

# Chess Engine Using Deep Reinforcement Learning

Kamil Klosowski

916847

May 2019

## Abstract

Reinforcement learning is one of the most rapidly developing areas of Artificial Intelligence. The goal of this project is to analyse, implement and try to improve on AlphaZero architecture presented by Google DeepMind team. To achieve this I explore different architectures and methods of training neural networks as well as techniques used for development of chess engines.

Project Dissertation submitted to Swansea University  
in Partial Fulfilment for the Degree of Bachelor of Science

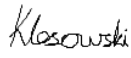


Department of Computer Science  
Swansea University

## Declaration

This work has not previously been accepted in substance for any degree and is not being currently submitted for any degree.

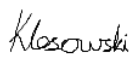
May 13, 2019

Signed: 

## Statement 1

This dissertation is being submitted in partial fulfilment of the requirements for the degree of a BSc in Computer Science.

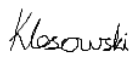
May 13, 2019

Signed: 

## Statement 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

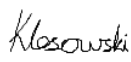
May 13, 2019

Signed: 

## Statement 3

I hereby give consent for my dissertation to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

May 13, 2019

Signed: 

# **1 Acknowledgment**

I would like to express my sincere gratitude to Dr Benjamin Mora for supervising this project and his respect and understanding of my preference for largely unsupervised work.

# Contents

<b>1</b>	<b>Acknowledgment</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>5</b>
2.1	Motivation behind the project . . . . .	5
2.2	Project Aims . . . . .	5
<b>3</b>	<b>Background and related work</b>	<b>6</b>
3.1	The beginnings . . . . .	6
3.2	First attempts . . . . .	6
3.3	Artificial Intelligence and the role of Checkers . . . . .	7
3.4	Deep Blue and Deep Thought . . . . .	7
3.5	AlphaGo . . . . .	8
3.6	Modern Chess Engines . . . . .	8
3.7	Perceptron . . . . .	9
3.8	Early neural networks . . . . .	10
3.9	Convolutional neural networks . . . . .	10
3.10	Residual neural networks . . . . .	11
3.11	Reinforcement and Deep Reinforcement Learning . . . . .	12
3.12	Bitboards . . . . .	13
<b>4</b>	<b>Technological choices</b>	<b>14</b>
4.1	Languages, Frameworks, and Hardware . . . . .	14
4.2	Architecture . . . . .	15
<b>5</b>	<b>Methodology</b>	<b>16</b>
<b>6</b>	<b>Risks</b>	<b>17</b>
<b>7</b>	<b>Scheduling and Software Life Cycle</b>	<b>18</b>
<b>8</b>	<b>Implementation</b>	<b>19</b>
8.1	Prototype . . . . .	19
8.2	Graphical interface . . . . .	19
8.3	Chess engine - Validation . . . . .	21
8.4	Reinforcement Learning approach . . . . .	22
8.5	Supervised Learning approach . . . . .	24
8.6	Interfacing between subsystems . . . . .	25
<b>9</b>	<b>Evaluation</b>	<b>25</b>
<b>10</b>	<b>Testing</b>	<b>26</b>
<b>11</b>	<b>Conclusions</b>	<b>27</b>

## 2 Introduction

### 2.1 Motivation behind the project

Ever since I have started programming, I have been interested in chess engines. They have always impressed me with the performance the authors were able to achieve and created in me a long-standing ambition to create one myself. During my programming "career" my interests have gradually shifted as I have explored different fields of computer science but I have never abandoned my plan to create a chess program. Recently I have got heavily involved in a field of data science and spent a considerable amount of my time exploring different achievements and breakthroughs of the past decade. The recent explosion in the availability of data and computational power, allowed a relatively old concept of neural networks to resurface and finally be utilised to its full potential. The last few years in the field of Artificial Intelligence are considered by many experts to be the beginning of the fourth industrial revolution[20][19]. Ever since the AlexNet[10], a convolutional neural network, placed first in the ImageNet Large Scale Visual Recognition Challenge by achieving top-5 error rate of 15.3% which outperformed the second place contender by 10.8%, neural networks continued to outperform traditional methods in many applications. In 2017, DeepMind published a highly influential paper about a generalized neural network architecture that was able to outperform top engines in Chess, Shogi and Go by learning only from self-play[21]. Reading this article inspired me to finally fulfil my ambition and create a chess engine based on the presented architecture. The fact that DeepMind is yet to publicise the engine they have used, combined with the vagueness of the description of used techniques, makes this a perfect project for this dissertation.

### 2.2 Project Aims

The primary aim of this project is to create a chess engine, utilising artificial neural networks, being able to compete with humans and improve from continuous self-play training. To achieve this, I will implement an architecture proposed in a Google's DeepMind's AlphaZero paper [21]. This objective can be considered very ambitious and poses a high possibility of failure. Despite this, the project offers a great opportunity for experimentation and, even if the goals are not achieved, it can present a variety of interesting and potentially innovative insights.

## 3 Background and related work

### 3.1 The beginnings

One of the earliest works on the topic of computer chess is a paper published by Claude E. Shannon [6] in 1949 where he theorized about the feasibility of a computer routine for a “modern” general purpose computer which would allow it to play chess. Possible problems to which a said routine could be generalized were also outlined, the list included language translation, military strategy, music creation, and even logical deduction. Shannon proposed two different strategies for the game, first based on John Von Neumann’s Minimax Theorem [13] using brute-force evaluation of the game tree and a second one which decreases the size of the search tree by introducing a “plausible move” evaluation. He explained how the first approach is not a viable option because of the enormous amount of possible games that would have to be evaluated which he estimated to be at least  $10^{120}$  which is today known as a “Shannon number”. If every atom in the universe ( $10^{80}$ ) were to compute a single static evaluation every nanosecond since the birth of the universe ( $10^{18}$ ) we would still be over 20 orders of magnitude short from solving chess. This fact alone makes any attempts of solving chess using brute-force ultimately futile. The second approach was inspired by observing human grandmasters which is closely connected to the scope of this project. In order to create an evaluation function, traditionally, multiple handcrafted features are used. The list includes material point values, positional imbalances, piece mobility, pawn structure, king safety and many other features usually extracted with a help of a human chess expert [21]. This approach creates an obvious limitation, program created in this manner will never be able to learn new features on its own thus restraining the possible strength of the program.

### 3.2 First attempts

In 1948 Alan Turing created an algorithm called Turochamp which is the earliest known computer game algorithm. He described it in an article called “Chess” originally published in a book “Faster than thought” [23]. Although it was never implemented, Turing tested it by executing the algorithm “by hand” but the game was lost. Not long after Turing’s publication, a student of his, Dietrich Prinz created an actual program to solve “Mate-in-Two” problems. The reason he was not able to create a full-scale chess engine was the lack of computing power provided by the available to him Ferranti Mark 1 computer at Manchester University. Said machine was able to evaluate around 30 positions per minute which was far from sufficient to play a full game of chess. To put this into perspective, modern chess engine Stockfish 8 is able to evaluate  $7 * 10^7$  positions per second using a 64 thread CPU [21]. Prinz described his results in a chapter “Robot Chess” published in a “Computer Chess Compendium” [14].

