

Project: Assess different AI technologies for a simple board game

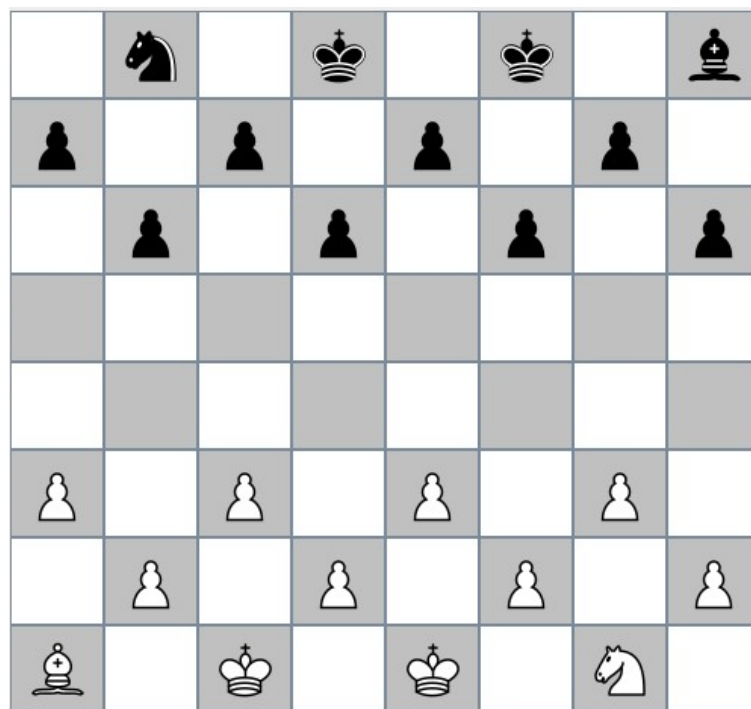
Student: Lee Donovan

Email: ld26@hw.ac.uk

Supervisor: Hans Wolfgang Loidl

Final Year Dissertation

Programme: BSc (Hons) Computer Science



Declaration

I, Lee Donovan confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Lee Donovan

Date: 21/04/2021

Abstract

Cheskers is a two player board game that takes elements from both chess and checkers (Bodlaender, 1995). The goal of the game is to checkmate your opponents just like in chess (Bodlaender, 1995). The aim of this project is to develop artificial agents to play this game using two different technologies, minimax search and neural networks. Once several agents have been developed I will be comparing the results and discussing pros and cons of each method such as the playing strength and computation demands of each agent, how long they take and how hard it was to implement. The game and minimax agents will be implemented in Java and the neural networks in Python. The aim of this document is to show the background research done, the full design and implementation and an evaluation of the final system and agents.

Overall my evaluation found minimax agents to perform significantly better than neural network agents with one agent, mmSimple2, winning 71% of its games against other agents, 99% against an agent that moves randomly and 90% against me. I think the biggest reason for neural networks being worse is the amount and quality of data I had to train them with. My code can be found here: <https://github.com/wockradd/Cheskers>

Table of Contents

Declaration	2
Abstract	3
1 Introduction	8
1.1 Context	8
1.2 Aims and Objectives	8
1.3 Planning	9
1.4 Evaluation	9
2 Background	10
2.1 AI	10
2.1.1 Agents	10
2.1.2 Task Environments	11
2.1.3 Searches	12
2.1.3.1 Multiagent Search	13
2.1.4 Machine Learning	15
2.1.4.1 Artificial Neural Networks	15
2.1.5 Alternatives for Games AI	17
2.2 Games	18
2.2.1 Board Games	18
2.2.1.1 Chess	18
2.2.1.2 Checkers	21
2.2.1.3 Cheskers	22
2.2.2 Existing AIs for These Board Games	24
2.2.2.1 Chess AI	24
2.2.2.1.1 The Turk	24
2.2.2.1.2 Deep Blue	24
2.2.2.1.3 AlphaZero	24
2.2.2.2 Checkers AI	25

2.2.2.3 Cheskers AI	25
2.3 Technical Infrastructure	26
2.3.1 Programming Languages	26
2.3.1.1 Python	26
2.3.1.2 Java	26
2.3.1.3 Other Languages Considered	27
2.3.2 Libraries	27
2.3.2.1 Tensorflow	27
2.3.2.2 Keras	28
2.3.2.3 Deeplearning4j	28
2.3.2.4 Swing	28
2.3.3 Base code	29
3 Implementation	30
3.1 System Overview	30
3.2 Requirements	31
3.3 Game	32
3.3.1 GUI	32
3.3.2 Board	33
3.3.3 Pieces	34
3.3.4 Player and Game	36
3.3.5 Human Interaction	36
3.4 Agents	37
3.4.1 Random	37
3.4.2 Minimax	37
3.4.2.1 Minimax Simple	37
3.4.2.2 Minimax Positional	38
3.4.3 Neural Networks	39
3.4.3.1 Neural Network 1	40

3.4.3.2 Neural Network 2	41
4 Testing and Evaluation	43
4.1 Game	43
4.2 Agents	44
4.2.1 Win Rates	44
4.2.2 Time taken	46
4.2.3 Boards Evaluated	47
4.3 Conclusions	48
5 Project Plan	49
5.1 Methodology	49
5.2 Timetable	50
5.3 Risk Analysis	52
5.4 Evaluation plan	53
5.5 Professional, Legal, Ethical and Social Issues	54
5.5.1 Professional Issues	54
5.5.2 Legal Issues	54
5.5.3 Ethical and Social Issues	54
6 Conclusions	55
6.1 Summary of Project	55
6.2 Future Work	56
7 References	58
8 Appendices	63
8.1 Initial Code	63
8.1.1 Simple Keras Model	63
8.1.2 Initial GUI Code	64
8.2 Diagrams	66
8.2.1 Game Class Diagram	66
8.2.2 Win Rate Table	67

8.2.3 Time Taken Table	67
8.2.4 Time Taken With and Without Pruning Table	67
8.2.5 Boards Evaluated	67
8.3 Apache 2.0 License	68

1 Introduction

1.1 Context

Computer scientists have been developing AI for games since 1951 with the first checkers and chess AIs being developed (Copeland, 2006). The approaches and results have only gotten better since. We are in something of an “AI renaissance”(Ensor, 2020) currently with advances in AI techniques and an increase in available data to train these AIs.

The purpose of this project is to develop different AIs for the board game cheskers, created by Solomon Golomb in 1948 (Bodlaender, 1995), and assess them based on time taken and win rate. Cheskers is a variant on chess with pawns and kings moving like checkers and promoted checkers respectively and knights and bishops moving similarly to how they move in standard chess. This should be interesting as both chess and checkers have had a lot of AI research, as discussed in section 2.2.2, so combining these games may lead to interesting results. Players win when they capture all their opponents kings and lose if they have no legal moves left to make. I will be describing the rules for chess, checkers and cheskers further in section 2.2.1.

1.2 Aims and Objectives

The aim of this project is to compare and contrast different AI techniques based on time taken to find move, win rate, etc. With the adversarial search I will also be looking at how many nodes it explores and the depth it reaches. See section 5 for a more in depth explanation of how I planned to achieve this aim and section 6 for my final conclusions.

The first objective of this project is to develop first an adversarial search AI with minimax and alpha-beta pruning to play cheskers. The second objective is to use the data from the many games this agent plays to develop a neural network to play cheskers. The third objective is to compare the different agents by doing a systems evaluation. The project will mainly use Java for the implementation with Python being used to train the neural network. See section 3 for an explanation of the implementation of the game and agents.

1.3 Planning

Due to the structure of the university timetable I decided to separate the project into tasks to be completed during term one and term two.

In term one I worked on the research report, implementing the UI and rules for the game, and a random AI for testing. In term two I worked on the dissertation, the minimax AI, the neural network AI, compared the different AIs and did a system evaluation based on these results. See section 5 for the timetable in more depth and an explanation of how things deviated as the project progressed.

1.4 Evaluation

I did a system evaluation based purely on how each agent performed. Because I would be the only person evaluating the final system it was very important to do this part correctly. As such I spent a large part of the research time of this project looking at how other board game AIs are evaluated. I also made sure to develop the system in a way that would make it easy to test such as writing scripts to run the program repeatedly. See section 5.4 for my initial evaluation plan and section 4 for the full system evaluation.

2 Background

2.1 AI

2.1.1 Agents

Agents can be defined as “anything that can be viewed as perceiving its environment through sensors and actuators”(Russell and Norvig, 2010) sensors being what it uses to perceive its environment (through percepts) and actuators being what it uses to act upon this environment (through actions). This can be seen diagrammatically in Figure 1.

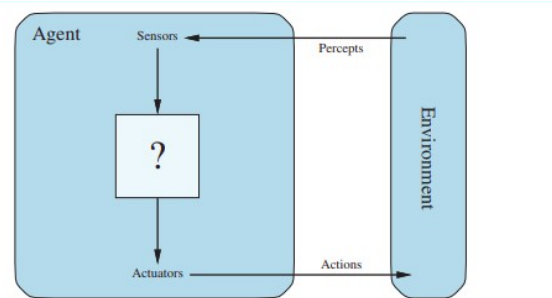


Figure 1 (Russell and Norvig, 2010) – Shows how agents interact with their environment through actions and percepts.

Russell and Norvig then break an agent down into two main sections, the agent function and the agent program. The agent function is a table that “maps any given percept sequence to an action”(Russell and Norvig, 2010) and the agent program is what implements this function.

A rational agent can be broadly defined as “[an agent] that does the right thing” (Russell and Norvig, 2010) meaning that every entry in the table for the agent function is filled out correctly. What is rational depends on four things; a performance measure, the agents prior knowledge, actions the agent can perform and the agents percept sequence (Russell and Norvig, 2010). This leads to the more formal definition of a rational agent “For each possible percept sequence, a rational agent should select an action that is expected to maximise its performance measure, given the evidence provided by the percept sequence and whatever built in knowledge the agent has” (Russell and Norvig, 2010).

With all this in place the agent can be said to be acting rationally and this is the type of agent I will be focusing on. The agents I develop will have different agent functions but the same agent program, in other words they will have a different mapping from percepts to actions but will use the same method for interacting with the environment.

2.1.2 Task Environments

Task environments are “the ‘problems’ to which rational agents are the ‘solutions’” (Russell and Norvig, 2010) and can be described through the PEAS description (performance, environment, actuators and sensors) (Russell and Norvig, 2010). These can be categorised using the following six properties.

- Fully observable vs. partially observable:

Fully observable environments are ones where “an agents sensors give it access to the complete state of the environment at each point in time” (Russell and Norvig, 2010). Environments where the agents sensors can detect all aspects that are relevant are effectively fully observable. If neither of these conditions are true it is a partially observable environment. As cheskers is a perfect information game my task environment will be fully observable.

- Single agent vs. multiagent:

Single agent environments only have one agent acting on the environment and multiagent ones have many. Russell and Norvig say “an agent playing chess is in a two-agent environment” (Russell and Norvig, 2010) so my task environment will clearly be a multiagent one, with the human player being the second agent in some instances.

- Deterministic vs. stochastic:

As defined by Russell and Norvig “If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic.” (Russell and Norvig, 2010). As there is no uncertainty in cheskers about the next state of the board based on the move played my task environment is deterministic.

- Episodic vs. sequential:

Episodic environments are ones where “the agents experience is divided into atomic episodes”

(Russell and Norvig, 2010) where the current decision does not depend on previous actions taken. In sequential environments decisions can depend on which actions were taken previously so requires the agent to have some kind of forward planning. My task environment is sequential.

- Static vs. dynamic:

An environment is dynamic if “the environment can change while an agent is deliberating” (Russell and Norvig, 2010) and static if not. As players in cheskers must wait for their opponent to move before they can move my task environment is static.

- Discrete vs. continuous:

This distinction applies to the state of the environment and how time is handled (Russell and Norvig, 2010). As cheskers has a finite number of board states and moves are discrete events the task environment is discrete.

In conclusion the task environment my agents will be attempting to solve will be a fully observable, multiagent, deterministic, sequential, static and discrete.

2.1.3 Searches

In most board states in cheskers there is no single action which will win the game and so a sequence of actions must be performed. Any agent developed to win this game must therefore work out a sequence of actions by searching. Russell and Norvig define a search as “the process of looking for a sequence of actions that reaches a goal” (Russell and Norvig, 2010) and break search strategies down into uninformed searches and informed searches. We can visualise searches using search trees with nodes representing states and edges representing actions. Figure 2 shows a search tree.

