cannot be marked as wrong but still more or less valid and quite acceptable as a heuristic function. So the impact of missing the strong move to overall gameplay is relatively small.

Nonetheless, the engine still makes significant mistakes. MiniMax is generally sensitive to extreme values and requires rather stable heuristic function. Model might evaluate lots of board positions mostly acceptable, but in case a significant error is made, impact in MiniMax is also profound. If an evaluation function is poor, expanding a high number of nodes may not improve the performance of the engine. Even if only a single value in the vast game tree is incorrect, it can lead to the selection of an entirely different branch. Such a value may represent an opportunity or a threat that in reality does not exist, thus causing the selection of a suboptimal branch or avoidance of a good one.

5.10.2 Rollouts

To minimize above mentioned problem, the final supervised engine implementation included expansion of nodes in layer 1 and their evaluation using a combination of the value head and 20 rollouts. By adopting this approach, the engine was able to demonstrate basic approaches to gameplay, such as capturing and protecting pieces, with basic intuition provided by the neural network. Even though engine was still capable of making a significant mistake, most of the time it played reasonably.

5.10.3 Expert comment on the final engine performance

A sample game was submitted to the expert for an evaluation.

Expert comment: *"In the opening phase, the engine actually follows a natural line which can be found in many games by strong players. The move 8 - Nd5 is, of course, a blunder, but from a strategic perspective it makes sense to centralize minor pieces."* [21]

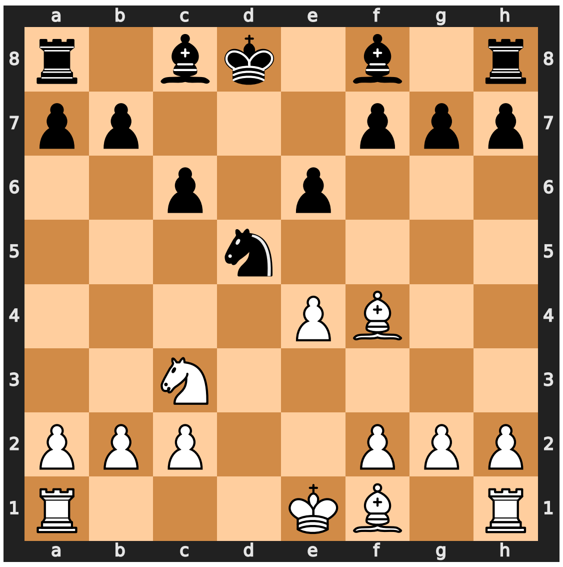


Figure 5.21 Blunder

"Similarly 9 - Nxd5 is a big mistake, as white could simply have captured on d5 with a pawn, but after that the game actually develops in a rather standard manner. Both sides develop pieces and deliver several checks, and no major mistake occurs." [21]

While rollout evaluation can serve as a robust predictor of who is likely to win. However one weakness can be demonstrated using this example: when the model prefers capturing with knights, black can repeat mistakes of the white, which could potentially result in the elimination of two similar blunders simultaneously.

Chapter 6
Reinforcement learning

This chapter is dedicated to the reinforcement goals of the thesis. Section 6.1 describes training process and dataset generation. Section 6.2 introduces in superiority of reinforcement learning and potential limitations caused by inserting human-like information. Sections 6.3 and 6.4 propose alternative methods for reinforcement learning while sections 6.5, 6.6, 6.7 and 6.8 recapitulate obtained results.

6.1 Reinforcement learning solution

6.1.1 Training process

The dataset for reinforcement learning was generated using self-play. Specifically, a process similar to that used in the AlphaZero project was used [6]. In AlphaZero project as well as in this thesis one training step consisted of generating 4096 board positions. Once the 4096 positions were generated, they were backpropagated to the neural network as a single batch over 8 epochs, with the learning rate being updated using a geometric sequence with a hyperparameter $gamma = 0.9$. Finally, a small amount of $L2$ regularization was applied to prevent overfitting.

6.1.2 Generating data

The dataset for reinforcement learning is generated through a process of self-play. In order to ensure that the engine trains effectively, two conflicting rules must be balanced.

- The first rule (diversity) is similar to the process of generating datasets for supervised learning. The positions used must be diverse and representative of a range of playing styles, including a variety of advantages and disadvantages. The dataset should also cover all stages of the game in equal amount.
- The second rule (quality) is unique to reinforcement learning. The target variables must be trainable and the target must match a learnable pattern. When examining a position rated as 1, it should be clear why it led to a win. The selected moves must also be predictable to a certain extent.

6.1.3 Diversity

To fulfill the first rule, a moderate extent of randomness must be introduced to add variety to games. If the engine were to rely solely on deterministic behavior, all games would be identical, causing the model to overfit those games and preventing further improvement. Even with a small amount of randomness, the engine may still fall into a vicious circle. If it discovers a strategy, it would continue to use it, causing the policy network to prune too much and restrict the engine's ability to consider other valuable moves. Then the engine would continue to behave in a similar manner over time, leading to a situation where the engine would become stuck in a pattern of behavior and would fail to return to a diverse range of gameplay. Thus to meet the first requirement, it is optimal to add as much randomness as possible.

6.1.4 Quality

However, excessive randomness also presents a problem, as it can result in a loss of learnable patterns in the data. In an extreme scenario where all moves are completely random, it would be impossible for any model to learn such data. In a more realistic scenario, if the engine were to play suboptimally, the quality of the dataset would decline, resulting in suboptimal training. To fulfill the second requirement, it is important for the engine to select moves it evaluates as best and to follow a pure deterministic approach.

As these rules are contradictory, a balanced sweet-spot must be found, to satisfy both of them to some degree.

6.1.4.1 Opening database

One common solution to the issue of insufficient variety in chess games is the use of an opening database. This can be a robust solution, but it also limits the learning process by restricting the engine's ability to discover new optimal openings. Furthermore, training on data from the opening can affect the engine's behavior in the middle game.

6.1.5 Solution

To address this issue, this implementation includes several preventive steps.