### 3.3 Artificial Intelligence and the role of Checkers

The term “Artificial Intelligence” is said to be first created during the Dartmouth Conference in 1956 [12]. This month-long event attracted many leading figures in the field of machine learning, including Nathaniel Rochester and Arthur Samuel, IBM researchers who would later become part of the team that created the first Checkers AI program that ran on the IBM 701. The choice of this game wasn’t accidental, it provided the same type of challenge that Chess and Go did, while offering manageable size of the game tree. The program was able to reach a better-than-human level of play after only 10 hours of computations. Although it was based on Minimax, the researchers augmented the algorithm with machine-learning strategies. One of them was to create a database of already visited branches of the game tree that was constantly updated during the game which significantly decreased the computing load. Additionally to replay memory, the algorithm was equipped with a dynamic evaluation function which was updated to minimise the estimation error after finishing each game. This approach is similar to currently used reinforcement learning techniques that will be used in this project. Samuel summarised the observations of his team and the techniques they used in his summary paper on machine learning in checkers[17].

Today the game of checkers has been ”solved” [18]. Jonathan Schaeffer has proven that perfect play always results in a draw. Calculations took over a decade, using from 50 up to 200 computers running the computations non-stop in parallel. The product of this experiment is an endgame database containing 39 trillion positions. He also released an updated version of his original checkers engine called Chinook, currently the engine cannot be beaten and the best achievable result is a draw. In his paper, Schaeffer also concluded that the number of checkers moves is roughly a square root of those in chess and that a significant breakthrough like quantum computing would be required to solve chess even weakly.

### 3.4 Deep Blue and Deep Thought

The victory of Deep Blue over world champion Gary Kasparov in 1997 was perhaps the most significant event in the history of Artificial Intelligence up until very recently. This game was the culmination of several iterations of chess engines and over a decade of work put into achieving the success. It began with a rivalry between two teams of researchers and their chess computers, Hitech and ChipTest. Both teams attempted to create computers consisting of, specifically designed for chess evaluation, integrated circuits. Hitech soon became the first chess computer to beat a human grandmaster in a tournament, winning against Arnold Denker with a score of  $3\frac{1}{2}$  out of 4 games. ChipTest was capable of evaluating 500,000 positions per second at the time. Behind the successes of both computers stood Feng-Hsiung Hsu who initially worked on Hitech but left, due to disagreement about what he considered to be an architectural flaw in the design, and started his work on ChipTest. Technologies he developed were later used to build Deep Thought, a computer able to evaluate 720,000 moves a second which won 1989 World Computer Chess Championship with a perfect score. Later that year, Deep Thought took on Gary Kasparov in 2 game match but lost both of the games. With freshly acquired backing from IMB the project was renamed to Deep Blue and vastly improved. In February 1996 it faced the Kasparov again and lost with a score 4-2 drawing 2 games

and winning 1. Single win was enough to convince researchers to further heavily upgrade the computer which gave it an unofficial nickname of "Deeper Blue". It was this version of the machine that faced Kasparov again in a historical match that changed the history of Artificial Intelligence. At the time capable of analysing  $10^7$  moves per second, Deep Blue won the match with a score of  $3\frac{1}{2} - 2\frac{1}{2}$

### 3.5 AlphaGo

Chinese game of Go is considered to be one of the hardest challenges of machine learning. With a board size of 19x19, typical game length of 200 moves and an average branching factor of 35, Go is inherently unsuitable for the methods used in other board game engines. Traditional Go engines possessed a level of play comparable to a novice human players. In 2016 DeepMind's Go engine[22] has achieved what few thought possible, it defeated Lee Sedol who is considered to be the greatest player of the past decade and 18 time world champion. In an event watched by over 200 million people, AlphaGo won a 5 game match with a score of 4-1. This also made it the first computer program to be granted a professional rank of 9th dan. This event was comparable to Deep Blue's victory in terms of impact on artificial intelligence field.

### 3.6 Modern Chess Engines

Despite significant breakthroughs in the field of computer chess and deep learning, engines based on artificial neural networks still are not doing as well as their more traditional counterparts. Currently, in CCRL 40/40 ranking [Fig. 1] in top 10, there is not a single engine utilising deep learning. Most relevant to this project entry, in said ranking, is Leela Chess Zero, an open source engine that is based on the original AlphaZero architecture. Leela became popular as it quickly increased in strength. Many strong grandmasters tested it and concluded that Leela's tactics are significantly different from those presented by engines based on MiniMax principles. It often favours positions that other engines evaluate as losing, which shows a certain level of what could be described as "human intuition". By allowing volunteers to contribute computational power to the project, Leela is continuously gaining in strength.

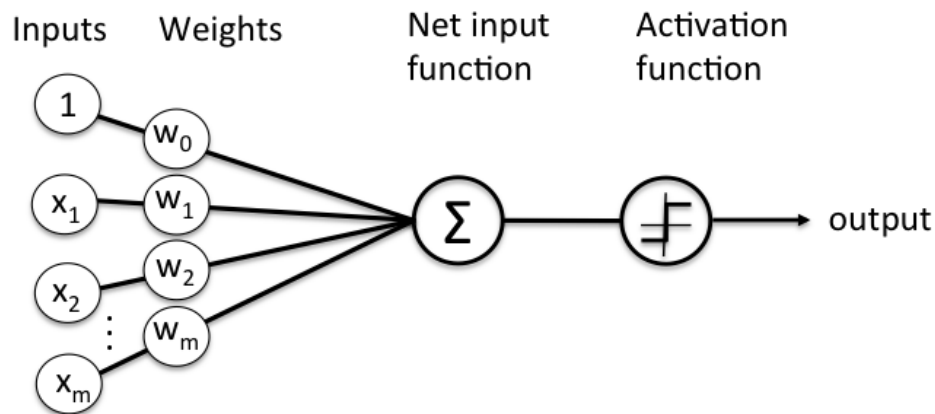
Rank	Name	Rating			Score	Average Opponent	Draws	Games
		Elo	+	-				
1	Stockfish 270918 64-bit 4CPU	3460	+28	-27	77.9%	-188.9	43.3%	457
2	Komodo 11.3.1 64-bit 4CPU	3402	+21	-20	70.6%	-140.5	48.6%	790
3	Houdini 6 64-bit 4CPU	3397	+15	-15	64.3%	-91.7	59.4%	1259
4	Fire 7.1 64-bit 4CPU	3324	+18	-18	60.2%	-64.2	57.0%	918
5	Ethereal 11.00 64-bit 4CPU	3305	+27	-27	45.4%	+25.8	67.6%	370
6	Deep Shredder 13 64-bit 4CPU	3284	+13	-13	49.9%	-0.4	62.1%	1859
7	Fizbo 2 64-bit 4CPU	3281	+16	-16	47.0%	+17.5	55.4%	1117
8	Xiphos 0.4 64-bit 4CPU	3276	+26	-26	41.0%	+53.8	63.2%	416
9	Andscacs 0.94 64-bit 4CPU	3275	+23	-23	44.1%	+36.1	56.3%	540
10	Booot 6.3.1 64-bit 4CPU	3272	+23	-23	44.3%	+34.1	57.0%	558