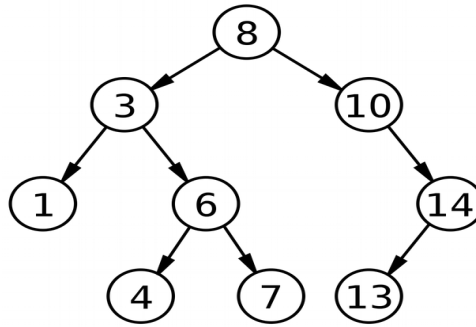


Figure 2 (Coetzee, 2005) – A simple binary tree.

Uninformed search strategies, also known as blind searches, include both breadth-first and depth-first search. These are both simple strategies with the agent searching the tree having no information about which states are more promising, it just searches until it finds a goal state. Neither of these algorithms are particularly efficient as the agent may spend a lot of time searching hopeless branches.

Informed searches “can find solutions more efficiently than can an uninformed strategy.” (Russell and Norvig, 2010) as these strategies use a heuristic function to evaluate the cost of each node, the lower the better. An example of an algorithm that uses this strategy is A* search. The question is then how do you come up with a heuristic function for a problem as complicated as checkers? This is where machine learning comes in which I discuss in section 2.1.4.

2.1.3.1 Multiagent Search

The previously mentioned search strategies are mainly for single agent environments so the last search strategy I will discuss is adversarial search which is specifically designed for multiagent environments (Russell and Norvig, 2010). This strategy was used by both Deep Blue and Chinook (Russell and Norvig, 2010), two game playing AIs I will go over in section 2.2.2. Minimax is one way to implement this strategy with the simplest minimax algorithm having two players playing a zero sum game with one player trying to maximise the score and one trying to minimise it. The algorithm then performs a depth-first search and backs up the minimax values through the tree. Figure 3 shows an adversarial search tree where the maximising agent has decided to take action a_1 because this leads to the node with the highest minimax value.

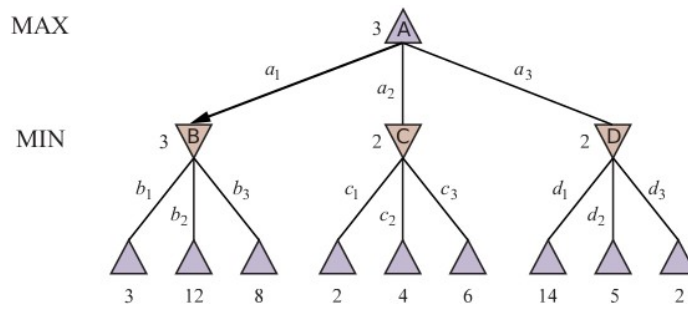


Figure 3 (Russell and Norvig, 2010) – A minimax search tree.

Unfortunately this is still a form of depth-first search and so is inefficient. A solution to this is alpha-beta pruning (Russell and Norvig, 2010), a pruning algorithm to allow potential branches to be ignored. This algorithm keeps track of two values, alpha and beta, which are the best values for the maximising and minimising players respectively. Figure 4 shows alpha-beta pruning being performed on the same tree as figure 3 resulting in two branches from node C being ignored. The values for alpha and beta are in the square brackets next to each node. Minimax with alpha-beta pruning is the type of search my first AI will use, with a blog post (Hartikka, 2017) on freecodecamp.org as my guide.

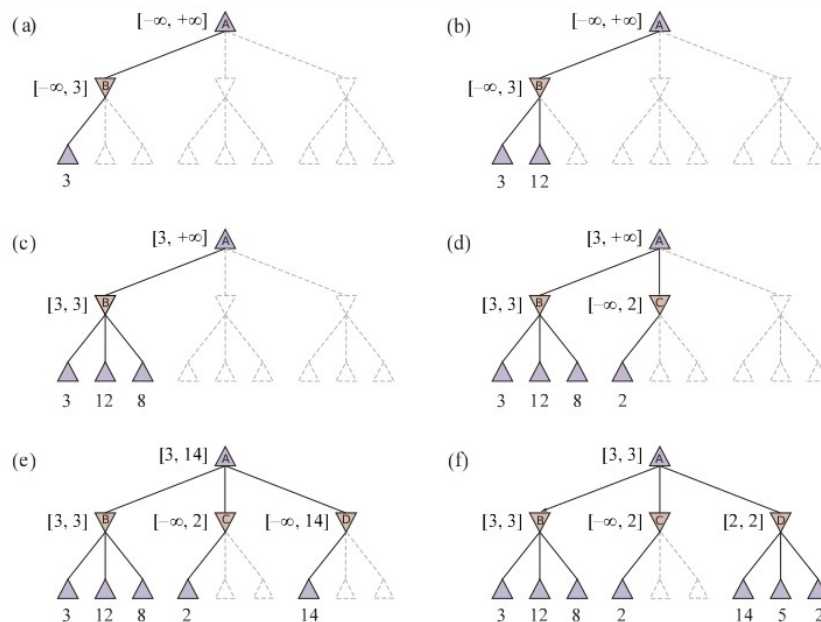


Figure 4 (Russell and Norvig, 2010) – Shows how alpha-beta pruning works in a search tree, step by step.

2.1.4 Machine Learning

The operational definition of learning as that “Things learn when they change their behavior in a way that makes them perform better in the future” (Witten and Frank, 2011). This is easy to measure with agents playing games, if the agents win rate is improving it can be said to be “learning”.

This normally involves building a model using either supervised or unsupervised learning. Supervised learning is when you build the model using a data set containing both inputs and desired outputs, unsupervised only having the inputs (Witten and Frank, 2011). An example of a model you might build is a neural network which I will go over in section 2.1.4.1.

2.1.4.1 Artificial Neural Networks

Neural networks are a type of model you can build using machine learning (Witten and Frank, 2011) and have many different variations. Here I will be focusing on the multilayer perceptron.

A perceptron is a linear function which using weights, biases and inputs returns an output. These only work for linearly separable data which means “the data can be separated perfectly into two groups using a hyperplane” (Witten and Frank, 2011) and only return a yes or no as an output. The linearly separable constraint can be visualised with figure 5 and 6. Here the groups are + and – and each example has two attributes with values of 0 or 1. The data in figure 5 is linearly separable because you can draw a line separating the groups and the data in figure 6 is not so this would be impossible to classify this using a perceptron. Figure 7 shows a perceptron diagrammatically.

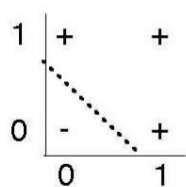


Figure 5 (Komendantskaya, 2021)

– Shows linearly separable data.

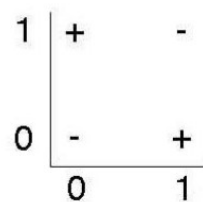


Figure 6 (Komendantskaya, 2021)

- Shows data that is not linearly separable.

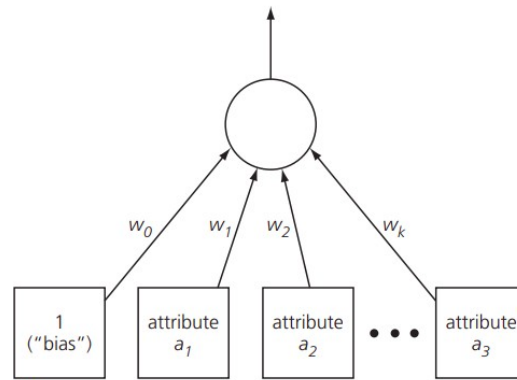


Figure 7 (Witten and Frank, 2011) – A single perceptron.

A perceptron is good start but for more complicated classification tasks, such as figuring out the best move in a board game, doesn't do everything we need. A multilayer perceptron then takes many of these perceptrons and links them up in multiple layers. This gets rid of the linearly separable constraint and allows each output node to return yes or no to a specific move. This technique has been use by many chess (David, Netanyahu and Wolf, 2016) and checkers (Chellapilla and Fogel, 1999) playing agents in the past. Figure 8 shows a simple neural network with one input layer, one hidden layer and one output layer.

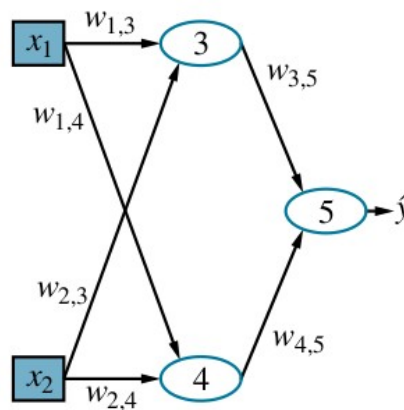


Figure 8 (Russell and Norvig, 2010) – A simple neural network.

This is what was used by the game playing AI AlphaGo with incredible results (AlphaGo: The story so far, n.d.) and is the second type of AI I will develop with this blog post (the_real_greco), 2019) on chess.com as my guide.

2.1.5 Alternatives for Games AI

As I stated in section 1.2 I will be developing my AIs using minimax and neural networks but there are many other methods for developing game playing AIs.

Monte carlo approximation is a strategy based on minimax where instead of looking at every possible move you look at a random subsample of moves. This strategy is useful in card games because of the amount of hidden information in these games (Russell and Norvig, 2010). Because cheskers is a perfect information game this strategy is not necessarily the best fit.

Convolutional Neural Networks, CNNs, are another strategy of machine learning based on neural networks. With a CNN you take all possible local fields from your input, for example all possible 3x3 section of the cheskers board, and this becomes the first layer in your network. This is useful for things such as image recognition because the importance of certain pixels can be affected by the pixels around it. This is also true with a cheskers board as the importance of a piece on a square can be affected by the pieces on the squares around it. This approach would probably improve on the multilayer perceptron method and so could be an additional AI we develop if there is time towards the end of the project.

2.2 Games

As Russel and Norvig say “Games have engaged the intellectual faculties of humans ... for as long as civilization has existed” (Russell and Norvig, 2010) and as such it should be no surprise the AI community is interested in them. They also point to many other factors which explain the popularity of game playing AIs (Russell and Norvig, 2010):

- The state of games is easy to represent.
- The actions agents can take are restricted by precise rules.
- The abstract nature of games allows for many different approaches.
- They are interesting because they are nontrivial.

Because of these factors game playing AIs are usually at the forefront of AI with current state of the art AIs doing tasks such as playing Go(Russell and Norvig, 2010). These factors don't apply to all games though so AI research has focused mainly on a subset of games, namely deterministic, fully observable, two player, turn taking games. Games of this type include chess, checkers and, the game I am interested in, cheskers.

2.2.1 Board Games

2.2.1.1 Chess

Chess is “one of the oldest and most popular board games” (Soltis, 2020) and first appeared in India around the 6th century AD (Soltis, 2020). Since then it has grown in popularity and spread to many cultures. Early chess was based on the existing Indian games chaturanga and shatranj went through many changes over the years before the rules and set design were standardised in the early 19th century (Soltis, 2020). These changes vary from largely insignificant, such as the bishop being called an elephant in Russia and the fool in France (Soltis, 2020), to huge changes like the win condition being different in different countries (Soltis, 2020).

Figure 9 shows the earliest known chess set which was discovered in Jordan in 1991 dating back 1,300 years (David, 2019).



Figure 9 (David, 2019) – Oldest chess set discovered.

Modern chess has 6 unique pieces, the pawn, rook, bishop, knight, king and queen and the goal of the game is to get your opponents king into checkmate (a position where it is unable to avoid capture) (Soltis, 2020). Each of these pieces is commonly represented by the icons and letters in figure 10. Modern chess also uses numbers to label the rows or ranks and letters for the columns or files. Figure 11 shows a board set up with the pieces in their starting positions. With this notation in place you can use algebraic notation to describe moves such as Bxe5 meaning bishop captures on e5 and Qh4e1 meaning the queen on h4 moves to e1.



King(K) Queen(Q) Bishop(B) Knight(N) Rook(R) Pawn(P)

Figure 10 (made by me using (Chess pieces, 2013)) – Standard images to represent pieces.