Firstly, the first move of the game is made randomly, as there is at least one valid opening strategy for each starting move, teaching the engine to play against any of them.

Secondly, the number of game tree expansions is randomly modified to make one player stronger than the other. Both players then select a random move above a randomly generated threshold based on the number of expansions and the score of the best move. Such approach also limits the weaker player more than the stronger player, because the threshold is generated in dependence on the number of expansions, which has a greater impact on the weaker player compared to the stronger player.

6.1.6 Draws

Another important requirement for the dataset for reinforcement learning is that it should not have a high proportion of draws. When the dataset contains too many draws, the model may have difficulty distinguishing between good and bad moves. However, this requirement can be easily addressed by using a sufficient number of game tree expansions or by using a faster state space search algorithm for mate seeking as explained in section 6.3. By doing so, the dataset will contain a sufficient number of wins and losses, allowing the model to learn from the outcomes of the games.

6.1.7 Hyperparameters tuning

Lastly, one potential issue to consider is the setting of the learning rate and the number of epochs for the training process. If the learning rate or the number of epochs is too high, the model may overfit the data, leading to suboptimal dataset generation and lack of diversity in further steps.

6.2 Superiority of reinforcement learning

Reinforcement learning has proven to be a superior method for training neural networks for two-player game engines, despite the ease and speed of training via supervised learning. The Deepmind's studies demonstrated that supervised learning can yield impressive results early on in the training process, but models trained in this manner tend to reach a performance limit beyond which they cannot improve [5]. Models that are trained using both supervised learning and reinforcement learning outperform those trained using only supervised learning in the end.

Finally, models trained using solely reinforcement learning outperformed even those trained using a combination of supervised learning and reinforcement learning after a certain amount of training time.

6.2.1 Intuition behind

The likelihood of surpassing the performance of a teacher (supervisor) using a supervised learning approach is low. In this thesis, the supervised model aims to imitate Stockfish, but with lower accuracy and at the expense of increased computational demands. The incorporation of human-like information into the model hinders its ability to discover novel strategies and push overall performance to the maximum. Inclusion of any supervision during the training process could make final engine suboptimal and should be avoided.

6.2.2 Human-like information

In order to avoid limitations of the model's ability to converge and improve its performance later on, methods that incorporate additional heuristics, such as those derived from human experts or other engines, should be avoided. For instance, evaluation of the position based on the difference in weighted pieces count between players or rewarding captures of pieces may initially appear to be a viable strategy for the training process, because having higher material count compared to the opponent increases chance of winning. However, this approach may lead to a focus on material-based gameplay rather than exploring alternative strategies, such as sacrificing pieces, which can be a more effective in certain scenarios.

This represents a significant challenge in reinforcement learning. For the architect, the goal is to train the engine to invent and discover new strategies that may not have been previously considered. However, this creates difficulties in the training process as it becomes challenging to assess the impact of different training strategies and approaches on the model's performance without a clear understanding of the desired outcome. It is difficult to determine which steps will result in an improvement and which would have a negative impact on the model's performance.

6.3 Quick search space algorithm for mate seeking

One idea of how to accelerate training process would be using quicker state-space algorithm in the endgames. There is some intuition behind this method and some hypothesis about principles, that could be responsible for accelerating the training process.

6.3.1 Discussion over deterministic and heuristic functions in a chess engine and place for deep neural networks

There have to be two types of functions in a chess engine: deterministic functions and heuristic functions.

Deterministic functions provide rules to control the expansion of the game tree and also position evaluation to stop the tree expansion.

Then there are heuristic functions. Heuristic functions simulate an expert's opinion. In a chess engine, these functions have two use cases. One of them is board evaluation - an expert speculating on the probability of winning in a specific situation. There is some room for different approaches or opinions on evaluation of a particular position. The other heuristic function is a pruning function in the game tree. In practice, there is no deterministic rule for guessing the next move. Certain number of moves can be evaluated, whether they will be played or not, and then assigned with probabilities. These functions can be expertly designed or replaced by neural networks.

The best place to use neural networks in a chess engine is to replace expertly designed heuristic functions. Board evaluation can be replaced by a neural network in the middlegame, where board positions are not leaf nodes of the game tree. However, with leaf nodes, it is a different case. The end of the game is a deterministic state in the game of chess, and these positions can be evaluated using a deterministic algorithm without any heuristic.

6.3.2 Benefits of deep learning based on stage of the game

Let's analyze a game of chess and try to identify where deep learning is most useful: opening, middlegame, or endgame.

It is believed that the best strategy for the opening is to use a database approach. There are not so many different positions, so you can evaluate a huge part of them with more computational time than during gameplay. Then memorize the result and use it in the game for as long as possible. Although one can use deep learning to find the best opening, this opening can always be memorized and used even without the model being present.

The most significant benefit of using neural networks in a chess engine is in the middlegame, where the state space is too large and the probabilities of winning are hard to predict. This is the key part where heuristic functions are used the most, so the model would highly benefit if these functions were more accurate.

Lastly, in the endgame, state-space is not as vast, so pruning is not as useful anymore. Additionally, positions are often in leaf nodes of the game tree, so there is no use for an evaluative heuristic function. Therefore, a chess engine would not benefit as much if these heuristic functions were improved.

It may be smart to use deep learning in the opening and middlegame, but use a faster state space search algorithm without deep neural networks to seek checkmate in the endgame.

6.3.3 Method description

Deep learning is well-suited for heuristic substitution, and a quick state search algorithm is ideal for seeking checkmate in the endgame. However, it can be challenging to distinguish between these two cases and select the appropriate algorithm.

Implementation used in this thesis utilizes a deep neural network approach in all stages of the game, but simultaneously also utilizes a quick state space search algorithm to seek a forced mate in the depth of 10 ply moves. If there isn't a forced mate opportunity found, the deep neural network is used. However, if a forced mate opportunity is found, control is transferred to the quicker state space search algorithm that discovered the forced mate.