Figure 1: Current standings of CCRL 40/40 computer chess ranking. [24]



### 3.7 Perceptron

The development of neural networks started with a very simple idea of a perceptron. It was conceived by a psychologist Frank Rosenblatt who attempted to create a mathematical representation of the operations that neurons perform in a human brain [15]. This model in various modified forms now serves as a basis for all artificial neural networks. The perceptron consists of a number of inputs that represent connections to other neurons. Those connections have unique weights which model the strength of synaptic connections to the input neuron. The connections are then aggregated and passed to the activation function which based on a certain threshold determines if the neuron propagates the impulse down the network. The original perceptron also had a bias input which was always set to 1, this simplified the computation of various activation functions. Rosenblatt's work was based on previous research conducted by Warren McCulloch and Walter Pitts which proved that a system composed of several neural units can perform binary logical operations such as OR, AND and NOT[16]. This was believed to be a huge step forward in the development of artificial intelligence. Even with such a monumental step forward, a single factor was missing, an ability to learn. At the time the process in which the human brain learned and acquired information was largely unknown. Donald Hebb proposed an idea that the retention of information occurs when a repeatedly stressed neuron undergoes a process of growth on a synaptic level [2]. This makes the transition of electric impulses through that connection easier and activates the neuron faster. This is linked to the idea of weight in the perceptron.



**Schematic of Rosenblatt's perceptron.**

Figure 2: Perceptron schematic

The arrangement of perceptron units in layers, created a multi-layered perceptron which is the prototypal model of what we now know as a neural network. By applying this concept and recreating it with custom hardware, Rosenblatt created a machine capable of classifying shapes from a small 20x20 array of inputs, he called it Mark I Perceptron. The exact same model can now be easily recreated in modern machine learning libraries and it is often a gateway project when learning about artificial intelligence. The most popular dataset for this problem is MNIST which is a set 10000 samples of 20x20 pixel handwritten digits. Currently, the state of the art neural networks achieved 99.79% correctness of classification.

### 3.8 Early neural networks

The advancements made by Rosenblatt propelled a wave of research focusing on perceptrons and possible areas where they can be used. However, the research quickly stagnated when the researchers were unable to effectively train the networks. The complexity of calculations grew rapidly with the size of the network. The progress required a breakthrough which came in the form of a backpropagation algorithm. The algorithm was derived by several researchers at the beginning of 1960 decade and was implemented to run on a computer by Seppo Linnainmaa in 1976 [11]. Process of backpropagation was recursive and iterative which allowed for easier training and updating of weights inside the network. The idea was to input data into the neural network and compared the output with a predetermined label which was taken as truth. The error was then calculated and propagated backwards through the network while updating the weight parameters. This process is currently being used in state of the art neural networks. The confirmation of the viability of neural networks came in 1989 with the paper called "Multilayer feedforward networks are universal approximators" which proved that neural networks can be used to implement any function[9].

### 3.9 Convolutional neural networks

Convolutional neural networks were the next step in the development of neural networks, they shared many similarities with ordinary multi-layered perceptrons by using the same mechanisms for weights, biases and connections. The main difference was the structure of layers. Instead of using a 2-dimensional architecture, a 3-dimensional one is used. This change was necessitated by the fact that ordinary neural networks don't scale well with input size. For example, to input a 256\*256 RGB image into the network for the purpose of classification and the usage with computer vision programs, 196608 input neurons would have to be used. This equates to 256\*256 pixels \* 3 colour channels. Networks of this size are extremally hard to train and don't generalise well. The solution to this problem is a convolutional layer, it consists of a stack of 2-dimensional arrays, in this example 3 arrays the size of 256\*256. This input is then aggregated with the help of pooling. Pooling is a process of feature extraction by transforming the image into smaller subsamples of the same image. The image is then further sampled down into what's called feature maps. Each subsequent subsampling lets the network focus on finer details of the image. Convolutional networks usually end with a fully-connected layer followed by the output layer which size is determined by the number of categories to classify. This type of network proved to be the best known approach for image

classification although several challenges needed to be overcome. One of them was the difficulty of propagating an error through the network with a higher depth. Some of the convolutional networks can reach over a 1000 layers of depth which introduces the problem of vanishing gradients [8]. This problem is currently solved by using residual neural networks.

### 3.10 Residual neural networks

To fix the vanishing gradient problem, Microsoft researchers came up with a relatively simple solution that revolutionised the field of computer vision. They proposed an architecture where the layers of a network were separated into blocks. Each block consisted of two layers with batch normalisation and RELU activation. To the output of the second layer before the activation function, a skip connection was added. This special skip connection is simply an identity connection with output values of the previous residual block. This architecture ensures that all parts of the network receive input that isn't diminished by the previous layers. Using this approach, researchers were able to effectively train networks with 1202 layers [7]. They also argue that even though the error rate is slightly higher than that of 110-layer network, it is likely due to overfitting on a too small of a dataset. They also suggested the usage of strong regularization such as maxout or dropout to mitigate this problem.

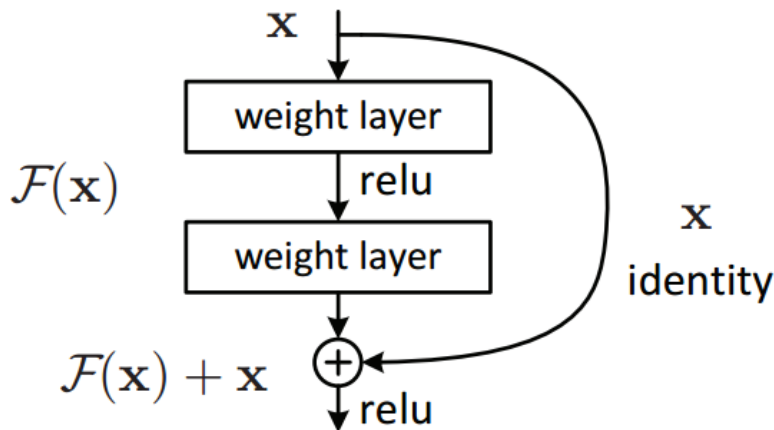


Figure 3: Residual network block

Since the original ResNet paper, other variations of this architecture have been proposed and demonstrated success and improvements over basic ResNet. One of the evolutions was introduced by Facebook Ai Research team in collaboration with UC San Diego researchers. It is called ResNeXt and uses parallel layers inside blocks. It has shown to be more robust and achieve better accuracy, however at the cost of computational complexity [25].