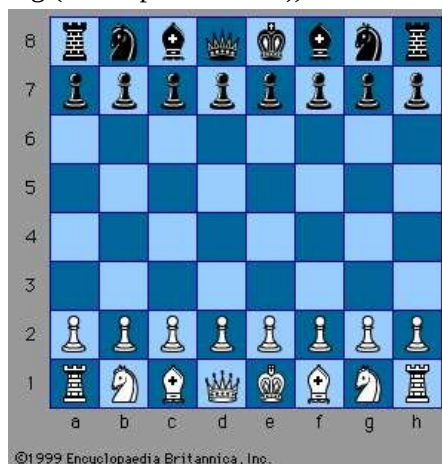


Figure 11 (Soltis, 2020) – A chessboard set up for a game.

Rules for moving pieces are as follows (Soltis, 2020):

- The king can move one square in any direction. If the king can be captured next turn it is said to be in check and must move.
- A rook can move vertically or horizontally to any unobstructed square.
- A bishop can move diagonally to any unobstructed square.
- A queen combines the movement of the rook and bishop.
- A knight moves in an L shape, moving 1 square in any direction and 2 squares in an orthogonal direction. It also has the ability to jump over pieces while it moves.
- A pawn can only move forward and normally just one square at a time. Exceptions to this are capturing which a pawn does on the squares diagonally in front of it and the first move the pawn makes where it can move two squares.

Finally there are 3 additional rules to the game; castling, pawn promotion and en passant (Soltis, 2020).

- Castling allows you to move your king two squares in the direction of a rook and move the rook to the square the king crossed. This can only be done if the king and rook haven't moved yet and the king cannot enter check at any time during this move.
- Pawn promotion allows you to promote your pawns to any other piece once they reach the far side of the board.
- En passant is a special move pawns can make where they can capture other pawns that moved two squares as if it moved 1 square. This move must be made the turn after the pawn moves 2 squares or it is lost. Figure 12 shows this in action with the black pawn performing an en passant.

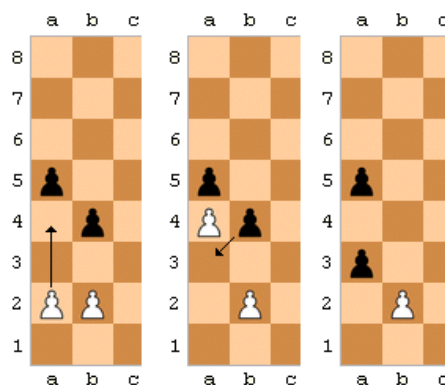


Figure 12 (*En passant*, 2006) – Shows how a pawn can perform an en passant.

The rules explained above are the standard rules in modern chess but some players like to adjust these rules to change the dynamic of the game (Soltis, 2020). These variations are known as “fairy chess” and the game checkers can be seen as a version of fairy chess.

2.2.1.2 Checkers

Checkers, also known as draughts, is another extremely old game with similar games being played around 1600 BC (Checkers | game, 2019). The 8x8 square board was introduced in the 12th century and rules compelling capture of opponent pieces in the 16th century “producing a game essentially the same as modern checkers” (Checkers | game, 2019).

Each player starts with 12 pieces which have many different names depending on the country such as men, uncrowned pieces and simply pieces. Here I will just refer to them as pieces. These pieces start in the configuration shown in figure 13. Each piece can move diagonally forward to an empty square or if this square contains one of the opponents pieces it jumps over this piece to the next black square and the opponents piece is captured (Checkers | game, 2019). This is shown in figure 14 with white capturing one of blacks pieces. If a piece can capture it must and once a piece has captured another it must capture additional pieces until it cant capture anymore (Checkers | game, 2019). The aim of the game is to capture all your opponents pieces.

One additional rule is “crowning” which is similar to pawn promotion in chess. Once a piece gets to the final row on the board (known as the kings row (Checkers | game, 2019)) it is promoted to a king, usually shown by stacking two pieces. The king moves and captures the same as a normal piece but can go backwards too (Checkers | game, 2019).

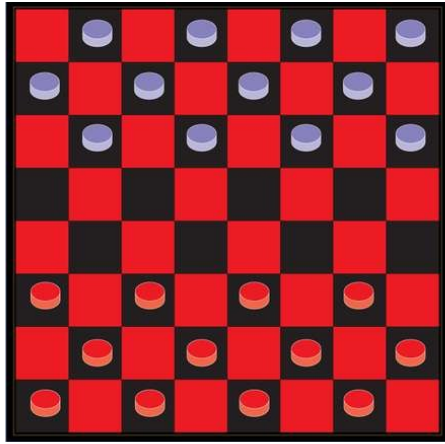


Figure 13 (Checkers | game, 2019)

– A checkers board set up for a game.

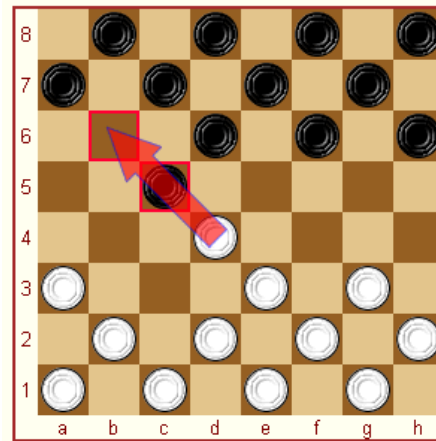


Figure 14 (YourTurnMyTurn.com: checkers rules, n.d.)

– Shows how pieces capture.

2.2.1.3 Cheskers

The final board game I will describe is cheskers which “can be seen as a cross-over between chess and checkers”(Bodlaender, 1995). This game was invented in 1948 by Solomon Golomb(Bodlaender, 1995) and takes the bishop and knight, with slight changes, from chess and the pieces and kings from checkers. The aim of the game is to put all the opponents kings in checkmate (Bodlaender, 1995). Figure 15 shows the initial board state for cheskers using the icons commonly used in chess.

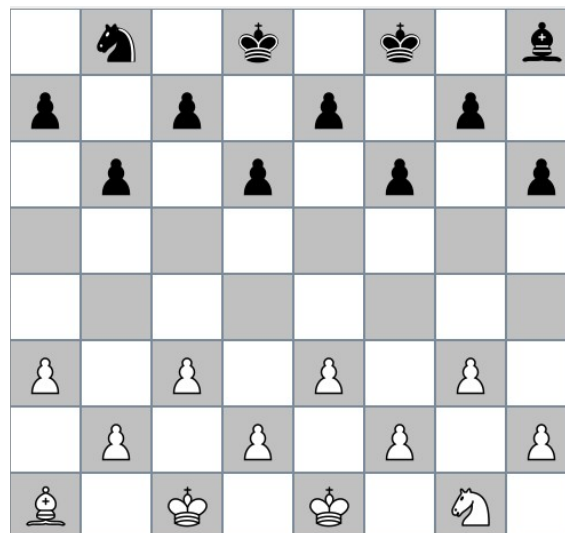


Figure 15 (made by me using (Chess pieces, 2013))

– a cheskers board set up for a game.

Rules for moving pieces are as follows (Bodlaender, 1995):

- Pawns move the same as pieces in checkers.
- Kings move the same as kings in checkers.
- Bishops move the same as bishops in chess.
- The knight moves like an extended knight in chess, that is it moves 1 square in any direction and 3 squares in an orthogonal direction. This piece is also sometimes referred to as a camel in fairy chess to avoid confusion with normal knights.

There is an additional constraint on movement of pieces so that if a pawn or king can capture it must, this means if there is a sequence of captures you must capture all these pieces and you couldn't move your knight instead for example (Bodlaender, 1995). Finally pawns can be promoted to any other piece once they reach the back row and a stalemate (a situation where one player cannot move) counts as a loss for that player instead of a draw as in chess (Bodlaender, 1995).

2.2.2 Existing AIs for These Board Games

2.2.2.1 Chess AI

2.2.2.1.1 The Turk

The first “AI” I would like to discuss was actually a fake known as the Turk. It was built in the 18th century and consisted of a man sitting at a table with a chessboard (Mechanical Turk - Wikipedia, n.d.). The Turk was well known for beating people such as Napoleon Bonaparte and Benjamin Franklin but was later revealed to be an elaborate hoax (Mechanical Turk - Wikipedia, n.d.). The Turk actually had a secret compartment where a skilled human chess player would hide and operate the machine (Mechanical Turk - Wikipedia, n.d.). Unfortunately it would be around 200 years before an actual chess AI would be built.

2.2.2.1.2 Deep Blue

Deep Blue was a chess program developed by Feng-hsiung Hsu at Carnegie Mellon University and later IBM (Knudsen, 2020). It “ran on a parallel computer with 30 IBM RS/6000 processors doing alpha-beta search.”(Russell and Norvig, 2010) and is probably most well known for defeating the then world champion Garry Kasparov in 1997(Russell and Norvig, 2010). Considering a machine in 1997 was able to beat a world champion using alpha-beta search and computing power has increased exponentially since then alpha-beta search should be a good method for building an AI to play cheskers.

Deep blue wasn't only using alpha-beta search however and used “an opening book of about 4000 positions”(Russell and Norvig, 2010) as well as “a large endgame database of solved positions”(Russell and Norvig, 2010) meaning Deep blue only had to rely on its searches during the midgame. I want my AI to have no outside information and rely entirely on its heuristics so I will not be implementing this.

2.2.2.1.3 AlphaZero

AlphaZero is an AI developed by DeepMind that “taught itself from scratch how to master the games of chess, shogi (Japanese chess), and Go” (Silver, Hubert, Schrittwieser and Hassabis, 2018). It was first released in 2017 and was implemented using a neural network that “plays millions of games against itself via a process of trial and error called reinforcement learning” (Silver, Hubert, Schrittwieser and Hassabis, 2018).

AlphaZero is significantly more complex than the AI I will develop using convolutional neural networks (the_real_greco, 2019) and Monte-Carlo Tree Search (Silver, Hubert, Schrittwieser and Hassabis, 2018). However the results show that neural networks are a valid approach to not only chess but other board games and was my main inspiration for doing this project.

2.2.2.2 Checkers AI

Checkers AIs are normally overshadowed by chess ones but there has been lots of work on checkers AI over the years too. Chinook is one that uses alpha-beta search and beat a human champion in 1990 (Russell and Norvig, 2010). Since 2007 Chinook was capable of playing perfectly so checkers can be considered a solved game (Russell and Norvig, 2010).

Neural networks have also been used to play checkers as seen in this paper (Chellapilla and Fogel, 1999) with promising results. This example uses both minimax search and neural networks to decide on its moves with many different networks being generated (Chellapilla and Fogel, 1999). This could be an interesting area to explore when developing my agents.

2.2.2.3 Cheskers AI

Unfortunately I was not able to find any implementations of a cheskers AI to draw inspiration from. The one link (Cheskers, n.d.) I did find seemed to be broken but I did find a video (Taraev, 2013) of it playing against a human player on YouTube.

2.3 Technical Infrastructure

2.3.1 Programming Languages

As this is a technical project the choice of languages to use is a very important one. My neural network implementation also relies on me building the minimax AI first so to minimise risk I decided to go with a language I am very familiar with for this section. The languages I decided on are Java for implementing the minimax AI, rules of the game and GUI and Python for the neural network. I believe these languages are good choices for this project and I will go over why in the coming sections.

2.3.1.1 Python

Not only is Python currently the most popular language in the world (Best language for Machine Learning | SpringboardIN Blog, 2020) it is also a very popular choice for machine learning (Best language for Machine Learning | SpringboardIN Blog, 2020). The main reason for this, and why I personally chose it, is its extensive collection of libraries and packages (Best language for Machine Learning | SpringboardIN Blog, 2020). These include libraries specifically for machine learning such as Tensorflow and PyTorch as well as libraries that can help machine learning developers such as pandas and matplotlib. On top of this it is a very flexible language allowing many different approaches and styles so I think it makes a perfect fit for the neural network side of my project.

2.3.1.2 Java

Java is the language I have had most experience with and as a general purpose language (Bloch, 2017) will be useful for many parts of this project. As Java is an object oriented language (Bloch, 2017) it is a natural fit for developing games such as cheskers as each element of the game (pieces, players, the board) can be their own class. This should simplify development and result in more understandable code then if I had chosen a language with a different paradigm.

Java is also a compiled language rather than interpreted (Bloch, 2017) it should run much more efficiently than Python for the minimax AI. Finally Java has the Swing library which makes it easy to make a simple GUI. This project doesn't necessarily need a GUI element but will be a nice addition and is one final reason why I chose to use Java.