6.3.4 Potential benefits

This method can not only increase the engine's performance, but it can also lead to a faster reinforcement learning process. The intuition behind the benefit of using a quick state space search algorithm for reinforcement learning is that if a player makes a significant mistake, the other player can immediately take advantage of it. Therefore, every major mistake will be taken into account, and the losing player will be punished much quicker while the winner will also be rewarded much quicker.

In games where major mistakes are not punished, the dataset can be inconsistent, resulting in slower and suboptimal training. This approach decreases dataset inconsistency by reducing the chances of winning a game from a clearly losing position.

Games using this method typically have fewer moves, and thus there are more games in the dataset with a given number of positions. Such approach allows for the use of fewer nodes to expand in the game tree while still having shorter endgames.

Additionally, there is be a higher proportion of middle game positions compared to endgame positions in the dataset, which are much more useful for neural networks.

6.3.5 Summary

The best use for neural networks in a chess engine is to replace expert-made heuristic functions in the middlegame. Using a quicker state space search algorithm can lead to better performance and more consistent datasets for neural network training. This approach results in more games with a better middlegame/endgame ratio, using fewer expansion nodes in the game tree, which is highly beneficial for reinforcement training.

6.4 Linear approximation of reward

Another idea, how to accelerate reinforcement learning process is linear approximation of a reward. There is some intuition behind this method and some hypothesis about principles, that could be responsible for accelerating the training process.

6.4.1 Method description

In standard reinforcement learning, rewards and punishments for winning or losing a game are often assigned as -1 and 1 or -1, 0, 1 for games that can result in a draw. However, in reality, the probability of winning for a winning player is not constantly 100% from the beginning of the game, but gradually increases from 0 to 1 (still within the range of -1 to 1). To take this gradual increase in probability into account, a linear reward function can be used to better approximate reality. In this approach, the board position is still evaluated within the range of -1 to 1, but the reward is beveled linearly until the end of the game. Specifically, the reward starts at 0 from the beginning of the game and increases linearly to 1 for the winning player, while decreasing linearly to -1 for the losing player. This approach allows for a more accurate evaluation of players' performance over the course of the game.

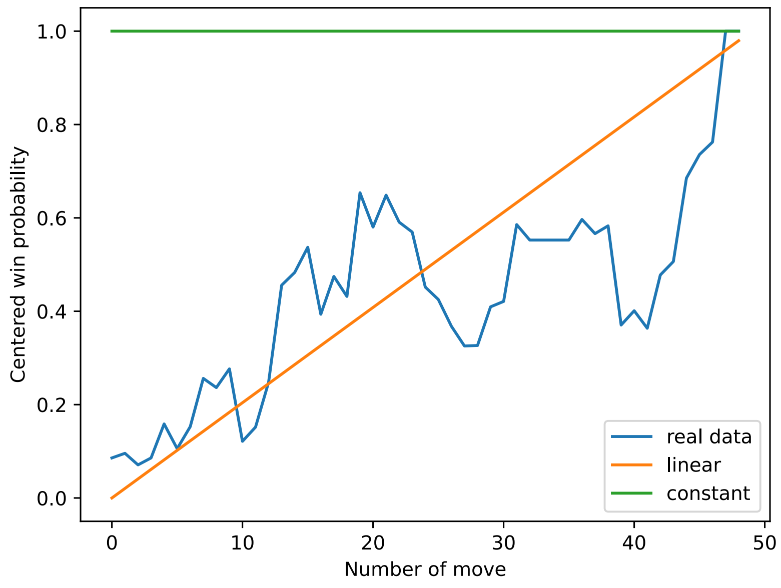


Figure 6.1 Real Distribution

In order to support the intuition, let's take one random game as an example. In this evaluated Stockfish game against itself, it can be observed, that linear reward approximates reality much closer than constant reward. For comparison, in this game, RMSE for linear reward is 0.19 in centered probability, which corresponds to value of 10% in standard probability range. However, but RMSE for constant reward is 0.6, which corresponds to 30% in standard probability range.

6.4.2 Relation of optimal move selection and position evaluation

Linear reward modifies the target variable for the value head of the network, which evaluates a board position, not a move. But the goal of the engine is to select a move using information about possible positions. So in this context evaluation a move has similar meaning to evaluation positions to which the move leads to.

6.4.3 Extra information about a position

In many games, a single blunder can lead to a quick loss, while a suboptimal move may lead to a loss over a longer time. Linear reward helps with differentiating between these two cases, thereby can lead to faster convergence during training.

For instance, if an engine is quite untrained and has to choose between two moves, both of which have led to a loss in the past, it may choose either one with similar probability. However, if the engine has additional information that one move led to an immediate loss, there may be a small chance to achieve a win, but if the second move led to a loss after next 50 moves, there is a higher likelihood that making slightly better decisions than in the past would lead to a success. Therefore, the chance of selection of a better move increased.

Overall, the utilization of linear reward functions can be a valuable technique in distinguishing between various types of positions, enhancing the quality and the speed of the learning process by providing additional information.

6.4.4 Decrease in the dataset inconsistency

The standard reward/punishment method of -1, 0, 1 often produces an inconsistent dataset, particularly when the engine is untrained and is unable to differentiate between good and bad moves. As a result, all games become highly randomized, making it challenging to predict the outcome of a given position. Even well-trained engine may generate inconsistencies, especially at the start of the game, as it is difficult to predict how the game will progress. For instance even such well-trained engine can generate unstable multiple evaluations to the one particular position with values such as -1, 1, 1, 1, -1.

Linear reward functions generate more consistent dataset, particularly in the initial stages of the game where prediction of the outcome is problematic. For instance, linearly rewarded values from the start of a game may appear as 0.02, 0.1, -0.1, and 0.05. Such values are more consistent compared to -1, 1.

Over time, dataset inconsistencies in the later stages of the game should decrease automatically since prediction of the result of the game becomes more reliable as it approaches its end. Thus, more profound dataset inconsistency is related the beginning stages of games, which linear reward helps with.