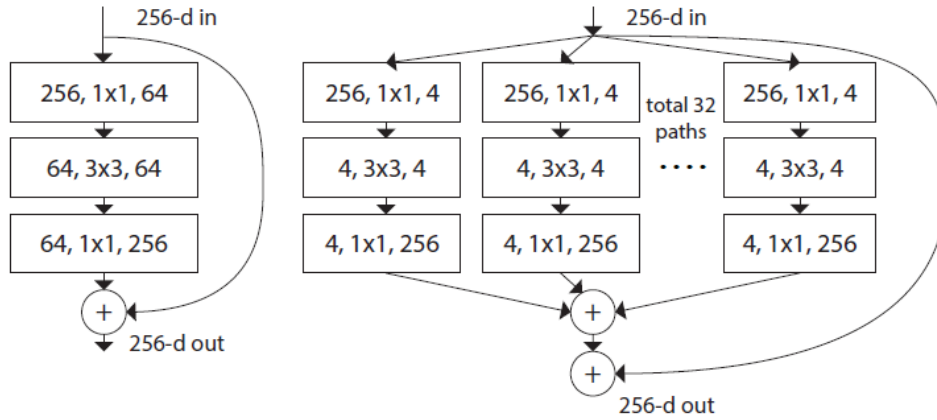


Figure 4: ResNeXt residual block

### 3.11 Reinforcement and Deep Reinforcement Learning

Reinforcement learning is an area of machine learning that focuses on training a certain agent to navigate a specified environment. It is a subset of unsupervised learning. The goal is to achieve a certain task without domain knowledge. At the beginning, the reinforcement agent usually starts with no knowledge of the domain. It acquires said knowledge by performing initially random actions, knowing only the desired result. This approach is particularly useful when the conditions that lead to a certain result are unknown. Currently, the reinforcement learning approaches are being employed with success in areas such as game artificial intelligence, self-driving cars and robotics. Inside the reinforcement learning, there are two main strategies, positive and negative reinforcement. Both of those strategies can be used in conjunction to achieve the goal. Using the example of self-driving cars, the positive goal would be to drive the highest number of miles without human intervention whereas negative reinforcement would be used when a car steers off the road or particularly heavy negative reward could be used when the car injures a human. A subarea which is currently receiving the most attention from the international research community is deep reinforcement learning. The main difference between classic and deep reinforcement learning is the usage of neural networks. The flexibility of the architecture of neural networks enabled this area of research to progress very rapidly. In theory, if one can specify the task and the conditions for both success and failure, a neural network agent can be trained to perform that task with sufficient computational power. This is precisely the approach that produced the international success of AlphaGo and Alpha Zero. One of the main disadvantages of this method is the need for extreme amounts of computational power. By looking at the data provided in the AlphaZero paper the authors inform us that the final version of the engine was trained using 5000 of Google's proprietary TPUs (Tensor Processing Units) for 400 hours. To put it in terms of consumer hardware, one 1st generation TPU has the computational capacity of approximately 10 Nvidia GTX1080ti[1] graphics cards (Fastest consumer graphics card available at that time). This means that the training computation used for that model equates to 2283 years of computational time using a

single GTX1080ti and around 4500 years using GTX1060 that I personally have access to. This enormous requirement for power is the reason that only the biggest technology companies are able to compete in this field.

### 3.12 Bitboards

Bitboards are binary data structures used to represent the state of a board game in the computer memory. Depending on the type of the game, different sizes are used. In the case of chess, a piece centric approach is used. Every piece type is represented by a 64-bit number, the presence of a positive bit indicates the position of a certain piece on the board. Two different bitboards are used to distinguish between the white and black pieces. A total of 12, 64-bit bitboards is needed to represent the piece positions in chess with additional data structures (which can also be bitboards) needed to represent the additional game state elements such as the castling, the number of moves without any progress and the draw condition.

Different kinds of mappings (Endianness) can be used to represent the piece positions on the bitboard, the most popular type of mapping is Little-Endian Rank-File mapping. In this mapping, the least significant bit represents the A1 square and the most significant represents the H8 square. As the name suggests the bit ordering follows the ranks first and then the files. Different types of mappings are equally viable and do not differ in the performance, the choice of the Little-Endian Rank-File mapping in this project is strictly due to preference.

Moves on bitboards are represented as bitmasks and executed using binary operations. Although there are methods to simplify the move process, most of the moves are not easily generalizable and require at least some 'hardcoding' which increases the complexity of this type of representation. Regardless of the added complexity, the use of bitboards is the most efficient representation method, both memory and computational complexity-wise.

## 4 Technological choices

### 4.1 Languages, Frameworks, and Hardware

The project is split into two distinct and independent parts, namely graphical interface implementing the Universal Chess Interface protocol to allow for visualising of the game state and move validation and the chess engine itself.

The first part uses Java with JavaFX framework. After considering multiple languages, I have come to the conclusion that Java is best suited to this task. The main reason for this choice is my familiarity with the language. There are other more efficient languages that could be used to accomplish this task such as C or C++ that would ensure higher performance for move evaluation. However, after considering all pros and cons, I have come to a conclusion that for the scope of this project, the time that would be saved during training step by selecting a more efficient programming language can be offset by a faster time of implementation which is significantly shortened by removing the need to research and learn a different language. As for the graphical interface framework, JavaFX was the most obvious choice, it is easy to use and I already possess a sufficient level of knowledge of it.

The second part which is the chess engine is going to be implemented using Python 3.X and TensorFlow framework. Python is the most popular programming language for machine learning and I am already proficient in it. TensorFlow is arguably the best deep learning framework with great support and is actively developed by Google engineers which makes it a perfect choice for this project. The availability of training material is great with plentiful books, videos and free online tutorials.

For the neural network training I will initially use a Nvidia GTX 1060 6GB, depending on the rate of convergence and the speed of the training I have also considered using Google Cloud which makes available for the general public, Google's proprietary Tensor Processing Units which are specifically designed for the use with TensorFlow AI accelerators that offer disproportionately higher performance compared to standard GPUs.

## 4.2 Architecture

### Graphical Interface

Simple graphical interface will be implemented using JavaFX to allow for quick visual inspection of the board state. In later stages user interface displaying the current position strength estimation, move history using algebraic notation, move and game controls (move withdrawal, resign, draw offering) and user menu, as well as game history will be implemented.