2.3.1.3 Other Languages Considered

C++ and Javascript were the other two languages I gave serious consideration too. C++ has the speed advantage over Java and so my minimax AI would be able to evaluate more moves and reach a greater depth on its search tree than a Java implementation would. I decided on Java instead though as I have much more experience with this language and its libraries which should speed up development.

Javascript was the first language I wanted to use as it has a version of Tensorflow and would easily allow my project to be hosted on the internet. This appealed to me as I was hoping to get data to train my neural network from humans playing against my minimax AI. Having access to good machine learning libraries would also mean I would only be using one language for the whole project which would result in a more concise code base. As I thought about the project more I decided involving humans would just add too much complexity and I wanted to focus on the AI so I decided against using Javascript fairly early on.

2.3.2 Libraries

2.3.2.1 Tensorflow

Tensorflow is an open source machine learning library developed by Google (TensorFlow, n.d.). It provides APIs for many different languages including Python, Javascript and C (TensorFlow, n.d.). I will be using the Python one. It is built around the idea of tensors which can be thought of as multidimensional array. For example a vector is a 1d tensor, a matrix is a 2d tensor and so on. Tensors can be used for many different machine learning applications including deep learning which is what I will be using it for (TensorFlow, n.d.). Tensorflow is an incredibly popular library and has even had specific hardware called Tensor Processing Units, TPUs, designed for it (TensorFlow, n.d.).

2.3.2.2 Keras

Keras is a deep learning library built on top of Tensorflow (Keras Team, n.d.). While Tensorflow can be used for many machine learning applications Keras focuses on deep learning and provides easy to use tools for building and training neural networks (Keras Team, n.d.). Since version 2.4 Keras comes packaged with Tensorflow and basically acts as a higher level interface for Tensorflow. Keras also allows saving neural networks in a specific format which can then be loaded by other applications.

I practiced using Keras before starting this project and you can find the code for creating and saving a simple Keras model in section 8.1.1 of this report.

2.3.2.3 Deeplearning4j

Deeplearning4j is an open source machine learning library developed from the Java Virtual Machine, the JVM (Deeplearning4j, n.d.). Since I am developing my machine learning applications in Python I will just be using this library as an easy way to load a Keras model into the JVM.

2.3.2.4 Swing

Swing is a GUI widget library for Java (javax.swing (Java Platform SE 7), 1993) built on top of another library AWT. Although it is old with more applications being developed using JavaFX and GTK it is still used today and is the GUI library I have the most experience with (Donovan, 2019). Technically a GUI is not required for this project so Swing is good enough for my purposes.

I practiced using Swing to make a simple cheskers GUI before starting this project and you can find the code for this in section 8.1.2 of this report. The result is the image on the front page of the report.

2.3.3 Base Code

I did look at many existing chess and checkers implementations for Java such as chesslib (loloof64 / chesslib Download, 2021) but none of them seemed like a good fit. These libraries seemed overly complex for what I wanted to do and would require some hacking to allow both chess like moves for knights and bishops and checkers like moves for pawns and kings. I did find one cheskers implementation in Java I would have liked to look at but unfortunately the link was broken (Cheskers, n.d.). Overall I decided to build my own engine as cheskers isn't too complicated and I felt I could implement a version I had full control over fairly easily and within the time frame.

3 Implementation

This section describes how I went about implementing the project requirements. It is roughly in the order I developed them. Here I will explain how the game and the different agents work as well as why I decided to do things the way I did. During each section I will also discuss how development went and places where I had to rethink what I was doing. All the code developed in this section will be submitted alongside this report.

3.1 System Overview

Figure 18 shows how the software was broken down and which languages each section was written in. This is a helpful diagram for understanding the final software architecture. From the diagram you can see that the game and AIs were both implemented in Java. The game will send the board state to the AIs and the AI will return the move it thinks it should make. The neural network was developed in Python using Keras, saved to a file and then imported into Java using the Deeplearning4j library.

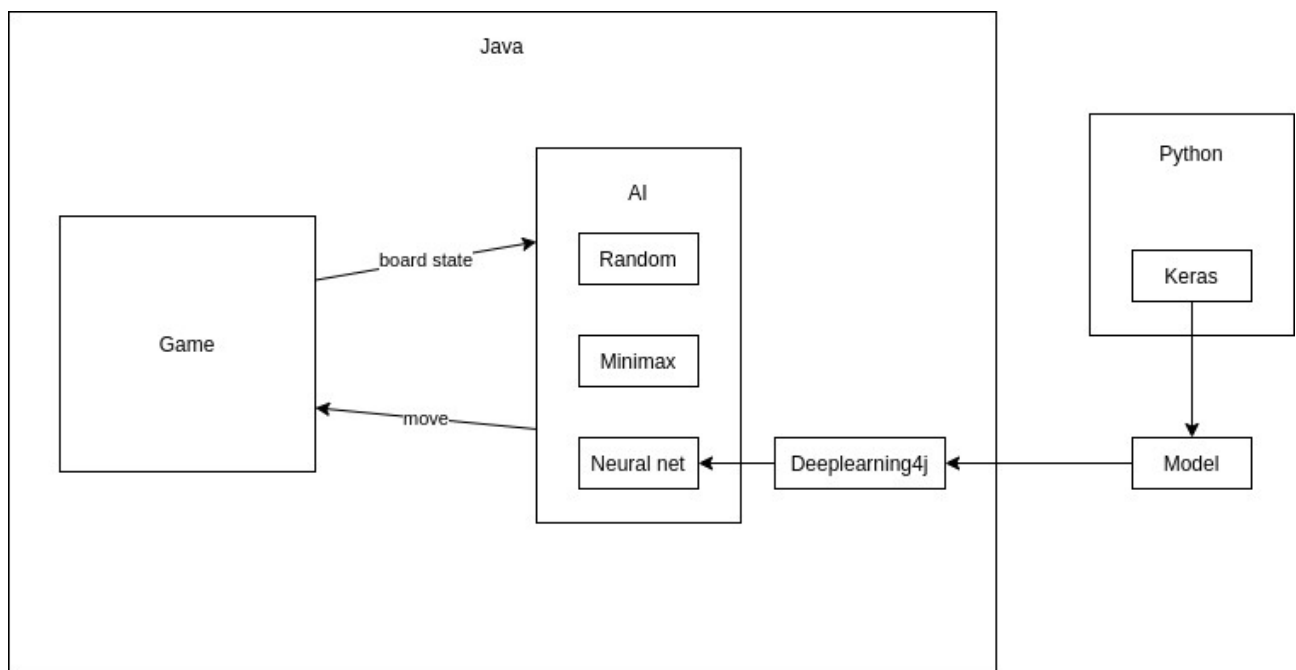


Figure 18 (made by me) – Shows how the game and AIs communicate and what languages will be used.

3.2 Requirements

Figure 19 shows my functional and non-functional requirements. As you can see I achieved most of what the system was meant to do. The first functional requirement, FR-1, is only ‘mostly’ finished as I wasn't able to add the King doing multiple jumps to capture pieces which is explained more in section 3.3.3. FR-5 was a functional requirement I didn't complete at all but was something of a stretch goal. For non-functional requirements some of my minimax agents take quite a long time to pick a move so not all of them meet NFR-3. I will be going over exactly how long each agent took in section 4.2.2.

Overall I am happy with the requirements the final system meets.

ID	Description	MoSCoW Analysis	Achieved?
FR-1	The rules for cheskers must be implemented in software.	Must	Mostly
FR-2	An agent that picks random moves in cheskers could be implemented	Could	Yes
FR-3	An agent using minimax search to pick moves in cheskers must be implemented.	Must	Yes
FR-3.1	The minimax agent should use alpha-beta pruning.	Should	Yes
FR-3.2	A second minimax agent using a different cost function should be implemented	Should	Yes
FR-4	An agent using a multilayer perceptron type of neural network to pick moves in cheskers must be implemented.	Must	Yes
FR-5	An agent using a convolutional neural network to pick to pick moves in cheskers could be implemented.	Could	No
NFR-1	The rules for cheskers should be written in Java.	Should	Yes
NFR-2	The minimax agent should be written in Java.	Should	Yes
NFR-3	The minimax agent should pick moves in a reasonable time.	Should	Mostly
NFR-4	The multilayer perceptron agent should be written in Python.	Should	Yes
NFR-5	The multilayer perceptron agent should pick moves in a reasonable time.	Should	Yes

Figure 19 – Requirements table.

3.3 Game

A full class diagram for the game can be seen in section 8.2.1. Sections of the full diagram are shown throughout this section to help explain the system. This part of development was largely done without much guidance as I was unable to find an existing game implementation I liked, explained in section 2.3.3. I have made a few games in Java using Swing before so I had a good idea of what to do and overall I think it went well.

3.3.1 GUI

I practiced with swing so already had a GUI working as shown in Figure 15. Figure 21 shows the class diagram for the display. The class diagram mentions parts used for human interaction with the GUI such as toButton which will not be discussed until section 3.3.5. As you can see the display is made up of standard Swing components such as JFrames and Jbuttons with a few methods for initialing the GUI.

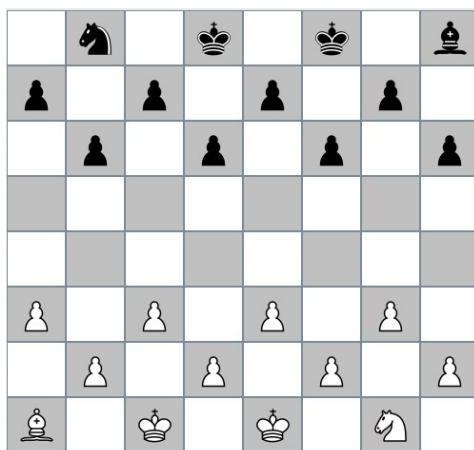


Figure 20 (made by me using (Chess pieces, 2013))

– A checkers board set up for a game.

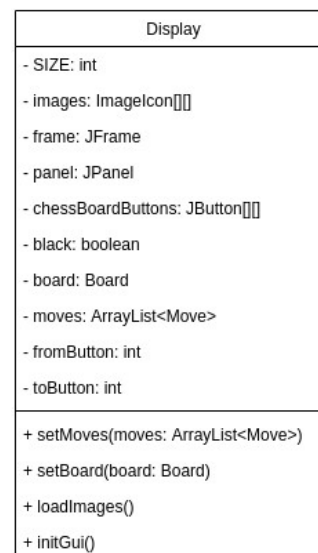


Figure 21 – Display class diagram.

3.3.2 Board

I decided to ignore the display for a while and focus on implementing the structure of the board. As shown in Figure 22, it is a class that holds 64 Squares in a 2d array of dimensions 8x8. Accessing `squares[i][j]` would be the square `i` along the x axis and `j` down the y axis on the board.

The board then has methods for setting things up, adding pieces to squares, returning a copy or flipped version of itself and methods for the AIs discussed in section 3.4. The class ‘Square’ is a simple one which can contain 0 or 1 Pieces (discussed in section 3.3.3).

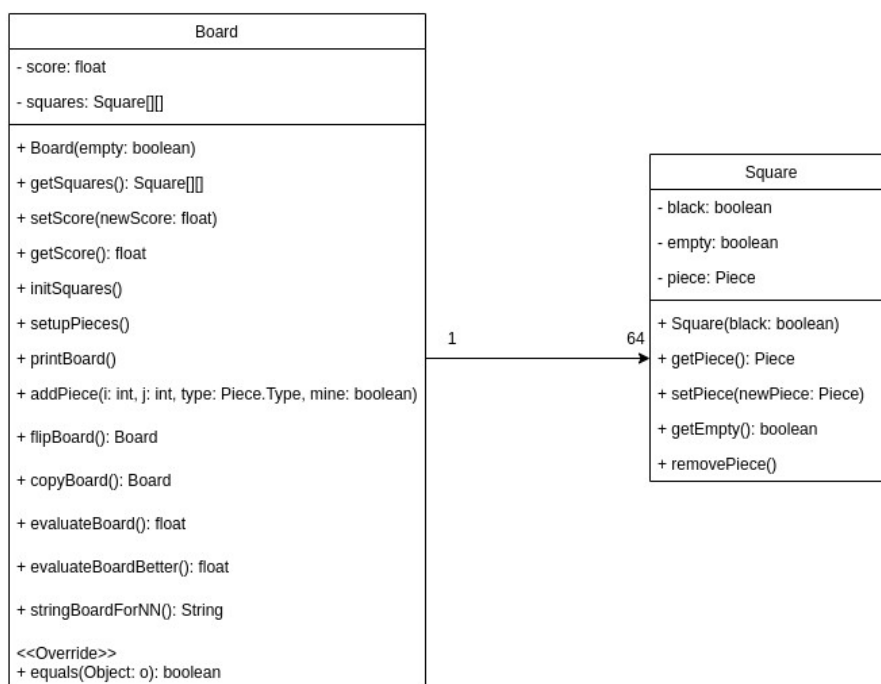


Figure 22 – Board and square class diagrams.