6.4.5 Impacts of dataset inconsistency

If it would be possible to fit all the data into one training batch, dataset inconsistencies would not have as significant impact. The value that minimizes MSE of 0.1 and -0.1 would be 0, the same as for -1 and 1, which would be equal to 0.

This is a standard approach, how models learn from high amount of data. In supervised learning this is not as significant issue even when using multiple batches, because all correct training data are already prepared before the training process. Thus shifted model parameters from one batch would be corrected in further training.

However, reinforcement learning is an iterative process, where actions, which generates next training dataset, are selected based on current model's state, so the impact of the shifted model's parameters is much more significant. In order to overcome mentioned effect, each step should contain vast and fully representative dataset, but in practice, these optimal conditions are impossible to achieve.

Consequently, the model's parameters could be shifted towards some unrealistic evaluations. In order to prevent this effect, it is crucial to consider these dataset inconsistencies even though in theory they may not appear to be a significant issue.

6.4.6 Summary

By having a reward distribution that is closer to the actual data, the convergence rate of the model can be accelerated, reducing the need for a large amount of data. Had the model had access to only one game, it would have been able to handle situation mentioned above better than in case of constant reward, and approximate the true probability more accurately. Subsequently, the following games would also be more precise, leading to a quicker learning process.

It is even possible that this training method could lead to a further increase in the engine's performance due to additional information provided by the target variable.

6.5 Results of reinforcement learning

6.5.1 Evaluation metric

6.5.1.1 Issue of Elo metric

The assessment of a chess engine's performance typically relies on the Elo metric, which becomes measurable at a threshold around 1000, corresponding to the skill level of a beginner chess player. However, to achieve such an Elo score through reinforcement learning, thousands of training steps must be made, as was demonstrated in the AlphaZero paper [6]. The generation of training data through self-play is a slow process, thus an alternative metrics to evaluate engine performance prior to achieving measurable Elo score is needed.

6.5.2 Proposed metric

Proposed metric utilizes Stockfish chess engine to rank all possible moves based on their strength. This metric calculates the average percentage of moves played according to Stockfish during the game. For instance, if there are ten possible moves from a given position and the engine plays the second best move according to Stockfish, the score of this move would be 2/10 corresponding to 20%. The overall score would then be determined by calculating the average of the scores of each move of a game.

6.5.2.1 Eliminating deterministically selected moves

There is one factor that may interfere with the proposed metric - evaluation of deterministic positions in the endgame. The primary aim is to assess the trainable component of the engine, which contains the model. Evaluating a terminal node as a mate and selecting the corresponding move does not necessarily imply that the engine has improved as a result of training. The impact of such evaluations would be greater in shorter games than in longer ones, thereby interfering the results. To address this issue, the Stockfish chess engine was employed to search for a forced mate. Once a forced mate was found, any subsequent moves weren't included in the evaluation process.

6.5.2.2 Using Stockfish as an evaluator

At first glance, it may seem problematic to use the Stockfish chess engine to evaluate a reinforcement learning-trained engine, as these engines may use different approaches to gameplay. Consequently, the reinforcement learning-trained engine may improve its performance without necessarily increasing the proposed metric's percentage scores. Although this objection may be relevant in the later phases of training, where Elo is clearly the winning metric, it is not a problem in the initial stages of training. At this point, the engine is so weak that it is evident which moves are bad and to what extent. For example, when the engine admits to move its king to the enemy's last rank, any engine, including Stockfish, would not suggest such a move. Althout Stockfish cannot represent absolute truth, it is feasible to use a much stronger engine for evaluation of much weaker model.

6.5.2.3 Making metrics smooth

The raw data generated using evaluation during training process are very noisy, so in order to see some trend and have representative results, averaging of result over 20 - 50 steps is necessary.

6.5.3 Experiments

There has been 2 experiments made in the reinforcement learning part of the thesis.

First experiment with ResNet18 and expansion of 30 nodes in average. Number of expansions was randomly changed to weaken one player and to diversify the dataset, so in extreme cases number of expansions could even be 15 - 45. One step took about 30 minutes on a modern CPU. Results of first experiment are as follows:

Method	Result (%)	Training steps
Constant reward - no QSM	50	100
Constant reward	47.1	220
Linear reward	44.5	220

Table 6.1 Results of first experiment

Constant reward with no QSM utilized was still on 50% after 100 steps. Constant reward with quicker mate seeking around 47.1% and linear reward was best with score of 44.5% Constant and linear reward methods were tested on 220 training steps, which corresponds to 5 days of training.

Second experiment involved ResNet50 and expansion of 50 nodes, in extreme cases even 25 - 75. One step took over an hour on modern CPU, which corresponds roughly to 10 days of training. Results of this experiment include a graph of whole training process, where models were evaluated after every training step. (x axis)

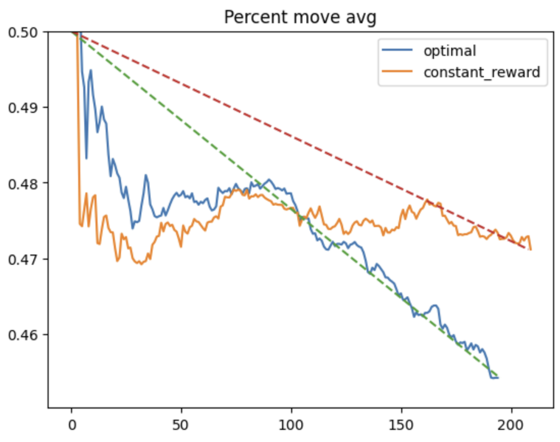


Figure 6.2 Result of second experiment

6.5.4 Conclusion

The results of constant reward with no QSM utilized are in line with expectations. A similar effect could be achieved by increasing the number of nodes for expansion. Therefore, it is not expected to be something interesting to study.

On the other hand, a linear reward showed some interesting results. Although 200 training steps are still at the very beginning of the training process [1], and there is still a lot of randomness present, the results look very promising in terms of accelerating the training process in

[1]compared to AlphaZero's 700,000 steps [6]

both experiments. It is still too early to claim whether it would outperform a constant reward ultimately, but it opens up an interesting hypothesis for further investigation.