### Legal Move Evaluation

The move evaluation will be implemented using Little-Endian Rank-File mapping of bitboards and bitwise operations to provide sufficient performance and quick game tree search. Java 8 Streams will be heavily utilised to accomplish quick collections operations while building the game tree. Instead of traditional Alpha-Beta tree search, Monte Carlo Tree Search will be used which will be combined with the neural network evaluation.

### Domain knowledge

The only domain knowledge available to the engine in the initial part of the project will be the rules of chess. Usage of other chess specific features will be considered after achieving a working solution.

### Communication Layer

To combine the evaluation part of the engine with the graphical interface, Universal Chess Interface protocol will be implemented. This will allow for easy swapping of the graphical layers and will make it compatible with different commercial interfaces.

### Neural Network

The final neural network architecture will be heavily influenced by this of AlphaZero. However, as DeepMind's paper leaves out most of the details, the structure of the network will be subject to experimentation and decided after conducting more research.

### Component Decoupling

I aim to create a highly decoupled structure of the program. Due to the usage of different programming languages for different parts of the project, most of the communication between components will be achieved via a console interface. This architectural decision will create high interchangeability and different graphical interfaces will be possible to be connected to the engine itself. Testing will also be significantly simplified as each part of the program can be assessed independently.

## 5 Methodology

The software engineering methodology that I have chosen to use during this project is Extreme Programming and specifically scaled down to a single person team Personal Extreme Programming (PXP) proposed by R. Agrawal and D. Umphress [4]. The reasons for this choice are as follows:

1. XP can be scaled down to a single person team.
2. It is innately suited to changing requirements which might occur during the development of this project.
3. Release planning and release schedules allow for partitioning of large tasks into small achievable releases which ensures steady progress and prevents a backlog creep.
4. Iterative nature of the methodology allows for constant improvements and experimentation which is imperative in the case of a chess engine as the program upon initial completion is highly unlikely to perform competitively.
5. It ensures that the code is written to a high standard which I plan to uphold with code reviews with a help of a colleague.
6. Unit testing mitigates the problem of hard to spot software bugs that are relatively easy to overlook while programming alone.
7. Rapid code integration can be achieved with the help of version control (GitHub).
8. Simplicity of this methodology mitigates a large overhead and enables the user to focus on the project.
9. Frequent refactoring encouragement of XP complements the scope of this project perfectly as performance improvements will be an essential to the success of this endeavour.
10. Acceptance testing will provide evaluation of the project and will allow me to more accurately estimate the success chance of the project at different stages of the development.



## 6 Risks

### Insufficient computational power

Due to how computationally demanding neural networks are, it is very likely that available to me hardware will not be sufficient to achieve satisfactory results. To solve this I have set aside a budget that can be used to train the network in the cloud.

### Inaccurate time estimates

Unpredictable nature of this project makes it hard to accurately estimate the time that each task will take. To mitigate this problem I have scheduled additional free time for unpredictable complications.

### Project too ambitious

The scope of the project is very broad and the technologies used are still very new. Most of the knowledge required to complete the project is a result of scientific papers published in the last two years. It is possible that I might not be able to complete the project. To solve this I have scheduled enough time for research. In case this was not enough, the project is still likely to produce valuable insights even when not completed fully.

### Steep framework learning curves

Although there is a lot of learning material available for TensorFlow, it is quickly becoming outdated with the rapid release schedule of the framework. This makes it hard to keep up with the current API. Similarly, the solution is more scheduled time for learning.

### Feature creep

In case of successful completion, the project offers a lot of useful features that might be tempting to implement. This can cause neglect of other work required for this dissertation. To not fall into trap of endless improvement, I intend to first complete all required work and only then start working on code improvements.

Risk Description	Prob.	Severity	Solution
Insufficient computational power	80%	Medium	Use cloud computing
Inaccurate time estimates	40%	Medium	Plan extra time
Project too ambitious	30%	Medium	Invest more time in research
Steep framework learning curves	40%	Low	Allow extra time for learning
Feature creep	20%	Low	Create initial specification

Table 1: Table of risks that might occur during the project development



## 8 Implementation

### 8.1 Prototype

To evaluate the architecture, I have created a low fidelity prototype in the form of a Tic-Tac-Toe program. I have chosen Tic-Tac-Toe as it poses the same type of challenge as chess but provides a very easy solution as the entire game tree can be computed in a few seconds and kept in the memory the entire time. Those characteristics enable me to focus on the neural network architecture without worrying about performance optimisations and allow me to easily fine-tune the hyperparameters and monitor the changes. I have used a scaled-down and altered version of the neural network that I plan to use in the full chess engine. The network takes in the current state of the board as input and outputs an estimate of the score. To simplify the problem, it was formulated as a supervised learning task. This was possible because of the availability of perfect evaluation function which calculates the win probability based on the complete evaluation of the game tree from the current position. Weights have been initialised randomly, leaky RELU activation function was used in combination with Adam optimiser. The move with the highest estimation according to the network was used, the program did not use any exploration strategies. The network after 8 hours of training was able to achieve to 98% Win+Draw performance losing only 2% of the games. The results were evaluated on 10,000 games vs a random agent. A performance this high was predictable as the network learned with perfect information and essentially memorised the entire game. The 2% loss rate might have been caused by too big a size of the network, too small learning rate or too short training time.

### 8.2 Graphical interface

After creating a prototype which validated the viability of this project I have started the work on the user interface. The technologies I have chosen for this task are Java and JavaFX. This choice was heavily influenced by my familiarity with Java and the same task could have been achieved using Python. In retrospect, using Python would make the integration of the engine with the GUI much simpler. The first part of this stage was to create a chessboard representation itself. I have used Canvas element of JavaFX to accomplish this. The pieces are rendered as separate images and redrawn every time a change occurs on the board. This approach decouples the business logic from the interface and allows me to focus on the engine logic and is a part of a design pattern called MVC (Model View Controller) [5]. Every piece is identified by its ID which consists of an encoded position on the board and the colour. This information is accessed when interacting with the model itself and making the move. The graphics controller interfaces with the engine to validate the legal moves and draws them in the form of red dots. It is not possible to make a move other than those permitted by the engine. Sourcing the information directly from the engine makes it easy to swap the engine for another one, for example with another variant of chess (960, King of the hill etc.). To determine the move, an event listener which detects a mouse drag over a piece is used. The coordinates of the mouse are then translated to a square number by a proportional division of the canvas and a simple calculation. If the move was legal the board state is updated. After that, the opposing player or AI agent can make the move.