3.3.3 Pieces

This was the first hard part of development. First I set up the class structure I wanted with an abstract class 'Piece' and 4 subclasses for the pawn, knight, bishop and king shown in Figure 23. Each piece has a type, an x and y coordinate and a list of possible moves. There are also 2 booleans, mine is used so the Player class knows which piece is his and mustMove which is used because certain pieces are forced to move in certain conditions.

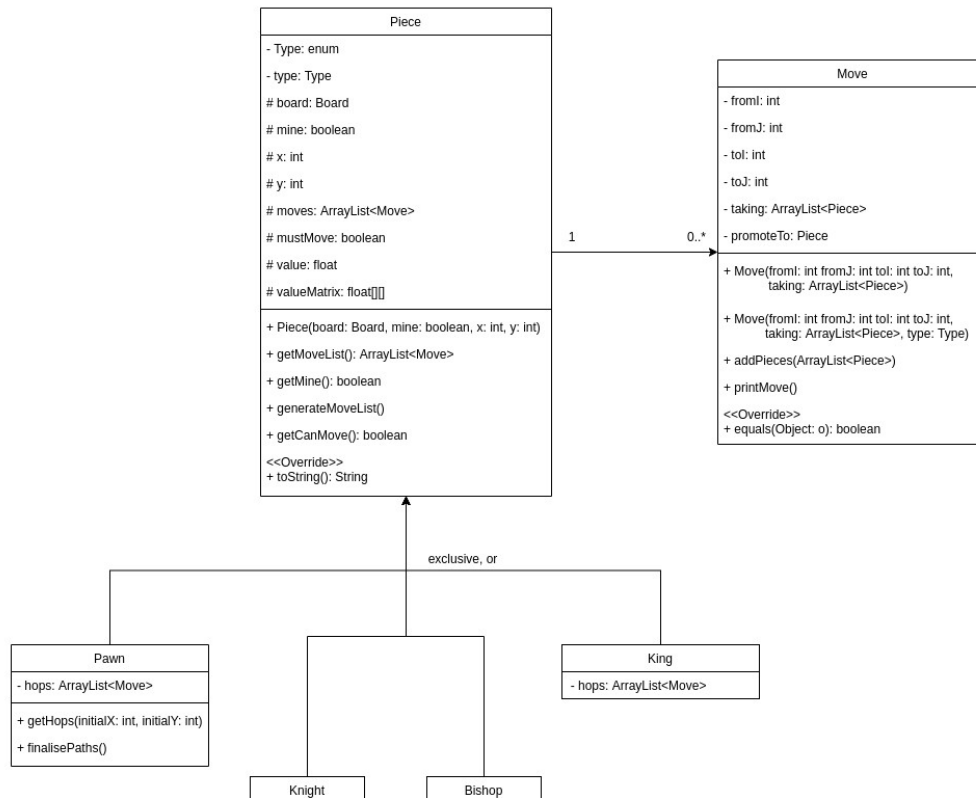


Figure 23 – Piece and Move class diagram.

This set up a good backbone, now I had to implement `generateMoveList()` for each subtype. This was easy with knights and bishops as they move just like regular chess pieces which are easy to implement. Basically just remove the piece from the origin square and add to the destination square overwriting whatever was already there. Pawns and kings were more challenging because they can hop.

Rather than moving to the destination square and the move being over, pawns and kings can do multiple hops, capturing multiple pieces. They are also forced to capture if they can so I used mustMove here. Pawns can only hop forward so for them I used a recursive function which would keep checking the two available squares in front of the pawn until it had to stop hopping. This worked well here but didn't work for kings.

Because kings can hop forward and backwards they can get stuck in loops or cause other bugs when capturing pieces with multiple hops. Figure 24 shows this in an example where the king could follow the arrows around in order or take arrow 5 straight away. There were many similar states causing issues and after working on the kings movement long into the minimax AI stage of development I never found a solution. Eventually I decided to settle on allowing the king to move in any direction (forward and back) but only one hop. The final piece movements can be seen in Figure 25, shows how king and pawn movement changes when there's a piece they can capture.

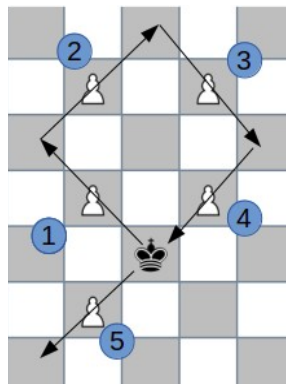


Figure 24 – Shows a state in which the king capturing multiple pieces leads to bugs.

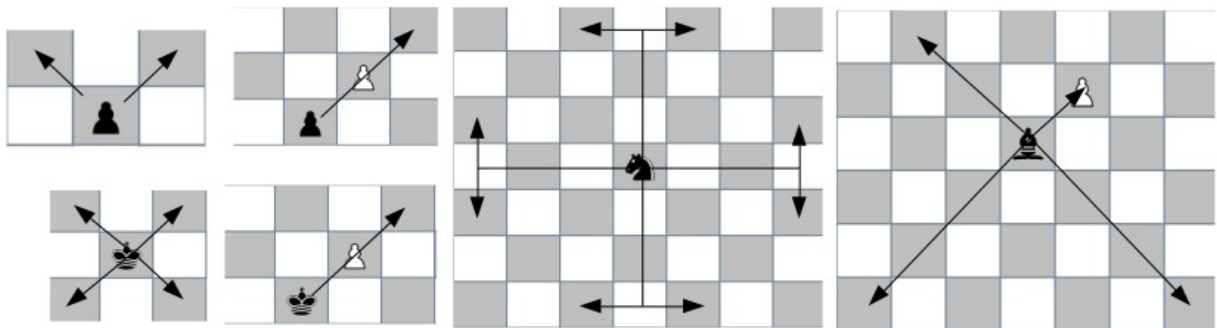


Figure 25 – The final movement rules implemented for each piece.

3.3.4 Player and Game

Finally I tied it all together with a ‘Player’ class and main game loop. Players are initialised with their own board and display. Most of the methods and variables in ‘Player’ in Figure 26 is explained in section 3.4. The main game loop gets the current player to pick a move, make that move and sends the flipped version of the resulting board to the next player . It runs until one of the players hasLost() method returns true which checks for checkmate or 250 moves have been taken to simulate stalemate. If the game ends in stalemate it will also say who was up material when it ended. Figure 26 also serves as a simplified class diagram of the whole game but section 8.2.1 has the full one.

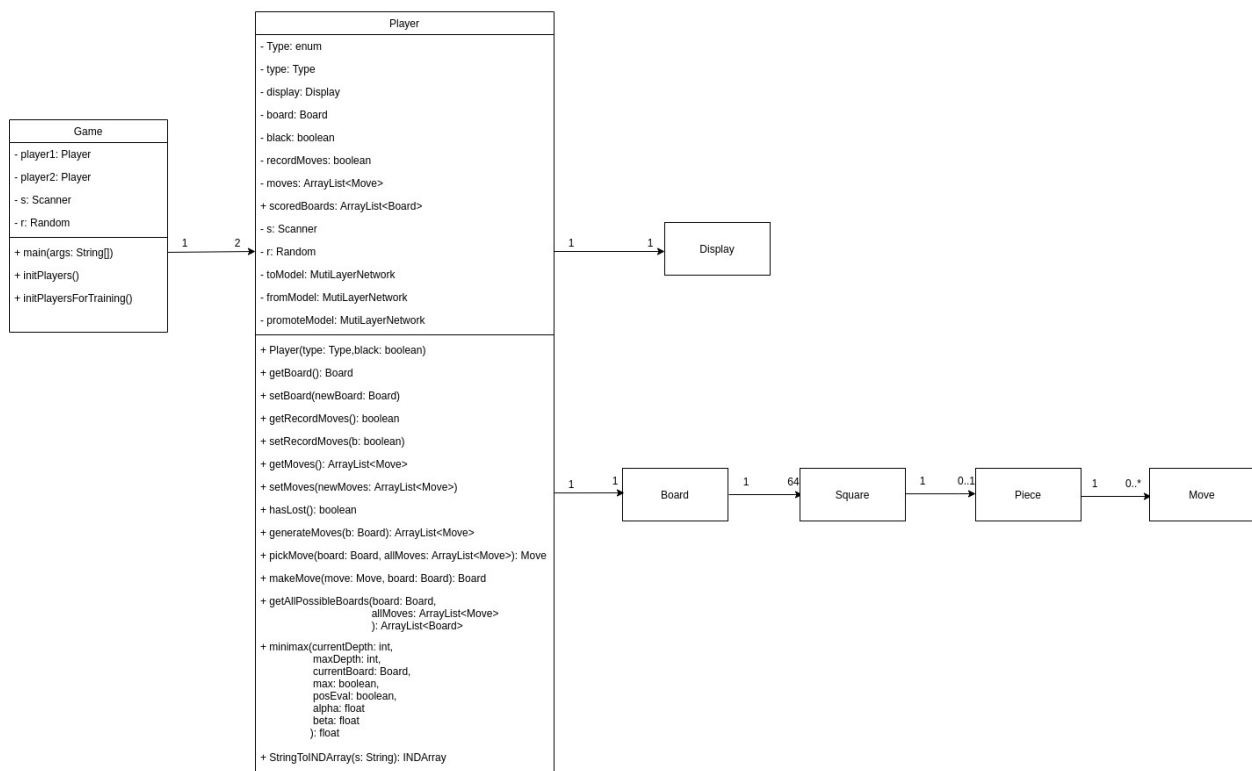


Figure 26 – Simplified full class diagram.

3.3.5 Human Interaction

Going back to the GUI I added human interaction with the game. This was actually done close to the end of the project after I had the minimax agents working. I added the integers toButton and fromButton to ‘Display’ which was shown in Figure 21. These were used to hold the number, between 0 and 64, corresponding to which square the user clicked on. I also added a different colour for which squares the user could click on and a busy wait in the main game loop. The main problem with my UI is that humans cant promote which I discuss more in section 6.2.

3.4 Agents

3.4.1 Random

Now I thought I had the game working accurately I wanted to test it with a simple agent. In the `pickMove()` method, shown in Figure 26, I added a check for if the players type was random. If it is random just pick a random index in the move list and make this move. After some tweaking I had two agents which could play random moves against each other until one of them ended up in checkmate.

3.4.2 Minimax

3.4.2.1 Minimax Simple

Implementing pieces was hard but this was harder. Now I was following a tutorial on implementing a minimax agent for chess (Hartikka, 2017). Also found a forum post (Stadnicki, 2012) and pseudocode (Alpha-beta pruning - Wikipedia, n.d.) to help. I assigned each piece a value based on the standard chess piece values (Soltis, 2020) so each board could be evaluated. The current players pieces added to the score, the other players subtracted. This is a simple evaluation function as it doesn't take into account the positions of the pieces.

I also used a physical board and tried configuration of pieces where it would have to think some moves ahead to test it. Figure 27 shows a configuration where the knight is guaranteed to capture the pawn if it can think two moves ahead. Once it seemed to be working I tried deeper depths until I found a depth that took too long. The effectiveness of the different depths is explained throughout section 4.2. Finally I added alpha-beta pruning with help from the pseudocode (Alpha-beta pruning - Wikipedia, n.d.) and which reduced the amount of boards I would have to evaluate at each depth. Figure 4 in section 2.1.3.1 shows how alpha-beta pruning works.