6.6 Time comparison of learning processes

The reinforcement learning training process is considerably slower compared to supervised learning one. The reasons for this include differences in the numbers of positions required to achieve the same performance threshold, as well as differences in the time required to obtain these positions.

6.6.1 Dataset inconsistency

One reason why reinforcement learning is very slow is due to dataset inconsistencies, which are much more significant compared to those in a supervised dataset.

There could be a lot of dataset inconsistency in reinforcement learning data. In this case, dataset inconsistency means pure dataset inconsistency from the definition: the same input data has different labels. Typical examples are are mostly positions from the opening which led to a win in one game but a loss in another. Another example is a position that is clearly winning, but is labeled as a loss, and the opposite scenario - a position that is clearly losing, but is labeled as a win. Consequently, in the early stages of the game, the engine learns from data that is not very representative. To achieve representative data, some strategy has to be found, and it takes training effort to gradually take over other strategies. This simply means that a significant amount of time the engine learns random play and it requires to repeat already discovered strategies many times to start using a new one.

6.6.2 Dataset generation

There are multiple reasons, why reinforcement learning process is so costly. Not only the dataset must be larger for the engine to extract knowledge, but there is also no practical use for older data, but also generating such data is a time-consuming process. The engine relies on neural networks, which are computationally demanding to use. To obtain a position in the dataset, a full game must be completed to assign the target variable. For each position, a move has to be made using the neural network-based engine, which is comparatively slow compared to other engines such as Stockfish.

6.6.3 Implementation

This implementation presents a minimalistic approach capable of studying chess engines from a data-science perspective. It is a straightforward Python implementation A, where nodes in the game tree are expanded sequentially. To expand the next node, the maximum node from all expanded nodes is selected. However, practical experiment was not able to benefit from GPU utilization. Transferring a single board position to the GPU RAM and then back introduces significant overhead, making it even slower than using solely CPU. To benefit from GPU acceleration, positions would have to be evaluated in larger batches, and results redistributed back. Additionally, the entire engine would need to be implemented more efficiently and massively distributed over multiple clusters in order to complete the entire training process.

6.6.4 Comparison to AlphaZero and LeelaChessZero

In AlphaZero project, authors were able to make 700 000 training steps of 4096 positions in 9 hours [6], which corresponds to 88 493 board positions in second. In comparison implementation utilized for first practical experiment generated about 3 positions per second. It is clearly not enough to train the engine to a measurable Elo range. Reinforcement learning is still in the 0.03% of the training process compared to AlphaZero project [2]. AlphaZero was still rated by Elo 0 after taking such small amount of steps [6]. So it was crucial to find another way for evaluation of engine's performance was needed.

Another example of an engine with such high performance is an open-source variant of AlphaZero called LelaChessZero which is massively parallelized over high number of contributors who use either Google cloud or their own computers [22]. As of December 2022, LeelaChessZero has played over 1.5 billion games against itself, playing around 1 million games every day, and is capable of play at a level that is comparable to Stockfish.

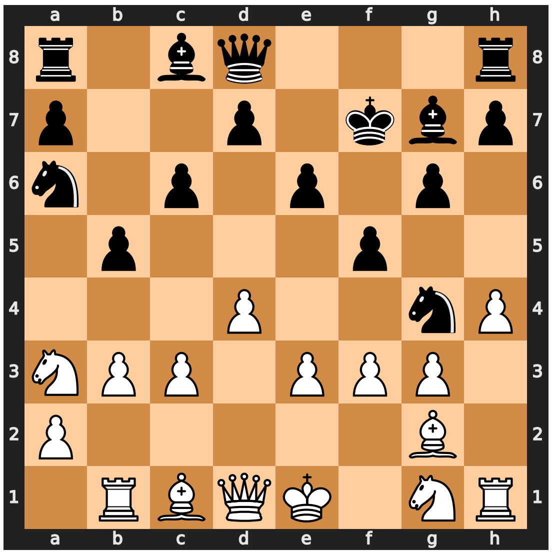
In order to achieve performance comparable to that of the best engines, an effective and highly distributable implementation, as well as a significant amount of computational resources, are required. To achieve such an efficient and highly distributed program, the thesis would need to focus on different domains of computer science. However, since an additional evaluation metric was proposed to supplement the unmeasurable Elo range, the thesis can still maintain its focus on a data science perspective without requiring an implementation that is highly effective and distributable.

[2] 220 / 700 000 = 0.000314

6.7 Achieved reinforcement learning performance

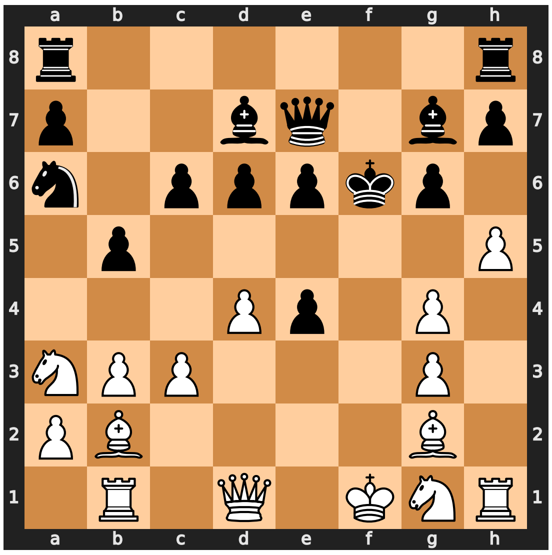
While the current training process is only at 0.03% compared to AlphaZero [6], and the gameplay is still much closer to ”random gameplay” than to that of an average beginner, some patterns can already be observed. Therefore, it is worthy of comment on these patterns that have been learned so far.