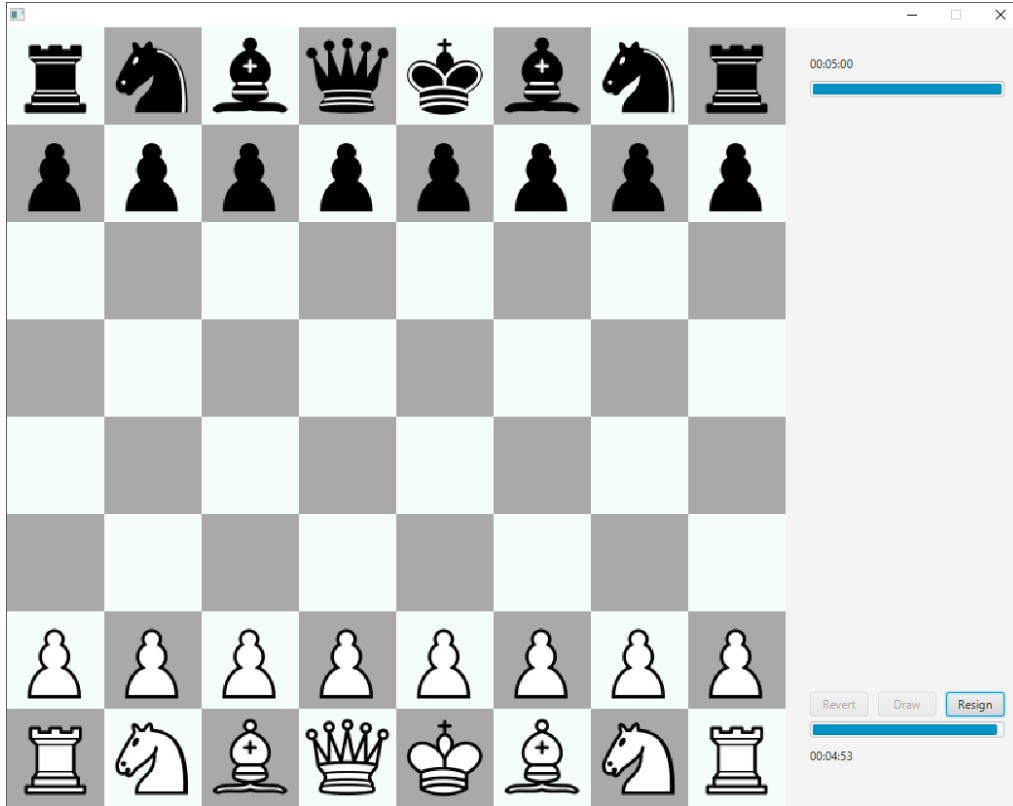


Figure 5: Main game window

When the chessboard UI was completed, I've implemented other elements of the interface such as the new game screen, game result dialog and time controls. The new game dialog allows the user to choose between a game with a human player and a game against the computer AI. User is also able to select the time limit and the time increment per move. The choice of the colour was also implemented but is currently disabled for the purpose of simplifying the training of the neural network (The artificial intelligence can only play black at this moment). To the right of the chessboard, user can see progress bars with the remaining time and controls to offer a draw, resign or revert the move. To finish the development of the user interface I have implemented the "play again" dialog which notifies the user about the end of the game and allows them to start a new game and redirects them to a new game window.

As the focus of this project is directed mainly towards the development of a neural network chess engine, I have decided to limit the graphical interface to a minimum to save valuable development time.

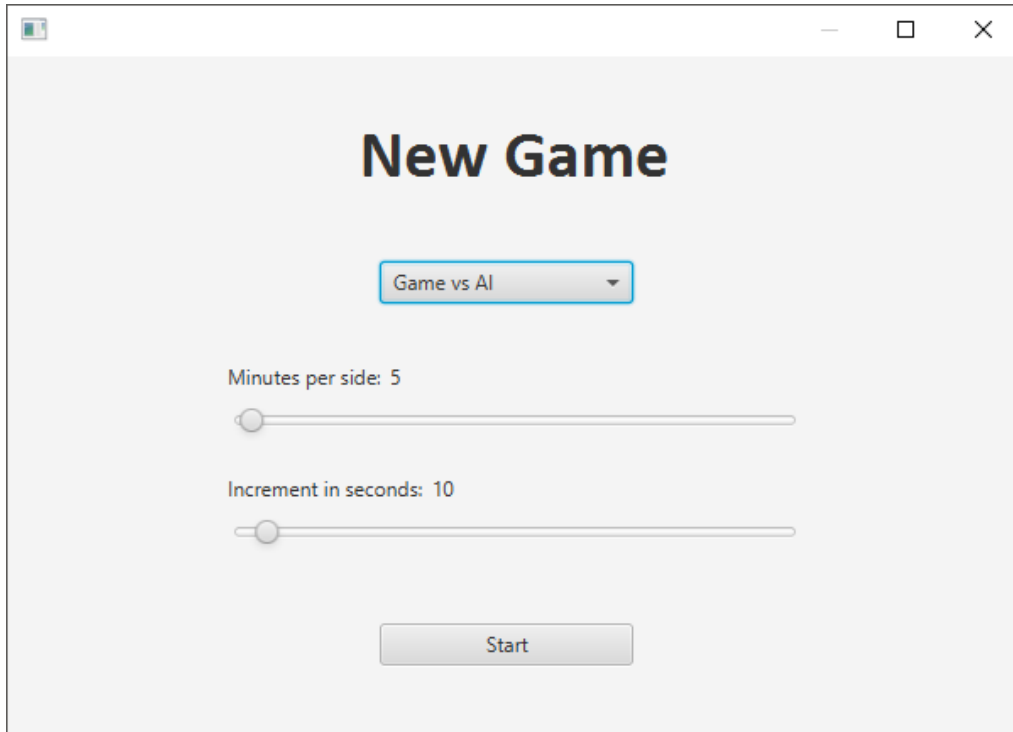


Figure 6: Starting game dialog

### 8.3 Chess engine - Validation

One of the most important parts of any chess engine is its evaluation functions. For this part of the engine speed is crucial. To create this part I have used bitboards and bit-shift operations. Bitboards allow the moves to be executed using direct processor instructions and have the smallest possible memory requirements. The drawback of their usage, however, is the vastly increased complexity of development. The representation is hard to think of intuitively and every operation has to be implemented separately. It is almost impossible to generalise moves and the bit shifting operations are unique for every chess piece. The codebase developed using this method is hard to operate in and some of the checks that need to be performed during a game require a lot of effort to implement. Despite all the problems, the speed increase over object oriented implementation is substantial. I was able to achieve an evaluation speed of 400.000 moves a second while calculating PERFTE (Performance test for move path enumeration) values. For comparison, Stockfish 10 engine was able to calculate 50 million moves per second on my machine. This part of the engine is used when evaluating the moves of a player as well as for the filtering of neural network moves.