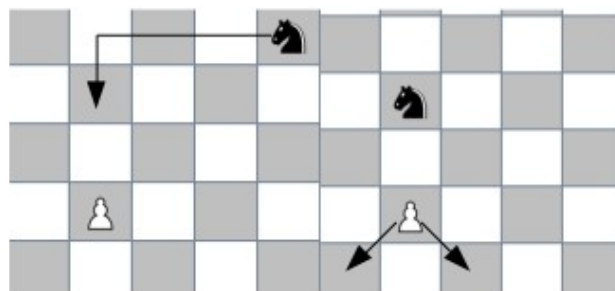


Figure 27 – Shows a move where if the black player thinks ahead they're able to capture the white pawn no matter what move the white player makes.

I now had a minimax agent that seemed to be working. As I will discuss more in section 4.2.1 I don't think the agent is bug-free but it does make good / interesting moves such as the one shown in Figure 27 and often can beat me. Finally I created 3 types of simple minimax agents, denoted MMSimple1, MMSimple2, MMSimple3, which searched at different depths.

3.4.2.2 Minimax Positional

These minimax agents used a different evaluation function which takes the position of each piece into account. I based these positional values on a chess minimax tutorial (Hartikka, 2017) and can be seen in Figure 42. Finally I set up 3 different positional minimax agents similarly to how I set up the simple ones. Figure 28 lists each type of minimax agent and which depth and evaluation function they use. I was playing against the minimax agents throughout development as well as playing them against each other and things seemed to be going well. I then started to collect data for my neural network agents by running 1000 games with a random selection of these agents and printed the moves to file.







	
<pre>[-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0], [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0], [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0], [-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0], [-2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0], [-1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0], [2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 2.0, 2.0], [2.0, 3.0, 1.0, 0.0, 0.0, 1.0, 3.0, 2.0]</pre>	<pre>[-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0], [-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0], [-1.0, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -1.0], [-0.5, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -0.5], [0.0, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -0.5], [-1.0, 0.5, 0.5, 0.5, 0.5, 0.5, 0.0, -1.0], [-1.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, -1.0], [-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]</pre>
	
<pre>[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5], [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5], [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5], [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5], [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5], [-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5], [0.0, 0.0, 0.0, 0.5, 0.5, 0.0, 0.0, 0.0]</pre>	<pre>[-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0], [-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0], [-1.0, 0.0, 0.5, 1.0, 1.0, 0.5, 0.0, -1.0], [-1.0, 0.5, 0.5, 1.0, 1.0, 0.5, 0.5, -1.0], [-1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, -1.0], [-1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0], [-1.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.5, -1.0], [-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]</pre>
	
<pre>[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0], [-4.0, -2.0, 0.0, 0.0, 0.0, 0.0, -2.0, -4.0], [-3.0, 0.0, 1.0, 1.5, 1.5, 1.0, 0.0, -3.0], [-3.0, 0.5, 1.5, 2.0, 2.0, 1.5, 0.5, -3.0], [-3.0, 0.0, 1.5, 2.0, 2.0, 1.5, 0.0, -3.0], [-3.0, 0.5, 1.0, 1.5, 1.5, 1.0, 0.5, -3.0], [-4.0, -2.0, 0.0, 0.5, 0.5, 0.0, -2.0, -4.0], [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]</pre>	<pre>[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0], [1.0, 1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0], [0.5, 0.5, 1.0, 2.5, 2.5, 1.0, 0.5, 0.5], [0.0, 0.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0], [0.5, -0.5, -1.0, 0.0, 0.0, -1.0, -0.5, 0.5], [0.5, 1.0, 1.0, -2.0, -2.0, 1.0, 1.0, 0.5], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]</pre>

Figure 42 – Positional piece score tables used for chess AIs.

3.4.3.1 Neural Network 1

Now I switched to python using jupyter-notebook (Project Jupyter, n.d.). I Imported the move data I had saved (explained in section 3.4.3) using pandas, a popular python library. Figure 30 shows the first 5 lines of initial dataframe. Based on a blogpost (the_real_greco), 2019) I decided to train three networks for each agent, one for picking the piece to move, one for picking where to move to and one for dealing with promotion. Based on this I split the dataframes and did some final parsing of the data before creating 3 models.

The toModel and fromModel take 512 inputs, a bitboard, has 64 outputs, a square. The promoteModel takes 512 inputs, a bitboard, and has 3 outputs, one for knight, bishop or king. Initially I was just trying to get something working so I didn't care about the shape beyond the inputs and outputs. They all use optimizer 'stochastic gradient descent', loss function 'mean squared error' based on the Keras code in section 8.1.1 and have two hidden layers with 100 neurons. The optimizer is what the neural network uses to try and improve and the loss function tells it how much it needs to improve. Figure 31 shows the basic shape of all the final neural networks used with only the number of neurons in each hidden layer changing.

[illegible]

Figure 30 – Initial dataframe structure.

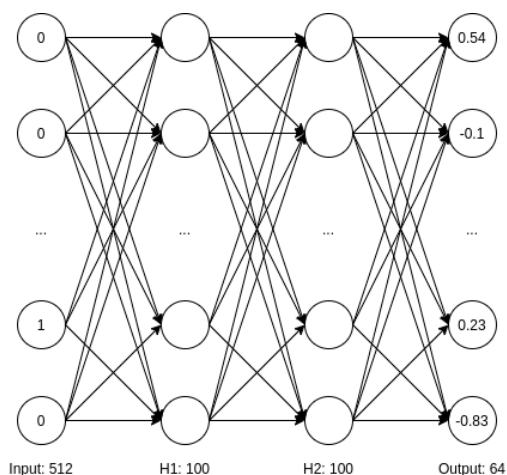


Figure 31 – Diagram of neural network structure.

I got around 61% accuracy with toModel, 33% with fromModel and 98% from promoteModel. This accuracy rate is how close the neural network got to the correct output given an input. These accuracy rates don't show how the network will do with new unlabeled data and at this point I just wanted a neural network agent to test so I saved these models to file in the h5 format to import into my existing project. After some fiddling including converting my project to a Maven project (Porter, Zyl and Lamy, 2021) I was able to import these models into java. As I expected the final agent was not very good and lost to the random agent most of the time.

3.4.3.2 Neural Network 2

For neural network agent 2, NN2, I knew I needed to improve on NN1. First I analysed the initial 1000 minimax games generated in section 3.4.2.2 closer. Based on this I tried to pick out the best agents or most interesting combination of agents. These results are shown in Figure 32 which shows the matrix of wins and loses against the 6 different minimax agents and based on this I reached the conclusions in Figure 33 and recorded moves based on this. The easiest way to read the matrix is line by line to see how many times that agent won overall. For example on line 1 you can see mmSimple1 beat mmSimple1 0 times overall, beat mmPositional1 42 times overall, etc. Figure 32 doesn't take into account how many games were played, just the overall wins and loses. If I took the total number of games into account I may have came to different conclusions so Figure 32 could be misleading. In section 4.2.1 I will be using a different matrix notation which takes into account the number of games played.

mmSimple1	[0	42	0	8	7	9]
mmPositional1	[-42	0	-13	5	-14	-9]
mmSimple2	[0	13	0	10	7	1]
mmPositional2	[-8	-5	-10	0	1	10]
mmSimple3	[-7	14	-7	-1	0	-2]
mmPositional3	[-9	9	-1	-10	2	0]

Figure 32 – Matrix of overall wins against each agent.

```

#Conclusions:
#mmSimple1 really good against mmPositional1
#mmPositional1 not good
#mmSimple2 is good against mmPositional1 and mmPositional2
#mmPositional1 is good against mmPositional1
#mmSimple3 is good against mmPositional1
#mmPositional3 pretty bad, best against mmPositional1 though

#To get good data:
#run 500 games mmSimple1 vs mmPositional1, record mmSimple2
#run 250 games mmSimple2 vs mmPositional1, record mmSimple2
#run 250 games mmSimple2 vs mmPositional2, record mmPositional2
#run 250 games mmPositional2 vs mmPositional3, record mmPositional2
#run 250 games mmSimple3 vs mmPositional1, record mmSimple3
#run 100 games mmPositional3 vs mmPositional1, record mmPositional3

```

Figure 33 – My conclusions based on Figure 32.

I also wanted to improve on the structure including how many hidden layers and neurons in each layer. I decided on two hidden layer with neurons equal to the mean of size of the input and output layers because of this post (Hyndman, 2010). I also decided to stick with mean squared error as my loss function due to this article (Brownlee, 2020) and changed the optimizer to Adam based on this blogpost (Quick Notes on How to choose Optimizer In Keras | DLology, 2018).

I got around the same accuracy as I did with the first neural network agent but with the new data and improved structure I wanted to save these models and import them to java again to see if I had made improved. NN2 still gets stuck in loops with NN1 but now it seemed to be better against the random agent, my final evaluation in section 4.2.1 contradicts this however.

This was the point when I moved from development to focusing on this report. I could have made more different neural network agents with the data from NN1 and structure from NN2 and so on but I was running out of time and believed NN2 was good enough. This is future work which I will go over in section 6.2.

4 Testing and Evaluation

My project evaluation was based on my evaluation plan in section 5.4. All the tables used to create the diagrams seen in this section can be found in sections 12.2.2 – 12.2.5 where you can see the exact numbers generated.

4.1 Game

This project focused mainly on the different agents developed so I didn't evaluate the game thoroughly. If I had done some user evaluations of the game the results would be discussed here.

Overall there are many things about the core game that could be improved. In terms of the whole project the kings limited movement, explained in section 3.3.3, is probably the main issue with the game as this affects the agents too. Purely talking about the game implementation other issues are that human players cant pick what their pawns get promoted to and the GUI highlighting squares breaks with two human player. These are all things I would like to improve on if I had more time and are discussed more in section 6.2.

4.2 Agents

4.2.1 Win Rates

Figure 34 shows my agents win rates against the agent that just moves pieces randomly. This is a good benchmark for seeing the agents playing strength. It shows that all the minimax agents beat the random agent almost every time however both neural network agents lose around 74% of the time so are worse than random. While I was developing NN2 I tested it against the random agent a few times and believed it was winning more often than NN1 however while doing the final evaluation this doesn't seem to be the case. I only tested it a handful of times so probably was just getting lucky.

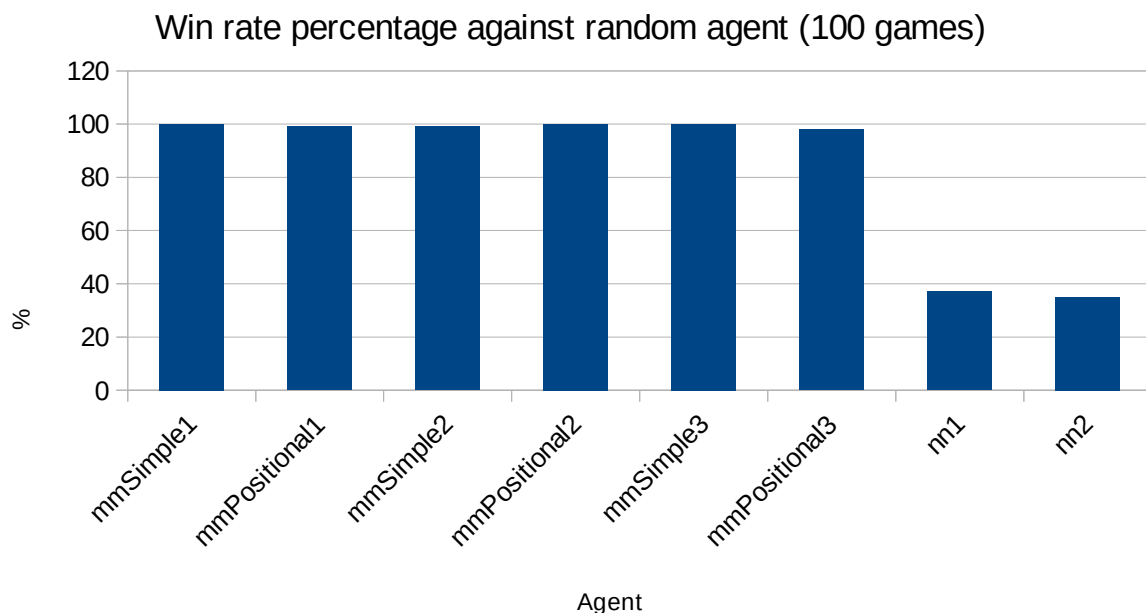


Figure 34 – Win rate of each agent versus the random agent.