The expert who has been given random and reinforcement self-play to distinguish random play from self play of trained engine, was able to correctly recognize trained engine and analyzed its game as follows:



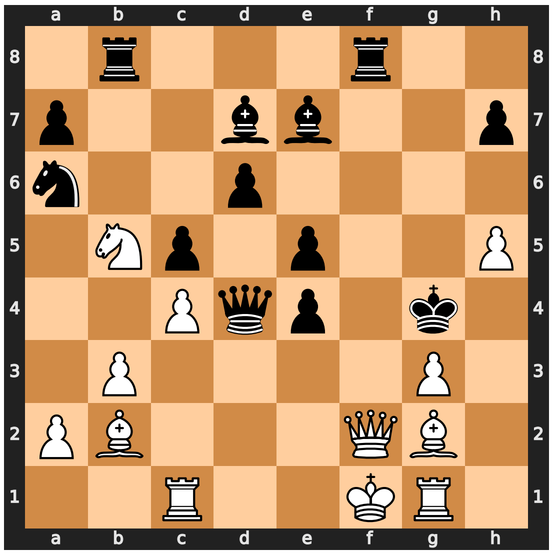
”We can see at least two patterns - it is desirable to develop knights (although they could be placed on more central squares), bishops and other pieces are developed, and pawns do not tend to be pushed forward without reason, thus making it possible to first shelter the king or make some space for one's pieces. This often does not happen, but the positioning of pawns looks quite natural.” [21]

Figure 6.3 Pawn structure



”Once the pieces enter the opponent's half of the board, the developing pattern continues, but single moves tend to be more ”random”. I do not mean this in the sense that they do not exhibit something which a chess player can recognize as a plan or motive, but even if they do” [21]

Figure 6.4 Middle game



"They disregard other aspects of the position. In the position above, black attacks the queen but moves his own queen to a square protected by 2 other white pieces, the bishop on b2 and the knight on b5. So the move Qd4 makes sense, but is still rather illogical." [21]

Figure 6.5 Recognizable pattern of a strategy

6.7.1 Observations

Finally, there are two noteworthy observations in relation to reinforcement learning stemming from these experiments.

An intriguing observation related to reinforcement learning is that the engine learns from the end of the game to the beginning, which contrasts with the chronological approach taken by human players.

Humans tend to first learn opening strategies, followed by middle-game tactics, and finally endgame techniques, as endgame strategies are less critical if the earlier stages have gone well, or the endgame is irrelevant due to mistakes in earlier position.

In contrast, during the early stages of reinforcement learning process, the moves made during the opening have a negligible impact on the outcome of the game, as the endgame is far away and the situation can change multiple times. However, the engine that always wins does so by checking the opponent's king, thus this is the first strategy that the engine is able to learn and is very frequently used, even if it often does not deliver a mate.

At this early stage of the training, the engine lacks knowledge about the value of pieces, if there is any value and if so, what the value is. Thus, the first piece that it recognizes as important is the king, which becomes the central point on the board while everything else is still unimportant. To learn strategies for the middle game, the reinforcement learning engine must gain experience and recognize the importance of protecting the king, as this makes it more difficult for the opponent to check. Therefore, other pieces also become important to protect the king. However, during this early stage of the training, the value of the other pieces remains negligible.

The second finding - it is easier to recognize patterns in the openings and endgames - suggests that learned strategies may be more generalizable, in contrast to almost randomly generated boards of the middle game. It is possible that modifying a strategy for the opening or endgame could be simpler than in middle game, as the learned patterns and strategies might be more easily applied to similar board positions. However, during the middle game, the number of features may increase, and if trainable parameters of the model are still close to their random initializations, it is possible that actions might be more likely to be randomly generated rather than being learned strategies. This could be due to the increased complexity of the game during the middle game and so it might make it more challenging to apply the learned strategies.

6.8 Conclusion

Although reinforcement learning can be much slower than supervised learning, it is still considered a superior method for maximizing performance.

Reinforcement learning provides the model with an opportunity to learn optimal heuristics that are tailored to its potential capabilities. This represents a significant advantage over other learning methods where the model may be forced to learn non-intuitive patterns from brute-force obtained data.

To address the challenge of slow learning, two methods have been proposed that accelerate the reinforcement learning process without relying on supervision or the incorporation of any human-like information. Although these methods are in the initial stages of training, the results appear promising so far.

Chapter 7
Conclusion

According to the stated goals of the thesis, a neural network-based engine has been developed. This engine does not rely on expertly designed heuristics and is capable of learning solely through the provision of pre-existing training data or solely through self-play.

A method enabling the rapid generation of datasets from the Stockfish chess engine was proposed. While subsequent findings suggest that employing an algorithmically generated dataset through a brute-force engine like Stockfish may not be the optimal approach for training neural networks intended for use in a chess engine, this generation method laid the foundation for subsequent practical experiments that would not have been feasible with human-generated data.

The MiniMax method and the output of a single neural network's forward pass were studied, as well as the relationship between the ResNet type and the depth of the generated dataset. Practical experiments revealed that both of these topics are closely interconnected, and the results were indeed interesting. It was demonstrated that neural networks are significantly limited in terms of deep analysis, regardless of the complexity of the model employed. Despite the inability of neural networks to find and utilize information hidden in the game tree, the opposite phenomenon has been found. Specifically, it was shown, that in some special cases, the MiniMax algorithm requires a deep game tree to be constructed in order to reveal even clearly visible patterns when evaluating a chess board position.

Finally, with the substantial support of the theoretical part of thesis and a comprehensive examination of previous solutions, coupled with the knowledge gained from the supervised learning segment, a satisfactory amount of information was acquired to compare the AI mechanisms responsible for training both methods. The advantages and disadvantages of both approaches were outlined, and the game engine was trained with both methods, using a comparable amount of time. The final games generated were algorithmically and expertly evaluated. Although the reinforcement learning approach proved to be a slow process, novel metrics were proposed to evaluate the training process even outside of the measurable Elo range. This allowed the evaluation and comparison of a novel training method, which has been proposed to accelerate the training process without incorporating any human-like information.

Chapter 8
Discussion

Although the game of chess has been extensively studied in the field of computer science, I have been fortunate enough to explore new areas and study topics for which I found no prior information, for which I am deeply grateful, because these discoveries were unexpected and added an element of excitement to the learning process.