## 8.4 Reinforcement Learning approach

The original approach used to train the Alpha Zero was reinforcement learning. The neural network played against an older version of itself. This ensured that the agent always played versus an opponent of sufficient strength. After a certain number of iterations, the opponent network was replaced by a newer snapshot of the network and the entire process was repeated until the network achieved the desired strength. The networks started from the state of no knowledge of the game apart from the move legality check that was applied separately after which illegal moves suggested by the network were filtered out. Using this approach the network learns to suggest only legal moves on its own. Although not specifically stated in the paper, the network will still have a chance of suggesting a non-zero probability for an illegal move. This is due to the fact that it is impossible to include the game rules within the network its dynamic nature itself.

The implementation of this method proved to be more complicated than expected. There is no explicit support within the TensorFlow for reinforcement learning models. To bypass this limitation, I have used placeholder tensors which were populated with game history after the conclusion of a game. After populating them, weights for the moves that occurred during the game were applied retrospectively factoring for the discount rate.

After considering different architectures I have decided to use a simple convolutional neural network with the following architecture [Fig. 7].

```
nn_input = keras.Input((12, 8, 8))

conv1_1 = keras.layers.Conv2D(filters=8, kernel_size=3, strides=1, **params_conv2d)(nn_input)
conv1_1 = keras.layers.SpatialDropout2D(.3)(conv1_1)
conv1_2 = keras.layers.Conv2D(filters=8, kernel_size=3, strides=2, **params_conv2d)(conv1_1)
conv1_2 = keras.layers.SpatialDropout2D(.3)(conv1_2)

conv2_1 = keras.layers.Conv2D(filters=16, kernel_size=3, strides=1, **params_conv2d)(conv1_2)
conv2_1 = keras.layers.SpatialDropout2D(.3)(conv2_1)
conv2_2 = keras.layers.Conv2D(filters=16, kernel_size=3, strides=1, **params_conv2d)(conv2_1)
conv2_2 = keras.layers.SpatialDropout2D(.3)(conv2_2)
conv2_3 = keras.layers.Conv2D(filters=16, kernel_size=3, strides=1, **params_conv2d)(conv2_2)
conv2_3 = keras.layers.SpatialDropout2D(.3)(conv2_3)
conv2_4 = keras.layers.Conv2D(filters=16, kernel_size=3, strides=2, **params_conv2d)(conv2_3)
conv2_4 = keras.layers.SpatialDropout2D(.3)(conv2_4)

conv3_1 = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, **params_conv2d)(conv2_4)
conv3_1 = keras.layers.SpatialDropout2D(.3)(conv3_1)
conv3_2 = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, **params_conv2d)(conv3_1)
conv3_2 = keras.layers.SpatialDropout2D(.3)(conv3_2)
conv3_3 = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, **params_conv2d)(conv3_2)
conv3_3 = keras.layers.SpatialDropout2D(.3)(conv3_3)
conv3_4 = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, **params_conv2d)(conv3_3)
conv3_4 = keras.layers.SpatialDropout2D(.3)(conv3_4)

flat = keras.layers.Flatten()(conv3_4)
dense_output = keras.layers.Dense(output_size)(flat)
softmax = keras.layers.Softmax()(dense_output)

model = keras.Model(inputs=nn_input, outputs=softmax)
```

Figure 7: Neural network architecture

The input of the network was a simplified version of that presented by DeepMind’s team, twelve 8x8 binary arrays were used to represent the pieces on the board, one for each piece types of both colours. I have decided to do not use other features used in the paper for the sake of simplicity, they were unlikely to introduce any improvements on the lowest level of the play.

The network consisted of 10 convolutional layers, each followed by a dropout layer. The usage of dropout regularization allowed me to make the network more robust and mitigate the risk of overfitting which was particularly high as the amount of training data was limited. The AlphaZero used a much larger network with ResNet architecture but this proved to be infeasible because of the increased requirement for training data and the lowered speed of inference.

Finally, the output from the convolutional layers was flattened and converted to a single fully connected layers with the size of 4096 followed by the softmax layer converting the output to a probability distribution over the move-space. The size of the output layer represents moves from each square of the board to any other. This is the simplest representation of the move-space but it includes possible illegal moves. However as previously mentioned, even when a non-zero probability for such moves occurs, they are easily filtered out. Considering this, I’ve decided to use it as it reduces complexity without major drawbacks, it is also the shape that AlphaZero used.

The hyperparameters were selected manually and where possible, values from my previous projects that have proven to work well were reused. Methods such as grid-search of hyperparameter space were infeasible because of computational complexity and lack of an easy way for validation.

Taking into an account the computational complexity of this approach I was able to generate around 200 games per hour which were not enough to even notice the improvement in the strength of the engine even after 50 hours of training time. The model also proved impossible to test. When matched against a random agent during 500 test games the draw ratio was 100%. This is to be expected considering the branching factor of chess and the end game move space. It is possible that the neural agent had strength higher than a random sample but the improvement was too low to be measured.

This problem was identified by me early into the planning phase of this project and was one of the highest risks. By extrapolating from the training graph of Alpha Zero we can estimate that the engine required approximately 2 million games to reach 1000 Elo and didn’t show noticeable improvement over random until after around 500.000 games into training [Fig. 8]. Considering that the model I used was trained over 10.000 games it predictable that the results would not be easily visible.

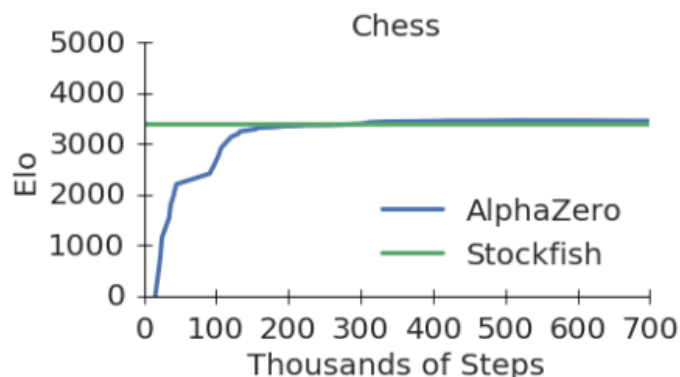


Figure 8: Training graph of AlphaZero. Each step consisted of 64 games.

There are also other possible reasons for no improvement, such as flawed network architecture, badly selected hyperparameters or code errors. All of those problems are very hard to resolve and require expert level knowledge of the neural networks.