Figure 35 shows the win rates of each agent versus each other agent. I assigned the agents randomly and ran 1000 total games to get these results. The final values are how many times that agent won versus how many games were played so a value of 0.5 means the agent won exactly half of its games against the other agent. By averaging the normalized numbers on each row you can get a score of how well the agent performs compared to the others. This is shown in Figure 41.

#random	[0.5	0.	0.	0.	0.048	0.	0.	0.64	0.6]
#mmSimple1	[1.	0.5	0.619	0.391	0.563	0.6	0.433	1.	1.]
#mmPositional1	[1.	0.333	0.462	0.207	0.5	0.273	0.286	1.	1.]
#mmSimple2	[1.	0.609	0.655	0.5	0.478	0.476	0.706	1.	1.]
#mmPositional2	[0.952	0.25	0.333	0.478	0.469	0.387	0.588	0.96	1.]
#mmSimple3	[1.	0.32	0.545	0.476	0.548	0.5	0.517	1.	0.944]
#mmPositional3	[1.	0.567	0.429	0.235	0.294	0.483	0.5	1.	0.917]
#nn1	[0.36	0.	0.	0.	0.	0.	0.	0.	0.]
#nn2	[0.4	0.	0.	0.	0.	0.056	0.	0.	0.]

Figure 35 – Matrix showing the win rate against each other agent.

random	0.197
mmSimple1	0.678
mmPositional1	0.562
mmSimple2	0.714
mmPositional2	0.602
mmSimple3	0.65
mmPositional3	0.603
nn1	0.04
nn2	0.05

Figure 41 – The win rate for each agent with the best and worst in bold, see Figure 28 for the differences between each agent.

From Figure 41 you can see mmSimple2 is the best with an average win rate of 71% and nn1 is worst with a 5% win rate. The positional agents consistently score less then the simple ones too so I think the piece value matrices in Figure 42 might not be appropriate for cheskers. After finding mmSimple2 to be the best agent I then played 10 games against it and lost all but one.

Strangely the minimax agents don't seem to get better as the search depth increases with mmSimple2 searching to depth 6 and being better then mmSimple3 which searches to 8. Something that could improve this is reducing the score of each board depending on how deep it is in the search tree. I think the agents that search deeper are attempting to reach really good boards 8 moves in the future but their opponent doesn't do exactly what that expected so it fails. Apart from this it could just be a bug.

Figure 35 also shows us interesting things about the neural network agents. Firstly the cells for nn1 versus nn2 in the bottom right are all zeros because these against always end up in a loop and no one wins. In the neural network versus neural network games whichever agent went first, the black player in cheskers, always had more or better pieces at the end. This means its more of an advantage to go first then to be nn2 so nn2 isn't much of an improvement over nn1 at all and ultimately they are both worse then random. I think working on the neural network structure a bit more would help improve these agents but the amount of good data to provide them with is the main issue.

4.2.2 Time taken

All these times were computed on a laptop with an 1.70GHz 4 core CPU and 8gb of RAM. Figure 36 shows the average time taken to pick a move for each agent as well as the standard deviation using error bars. As you can see the time taken increases dramatically for each additional depth the minimax agent searches while positional board evaluation doesn't affect it much. I decided depth 8 with alpha-beta pruning was about as long as allowed as this can be fast if there's a forced move but can take about 10 seconds for a single move sometimes. This difference in time when there's a forced move is what causes the huge standard deviations seen in Figure 36. The random and neural network agents are very quick compared to the minimax agents as they don't actually have to evaluate any boards except the current one.

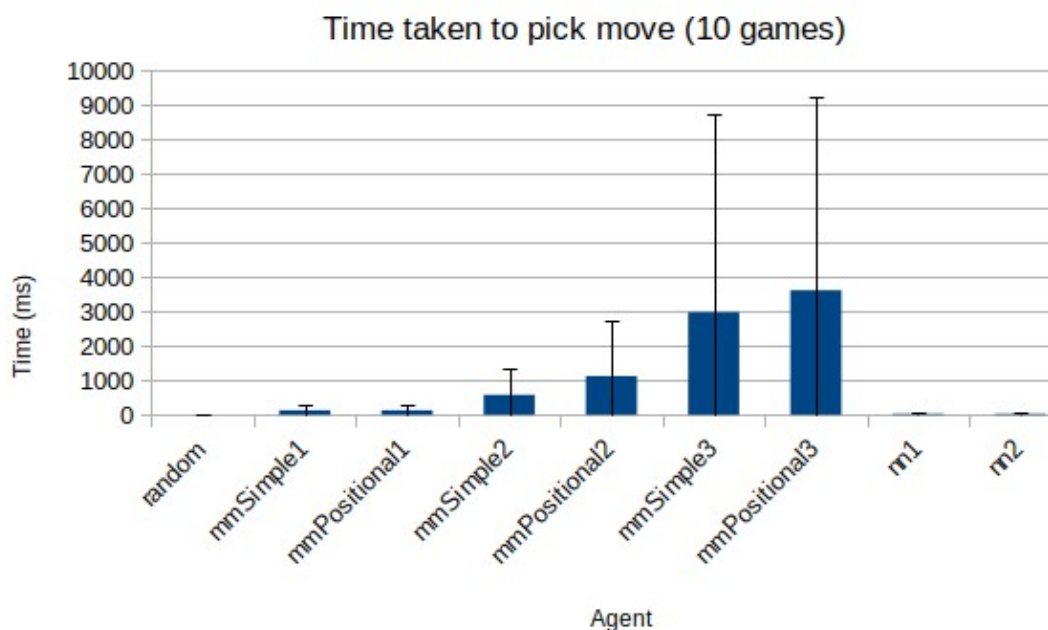


Figure 36 – Shows average time taken for each agent and the standard deviation.

Figure 37 uses a logarithmic scale and shows how the time taken decreases drastically when pruning is not used. I only tested the simple minimax agents as the other minimax agents use the same depths with different evaluation functions which doesn't affect the time taken by much. Overall using pruning results in between a 68% and 99% decrease in time taken.

Average time per move, with and without pruning (10 games)

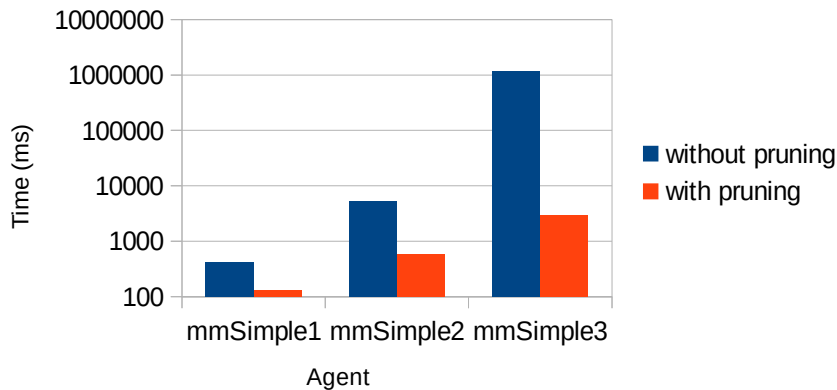


Figure 37 – Logarithmic diagram showing how pruning affects time taken.

4.2.3 Boards evaluated

As you can see from Figure 38 the number of boards evaluated at each depth increases significantly. This is also what causes the increase in time taken. Similar to how forced moves decrease the average time taken they also decrease the average amount of boards evaluated. Figure 39 uses a logarithmic scale and shows how alpha-beta pruning decreases the number of boards evaluated which is similar to Figure 37 with the amount of boards evaluated without pruning dwarfing the number evaluated with pruning.

Average number of boards evaluated per move (10 games)

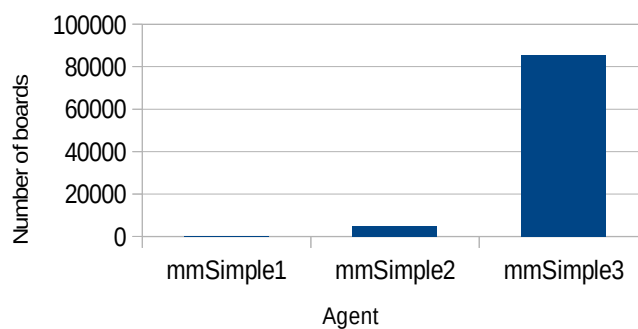


Figure 38 – Shows how the number of boards evaluated increases with the depth the agent searches to.

Average number of boards evaluated per move, with and without pruning (10 games)

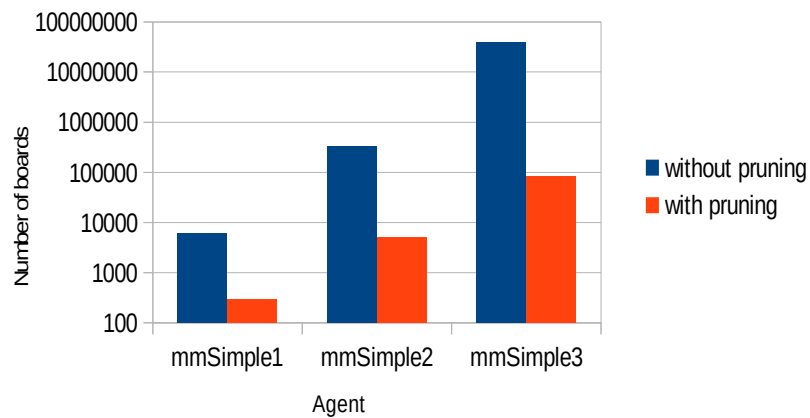


Figure 39 – Logarithmic diagram showing how pruning decreases the number of boards evaluated at each turn.

4.3 Conclusions

In summary mmSimple2, which searches at depth 6 and uses a simple but fast static evaluation function is the best agent. It wins against me 90% of the time, a random agent 99% of the time and against all other agents 74% of the time. It picks moves with an average time of 579ms and a max time of about 5 seconds so is fast enough to run 100s of games with and play against a human opponent. Overall I am very proud of this agent.

I am disappointed I wasn't able to build a neural network agent that could compete with the minimax ones but I think this is largely due to bad data. If I had more time I would like to continue working on this and also look into why the minimax that search past depth 6 get worse. This is discussed more in section 6.2.

5 Project Plan

5.1 Methodology

I used an agile like methodology for developing this software. In my case this mostly consisted of sprints each week to try and get a new prototype built or feature added. Because of the short time frame for this project and the amount of background research I did before starting the project this seemed like a good methodology. This method also means I should have something to show to my supervisor every week which will help with getting feedback and improving the overall quality of the project.

For the most part I was able to stick to this methodology by doing a task each week. Tasks that took longer then expected were implementing the full piece movement for the game and working on the minimax agents. Full movement rules for each pieces were still being worked on long into development of the minimax agents and minimax development continued basically until the end of the project. Thankfully I assigned the weekly tasks such that most of them would take less then a week which gave me time to catch up if there were issues.

5.2 Timetable

As stated in section 1.3 I broke the project down into tasks to achieve in term one and term two. See figures 16 and 17 below for a visual breakdown of the timetable for this project.

WBS	Name	Start	Finish	Work
1	Term 1	Sep 14	Dec 10	89d
1.1	Preliminary work	Sep 14	Oct 23	45d
1.1.1	Decide on project	Sep 14	Sep 25	10d
1.1.2	Find supervisor	Sep 14	Sep 25	10d
1.1.3	Conduct literature review	Sep 28	Oct 23	20d
1.1.4	Fill out ethics form	Oct 19	Oct 23	5d
1.2	Work on prototype	Oct 26	Nov 6	10d
1.3	Report introduction	Oct 26	Oct 30	5d
1.4	Literature review structure	Nov 2	Nov 6	5d
1.5	Report first draft	Nov 9	Nov 27	15d
1.6	Improve report	Nov 30	Dec 9	8d
1.7	Submit report	Dec 10	Dec 10	1d
2	Christmas break	Dec 14	Jan 8	20d
2.1	Implement movement for all pieces	Dec 14	Dec 25	10d
2.2	Implement random AI	Dec 28	Jan 1	5d
2.3	Intergrate UI	Jan 4	Jan 8	5d
3	Term 2	Jan 11	May 13	129d
3.1	Dissertation draft	Jan 11	Mar 19	50d
3.2	Implement minimax AI	Jan 11	Feb 12	25d
3.2.1	Implement 1st minimax AI	Jan 11	Jan 22	10d
3.2.2	Add alpha beta pruning	Jan 25	Jan 29	5d
3.2.3	Implement 2nd minimax AI	Feb 1	Feb 5	5d
3.2.4	Save the games played by these AIs	Feb 8	Feb 12	5d
3.3	Implement neural network AI	Feb 15	Mar 19	25d
3.3.1	Build network	Feb 15	Feb 26	10d
3.3.2	Train network	Mar 1	Mar 12	10d
3.3.3	Improve network	Mar 15	Mar 19	5d
3.4	System evaluation	Mar 22	Apr 2	10d
3.5	Finalise Dissertation	Apr 5	Apr 21	13d
3.6	Submit dissertation	Apr 22	Apr 22	1d
3.7	Online presentation	May 7	May 13	5d

Figure 16 – Timetable.