The connection of MinMax algorithm and output of neural network forward pass is indeed interesting. It can provide better insight into how neural networks operate and find best usage for them in connection to the game of chess.

This insight can help with selection of training data for neural networks. While using dataset generated from Stockfish has yielded some results, it may not be the most optimal method for training since the data is generated through an evaluation of deep game tree and lacks the intuitive decision-making process seen in human games. Thus, a better approach may be to use open data sets, such as those available on Li-chess.

This observation could prove to be particularly important for understanding projects like Maia [13], which rely solely on a single forward pass. By identifying the specific situations in which neural networks may fail, we can better anticipate and address potential limitations in these types of projects.

The results obtained from the project are quite surprising, with the neural network being able to match 50% of the moves made by Stockfish at a depth of 6. Although the final engine may be weak, likely due to suboptimal training data, this was not the primary goal of the thesis. Nonetheless, the insights gained through this project are valuable and worth the effect of inability to create strong chess engine.

While the supervised and reinforcement learning aspects of the thesis may appear quite distinct from each other, the supervised learning phase was a crucial step prior to the more advanced reinforcement learning phase. The supervised learning stage provided valuable insights into the potential applications of neural networks in a chess engine, and enabled quick testing of model capabilities, implementation, and hyperparameter tuning. As a result, this phase laid the groundwork for the more advanced reinforcement learning phase.

The supervised learning phase of the thesis provided important insights into the primary benefits of reinforcement learning approach. Specifically, it demonstrated that neural networks then have the ability of learning optimal heuristics that are uniquely suited to their capabilities and are not constrained by the limitations in the training data. Thus making reinforcement learning best method for maximizing performance, despite long training process.

Research and experimentation have made it possible to gain a deeper understanding of the reinforcement method and its potential applications. Two practical experiments were conducted to test the effectiveness of proposed innovative linear reward method, revealing that it can significantly accelerate the beginning stages of the training process.

While the Python implementation used in this thesis was not optimal, there is surely a lot of room for improvement in the future. One potential avenue for further exploration is the use of ready-made open source solutions, such as LeelaChessZero, which may offer more efficient framework for testing linear reward method. This is an area of research planed for further investigation, as the initial results looks promising so far.

Appendix A
Implementation

Main purpose of this thesis was data-science research in the chess domain and to satisfy this goal, many various experiments had to be implemented and later abandoned. It was never an intention to deliver a matured code base, which allowed to focus on quick testing of various ideas and quality of research.

The entire implementation has been developed utilizing the Python programming language, with the aid of high-level libraries. In particular PyTorch [23] providing neural networks, binary heap [24] and Sklearn [25], Pandas [26], Numpy [27] and Matplotlib [28] for data analysis and visualization. Python chess library [29] for auxiliary chess related needs and of course Stockfish [11] chess engine.

Folder with source code consist of multiple .py and .ipynb files. Python files represents the core of the project. Most important python files are: generator.py - for generating the supervised and reinforcement dataset, encoder.py and decoder.py for transforming input and output in/out of neural network, models.py - for creating and transforming ResNet model to suit required purpose, fit.py - for implementing training the neural network engine.py - for deploying model in the chess engine, then multiple files for evaluation metrics. Jupyter notebook files are then used for analysis. Then project contains multiple testing files checking especially correctness of data transformations between moves, chess boards neural network.

A.1 Summary

The Git repository comprises three branches and consists of approximately 400 commits spanning period since October 2022 until May 2023. The core of the project in Python files sums to approximately 2500 lines of code. Core python files and also files for visualization of measured results of the training are provided. The dataset for the training is not included due its size (60GB), but it can be easily generated.

Bibliography

1. FOUNDATION, Wikimedia. *Deep Blue (chess computer) - Wikipedia* [https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)]. 2023. (Accessed on 02/20/2023).
2. FELDMANN, Rainer. Computer chess: Algorithms and heuristics for a deep look into the future. In: PLÁŠIL, František; JEFFERY, Keith G. (eds.). *SOFSEM'97: Theory and Practice of Informatics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 1–18. ISBN 978-3-540-69645-2.
3. SUTTON, Richard S.; BARTO, Andrew G. *Reinforcement Learning: An Introduction*. 2nd. Cambridge, Massachusetts; London, England: The MIT Press, 2018. ISBN 9780262039246.
4. SILVER, David; HUANG, Aja; MADDISON, Chris J; GUEZ, Arthur; SIFRE, Laurent; VAN DEN DRIESSCHE, George; SCHRITTWIESER, Julian; ANTONOGLOU, Ioannis; PANNEERSHELVAM, Veda; LANCTOT, Marc, et al. Mastering the game of Go with deep neural networks and tree search. *nature*. 2016, vol. 529, no. 7587, pp. 484–489.
5. SILVER, David; SCHRITTWIESER, Julian; SIMONYAN, Karen; ANTONOGLOU, Ioannis; HUANG, Aja; GUEZ, Arthur; HUBERT, Thomas; BAKER, Lucas; LAI, Matthew; BOLTON, Adrian, et al. Mastering the game of go without human knowledge. *nature*. 2017, vol. 550, no. 7676, pp. 354–359.
6. SILVER, David; HUBERT, Thomas; SCHRITTWIESER, Julian; ANTONOGLOU, Ioannis; LAI, Matthew; GUEZ, Arthur; LANCTOT, Marc; SIFRE, Laurent; KUMARAN, Dharshan; GRAEPEL, Thore; LILLICRAP, Timothy; SIMONYAN, Karen; HASSABIS, Demis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*. 2018, vol. 362, no. 6419, pp. 1140–1144. Available from DOI: 10.1126/science.aar6404.
7. FRANCOIS-LAVET, Vincent; HENDERSON, Peter; ISLAM, Riashat; BELLEMARE, Marc G.; PINEAU, Joelle. *An Introduction to Deep Reinforcement Learning*. Foundations and Trends in Machine Learning, 2018. Available from DOI: 10.1561/2200000071.
8. LU, Yulong; LU, Jianfeng. A Universal Approximation Theorem of Deep Neural Networks for Expressing Probability Distributions. In: LAROCHELLE, H.; RANZATO, M.; HADSELL, R.; BALCAN, M.F.; LIN, H. (eds.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2020, vol. 33, pp. 3094–3105. Available also from: https://proceedings.neurips.cc/paper_files/paper/2020/file/2000f6325dfc4fc3201fc45ed01c7a5d-Paper.pdf.
9. SHAFIQ, Muhammad; GU, Zhaoquan. Deep Residual Learning for Image Recognition: A Survey. *Applied Sciences*. 2022, vol. 12, no. 18. ISSN 2076-3417. Available from DOI: 10.3390/app12188972.