## 8.5 Supervised Learning approach

After encountering the aforementioned problems, I have decided to modify the architecture to address them. The training method was changed from reinforcement learning to supervised learning. This method relies on introducing labelled examples to the engine and making it try to approximate the result. To implement this approach I required a database of chess games and only two options were available. One of them was to use an existing database of expert games and train the engine on those examples, the other option was to use an existing chess engine that uses traditional methods for calculation and attempt to teach the network to mimic it. This method is often called supervised pre-training, it is often used when either a limited dataset of labelled example is available for training or when the reinforcement learning problem is very hard to train from scratch. Google has also used this method for the original AlphaGo to accelerate the initial phases of the training, they have, however, since dropped this approach as it introduced unwanted patterns into the engine. The reasoning was that the engine will be able to become much stronger if the human element isn't introduced into the system thus eliminating any inefficient or bad play habits. I have chosen to use Stockfish10 as the self-play engine and the python-chess library as a wrapper over it. Stockfish is an open source engine and is available for anyone to download for free and the source code is also open for anyone to contribute on GitHub. The engine was paired against itself to produce a stream of games from which labels used to train the network were derived. I have generated 10.000 games using this method and I am continuously training the engine. The architecture of the network changed only slightly to change the input layer from binary input to an integer representation of characters representing pieces on the chessboard. This made it easier to parse the input from the games and should not change



the performance of the network. After training on the generated games the engine seems to have improved in strength slightly, I will describe the results more thoroughly in the evaluation section. This type of pre-training is a very easy pre-processing step that can be later paired with the original approach. The trained model can be easily transferred and further trained.

## 8.6 Interfacing between subsystems

Finally, to complete the integration between the chess engine created in Python and the GUI created in Java, I have created a simple Python web API using Flask. This allowed me to create a flexible connection between the two subsystems. Said API can be hosted on any server and use any underlying hardware. The communication is performed using HTTP protocol calls. This is different from the initial plan of using a UCI (Universal chess interface) but after careful consideration and considering the implemented architecture, it is clearly a superior approach.

## 9 Evaluation

Considering the achieved performance of the trained neural network model it was hard to measure it precisely. Even when attempting to compare the model to a random agent, the measured performance did not exceed statistical error. This outcome is caused by the problem previously explained, namely the complexity of the chess endgame. However, when attempting to evaluate the engine manually, one can recognise that the moves performed by the engine are noticeably different from that of the random agent. Although not directly measurable, this demonstrates that the engine started to absorb patterns from the games it was trained on. When comparing the different approaches, the most notable change occurred after a large batch of 7000 games during the supervised pre-training phase. I do however recognise that this effect has a chance of being my own bias or placebo effect.

## 10 Testing

Testing of the entire system was performed in multiple phases. The graphical interface was tested using manual testing. I have decided that this is the best approach which satisfies the basic needs for this project. The focus of the project was directed towards the engine itself and the interface was developed solely for demonstration purposes and easier evaluation. During the testing, I have encountered several bugs which were fixed according to their severity. To the best of my knowledge, there are no leftover bugs in the user interface part.

The second part of the testing focused on the evaluation functions. This part was decidedly the hardest to test. Because of the fact that the evaluation relies on bitboards and the state of the board using unit tests to test it was impractical and inefficient time-wise. To test the evaluation I have used the PERFT method. There are pre-made tables available with the calculated numbers of possible moves in chess for selected positions, comparing the number achieved by the developed engine running a move calculations with a certain depth, I was able to detect the problems and identify the discrepancies between the results. I was able to eliminate most of the encountered errors with the exception of a few, hard to eliminate corner cases. The most notable example is the en passant takedown discovered check. I have decided that it was not sensible to focus on the hard to implement fixes from the project management perspective. Due to the limited timescale and large complexity of the undertaken project, some of the errors remain unfixed. The severity of those errors is not critical and unless the user purposefully tries to break the system, the core functionality remains unaffected.

The testing of the neural network was performed by validating the input and output to the network manually. It was possible to unit test this part of the system but the time pressure did not allow me to do it.

The last part of the testing process was the smoke testing [3], also known as build verification testing. The goal of this process is to ensure that the core functionality of the fully integrated system works. This was done manually and to ensure that the system is ready for presentation during the project fair.

## 11 Conclusions

The result of this project can be interpreted in multiple ways. I am inclined to consider it a success. The scope of the project was admittedly too ambitious and in retrospect, better suited for a doctoral thesis rather than bachelors dissertation. The topics covered in this project are mostly results of the work of world-leading research teams backed by considerable resources and manpower. It was unrealistic on my side to attempt to replicate results that required months of research of a large company like Google and its DeepMinds team as well as the computational power of the large server clusters of specialised hardware. This project, however, resulted in myself committing several hundreds of hours into research and development. This was an opportunity to develop my knowledge and skills which is, in my opinion, the core goal of the 3rd year project. I was able to complete the minimum viable product and have laid solid groundwork and platform for future research and work on this project. If an opportunity arises I would like to continue the work on this chess engine, perhaps during my master thesis. As for the delivered system itself, disregarding the strength of the engine, it satisfies all the requirements specified during the planning phase. The created neural network presented promising results given limited hardware resources available. It is possible, given a larger computational effort, that the engine would start to gain strength but as described previously, this turned out impossible to evaluate. To summarise, I am satisfied with the achieved result and have gained considerable experience in the machine learning field during the development of this project. This experience will become very valuable in my future endeavours and professional career.

## References

- [1] Contributing to leela chess zero. creating the caissa of chess engines.... - chess forums.
- [2] Hebb, d. o. the organization of behavior: A neuropsychological theory. new york: John wiley and sons, inc., 1949. 335 p. \$4.00. *Science Education*, 34(5):336–337, 1950.
- [3] Smoke testing, Jul 2018.
- [4] Ravikant Agarwal and David Umphress. Extreme programming for a single person team, 01 2008.
- [5] F. Buschmann. *Pattern-Oriented Software Architecture, A System of Patterns*. Wiley Series in Software Design Patterns. Wiley, 1996.
- [6] Claude E. Shannon. Xxii. programming a computer for playing chess. 41:256–275, 03 1950.
- [7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [8] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.
- [9] Kurt Hornik, Maxwell B Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1 1989.
- [10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [11] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, Jun 1976.
- [12] James Moor. The dartmouth college artificial intelligence conference: The next fifty years. *AiMagazine*, 27, 12 2006.
- [13] J. V. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.
- [14] D. Prinz. Robot chess. *Computer Chess Compendium*, page 213–219, 1988.
- [15] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957.
- [16] W S McCulloch and W Pitts. A logical calculus of the ideas immanent in nervous activity. 1943. *Bulletin of mathematical biology*, 52:99–115; discussion 73, 02 1990.

- [17] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959.
- [18] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, jul 2007.
- [19] K. Schwab. The fourth industrial revolution: what it means, how to respond. (accessed: 2018-11-1) <https://www.weforum.org/agenda/2016/01/the-fourth-industrial-revolution-what-it-means-and-how-to-respond/>.
- [20] K. Schwab. *The Fourth Industrial Revolution*. Crown Publishing Group, 2017.
- [21] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv e-prints*, 2017.
- [22] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.
- [23] Alan M. Turing. Chess. *Computer Chess Compendium*, page 14–17, 1988.
- [24] [www.computerchess.org](http://www.computerchess.org). Ccrl 40/40 ranking (accessed: 2018-10-31).
- [25] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016.