10. FOUNDATION, Wikimedia. *Stockfish (chess) - Wikipedia* [`https://en.wikipedia.org/wiki/Stockfish_(chess)`]. 2023. (Accessed on 04/20/2023).

11. THE STOCKFISH DEVELOPERS (SEE AUTHORS FILE). *Stockfish.* [N.d.]. Available also from: `https://github.com/official-stockfish/Stockfish`.

12. THE STOCKFISH DEVELOPERS (SEE AUTHORS FILE). *SF NNUE · Issue #2728 · official-stockfish/Stockfish · GitHub* [`https://github.com/official-stockfish/Stockfish/issues/2728#issuecomment-650523408`]. [N.d.]. (Accessed on 04/20/2023).

13. MCILROY-YOUNG, Reid; SEN, Siddhartha; KLEINBERG, Jon; ANDERSON, Ashton. Aligning Superhuman AI with Human Behavior: Chess as a Model System. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery &amp; Data Mining.* Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 1677–1687. KDD '20. ISBN 9781450379984. Available from DOI: `10.1145/3394486.3403219`.

14. MCILROY-YOUNG, Reid; SEN, Siddhartha; KLEINBERG, Jon; ANDERSON, Ashton. *Introducing Maia: a Human-Like Chess Engine | Computational Social Science Lab* [`http://csslab.cs.toronto.edu/blog/2020/08/24/maia_chess_kdd/`]. [N.d.]. (Accessed on 04/01/2023).

15. LICHESS. *maia9 : Activity • lichess.org* [`https://lichess.org/@/maia9`]. [N.d.]. (Accessed on 12/12/2022).

16. COMUNITY, lichess. *Is it true that lichess ratings are somewhere around FIDE rating+ 400 • page 1/2 • General Chess Discussion • lichess.org* [`https://lichess.org/forum/general-chess-discussion/is-it-true-that-lichess-ratings-are-somewhere-around-fide-rating-400`]. 2021. (Accessed on 04/24/2023).

17. LICHESS. *maia1 : Activity • lichess.org* [`https://lichess.org/@/maia1`]. [N.d.]. (Accessed on 12/12/2022).

18. OSHRI, Barak; KHANDWALA, Nishith. *Predicting moves in chess using convolutional neural networks.* Stanford University Course Project Reports-CS231n, 2016.

19. LAI, Matthew. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549.* 2015.

20. SAMADI, Mehdi; AZIMIFAR, Zohreh; JAHROMI, Mansour Zolghadri. Learning: An effective approach in endgame chess board evaluation. In: *Sixth International Conference on Machine Learning and Applications (ICMLA 2007).* IEEE, 2007, pp. 464–469.

21. RÝDL, Jiří. *Personal consultation with expert* [`https://ratings.fide.com/profile/346390`]. 2023.

22. *Lc0 training. - Leela Chess Zero* [`https://lczero.org/blog/2018/10/lc0-training/`]. [N.d.]. (Accessed on 04/20/2023).

23. PASZKE, Adam; GROSS, Sam; MASSA, Francisco; LERER, Adam; BRADBURY, James; CHANAN, Gregory; KILLEEN, Trevor; LIN, Zeming; GIMELSHEIN, Natalia; ANTIGA, Luca; DESMAISON, Alban; KOPF, Andreas; YANG, Edward; DEVITO, Zachary; RAISON, Martin; TEJANI, Alykhan; CHILAMKURTHY, Sasank; STEINER, Benoit; FANG, Lu; BAI, Junjie; CHINTALA, Soumith. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems 32.* Curran Associates, Inc., 2019, pp. 8024–8035. Available also from: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

24. RV, Ramesh. *binary-heap · PyPI* [`https://pypi.org/project/binary-heap/`]. [N.d.]. (Accessed on 04/28/2023).

25. PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830.

26. TEAM, The pandas development. *pandas-dev/pandas: Pandas*. Zenodo, 2020. Version 2.0. Available from DOI: `10.5281/zenodo.3509134`.

27. HARRIS, Charles R.; MILLMAN, K. Jarrod; WALT, Stéfan J. van der; GOMMERS, Ralf; VIRTANEN, Pauli; COURNAPEAU, David; WIESER, Eric; TAYLOR, Julian; BERG, Sebastian; SMITH, Nathaniel J.; KERN, Robert; PICUS, Matti; HOYER, Stephan; KERKWIJK, Marten H. van; BRETT, Matthew; HALDANE, Allan; RÍO, Jaime Fernández del; WIEBE, Mark; PETERSON, Pearu; GÉRARD-MARCHANT, Pierre; SHEPPARD, Kevin; REDDY, Tyler; WECKESSER, Warren; ABBASI, Hameer; GOHLKE, Christoph; OLIPHANT, Travis E. Array programming with NumPy. *Nature*. 2020, vol. 585, no. 7825, pp. 357–362. Available from DOI: `10.1038/s41586-020-2649-2`.

28. HUNTER, John D. Matplotlib: A 2D graphics environment. *Computing in science & engineering*. 2007, vol. 9, no. 3, pp. 90–95.

29. FIEKAS, Niklas. *python-chess: a chess library for Python — python-chess 1.9.4 documentation* [`https://python-chess.readthedocs.io/en/latest/`]. [N.d.]. (Accessed on 04/28/2023).