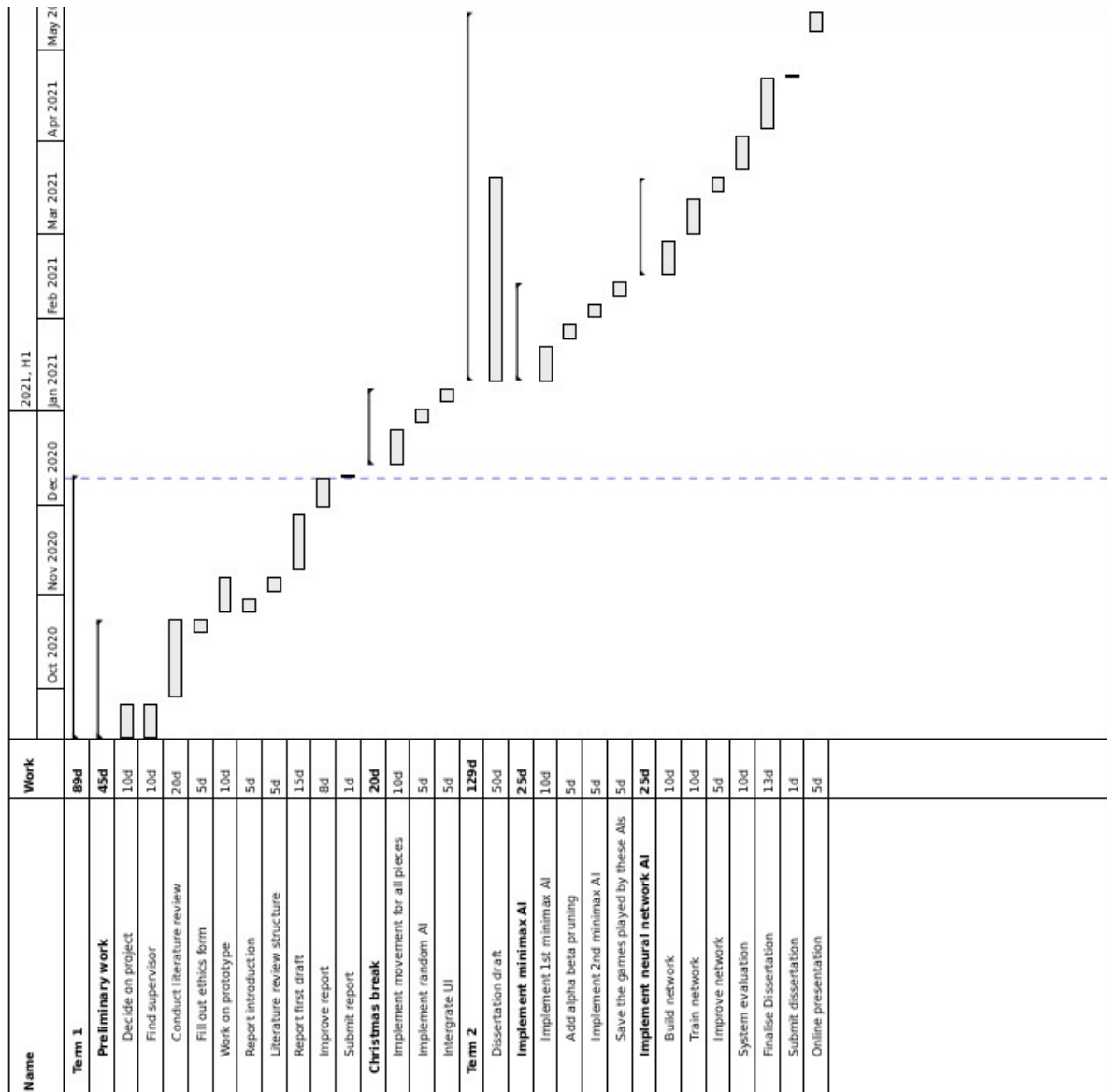


Figure 17 – Gantt chart.

5.3 Risk Analysis

Figure 40 shows the risk analysis I performed before starting the project to help minimise the risks. None of the risks led to significant problems during the project but keeping them in mind and sticking to the timetable in section 5.2 helped a lot.

Risk	Likelihood	Impact	Mitigation strategy
Losing digital data.	Moderate	Moderate	Frequent backups, both physical and online.
Running out of time.	Moderate	High	Sticking to the proposed timetable, see section 5.2.
Being unable to implement the game logic.	Low	High	I have experience in building games in Java so I don't see this being to much of an issue. I could strip down the game code to the bare essentials if it does start becoming an issue.
Being unable to implement the minimax agent.	Low	High	I have implemented a minimax agent for tic-tac-toe in previous years so this should be easily achievable. Doing a thorough literature review before starting should help mitigate this risk.
Being unable to implement the neural network agent.	Moderate	Moderate	I have never implemented neural networks before so this is a moderate risk. If I am unable to do this I could pivot to just comparing different minimax agents, not ideal but should still yield interesting results.
Not having access to a powerful machine.	Low	Low	Due to covid I will not be going into uni to use the machines there but I can remotely log in to use them.

Figure 40 – Risk analysis table.

5.4 Evaluation Plan

The main evaluation strategy will be looking at the win rate of each of the agents. As a benchmark for this I will be implementing an agent which just picks moves at random and recording games this agent plays against itself. When I implement the minimax agent I can then check its win rate and hopefully it is higher than the random agent. I will then do a similar thing with the neural network agent, comparing it's win rate to both the random and minimax agents.

In addition to the win rate I will also be using time taken to pick a move as an evaluation metric. This can be measured easily by simply recording the time before a move and the time after. If my minimax agent takes too long to pick a move I can simply reduce the depth to which it searches but I'm unsure how I would address this with the neural network. Possible solutions are to limit the number of hidden layers and the neurons in this layer.

Using these two measures I should be able to do a systems evaluation to accurately assess how well each of my agents do. I can then compare and contrast to get my final results.

5.5 Professional, legal, ethical and social issues

5.5.1 Professional issues

In order to maintain professionalism through the course of this project I made sure to stick to the BSc code of conduct (Bsc.org, 2021) and reference and credit anyone who's work I made use of. This can be seen in this report as well as the code. I also had regular meetings with my supervisor who has more experience with these issues so he was able to help point me in the right direction when I needed help.

5.5.2 Legal issues

This project will be using software developed by other people, mainly in the form of libraries, so there are many legal issues to consider. The main one is what kind of copyright license I should use. I want my code to be freely available for people to use and edit after I am done so I have decided on using the Apache 2.0 license (Apache License, Version 2.0, 2019). This is the same license used by Tensorflow and deeplearning4j which are two libraries I will be using. This is included in the code under a file called LICENSE and in section 8.3.

5.5.3 Ethical and social Issues

As this project does not involve any humans there are no real ethical or social issues to consider.

6 Conclusions

6.1 Summary of Project

As stated in section 1.2 my aim for this project was to compare and contrast different AI techniques for the board game cheskers by achieving the 3 main objectives, develop a minimax, develop a neural network agent and compare the agents. Overall I think I achieved this aim by doing a lot of research before starting the project and working on it a little everyday. Unfortunately I didn't completely meet all the requirements, most annoyingly failing to implement full king movement, and my agents could definitely be improved. This potential future work is discussed in section 6.2.

My final results show a wide range of agents, some performing better then others. The best overall agent was the minimax agent mmSimple2, which searches at depth 6 and ignores the position of pieces on the board. Surprisingly this is better then agents which search deeper and evaluate boards based on the position of pieces. It wins against me 90% of the time, a random agent 99% of the time and against all other agents 74% of the time. The worst agent was my first neural network agent, nn1, which loses to the random agent 37% of the time and against all other agents 4% of the time.

6.2 Future Work

No project like this is ever really finished and there are lots of things I would like to continue working on.

One of the first would be refactoring the code so the Player class has subclasses for a human player, a minimax player and so on. This would make the code much easier to maintain. After this I think getting the full king movement working is the most important as this affects all player types. Once the code has been refactored and the full piece movement implemented I would like to focus on the GUI more as this was not a main focus of the project.

To run the game you have to run it in a terminal and select the player type from a list. I would like to add a menu so users could select the player types as well as other possible options without having to use the terminal. This would help to make the game more user friendly. Two related issues with the GUI are that if there are two or more moves that start and end in the same place human players cant choose which of these moves to take and they also cant pick what their pawns get promoted to. These issues are caused by the same thing, users can only click on the board squares to input their move. To fix these I'd need an additional GUI component where users could select the correct move after clicking the start and end square. One final issue is the GUI highlighting which squares the user can click on also doesn't work properly when you have two human players.

Once I am happy with the GUI I would focus on the agents again. I would first like to build a new neural network agent, nn3, with the data used to build nn2 and the structure of nn1. This is because I changed both the data fed into the network and the structure between developing nn1 and nn2 so I'm not sure if the differences are down to the data or structure. After this I'd like to add a feature to the minimax agents where boards are worth less the deeper they are in the search tree to see if this improves the agents which search deeper.

Finally I'd like to keep building new neural network agents using different structures, model parameters and input data. An interesting direction to explore could be using convolutional neural networks which are commonly used for image recognition. These are used by the chess AI alphaZero (the_real_greco, 2019) and were briefly discussed in section 2.1.5. Continuing to collect data while improving my other agents could also improve the neural network agents performance and I would hope to get one that can beat the random agent over 50% of the time.

7 References

(the_real_greco), W., 2019. Understanding AlphaZero: A Basic Chess Neural Network. [online] Chess.com. Available at: <https://www.chess.com/blog/the_real_greco/understanding-alphazero-a-basic-chess-neural-network> [Accessed 9 October 2020].

2006. En passant. [image] Available at:

<https://commons.wikimedia.org/wiki/File:Ajedrez_captura_al_paso_del_peon.png> [Accessed 21 April 2021].

Apache.org. 2019. Apache License, Version 2.0. [online] Available at:

<<https://www.apache.org/licenses/LICENSE-2.0>> [Accessed 5 December 2020].

Bcs.org. 2021. [online] Available at: <<https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/>> [Accessed 9 December 2020].

Bloch, J., 2017. Effective Java. 3rd ed.

Bodlaender, H., 1995. Cheskers. [online] Chessvariants.com. Available at:

<<https://www.chessvariants.com/crossover.dir/cheskers.html>> [Accessed 9 October 2020].

Brownlee, J., 2020. How to Choose Loss Functions When Training Deep Learning Neural Networks.

[online] Machine Learning Mastery. Available at: <<https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/>> [Accessed 23 March 2021].

Chellapilla, K. and Fogel, D., 1999. Evolving neural networks to play checkers without relying on expert knowledge. IEEE Transactions on Neural Networks, 10(6), pp.1382-1391.

Coetzee, D., 2005. Binary Search Tree. [image] Available at:

<https://commons.wikimedia.org/wiki/File:Binary_search_tree.svg> [Accessed 9 October 2020].

Copeland, J., 2006. The Modern History of Computing (Stanford Encyclopedia of Philosophy). [online]

Plato.stanford.edu. Available at: <<https://plato.stanford.edu/entries/computing-history/#MUC>> [Accessed 25 October 2020].

David, A., 2019. World's oldest surviving chess piece unearthed in Jordan. [online] haaretz.com. Available

at: <<https://www.haaretz.com/archaeology/.premium.MAGAZINE-world-s-oldest-surviving-chess-piece-unearthed-in-jordan-1.8201206>> [Accessed 21 November 2020].

David, O., Netanyahu, N. and Wolf, L., 2016. DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess. Artificial Neural Networks and Machine Learning – ICANN 2016, pp.88-96.

Deeplearning4j.org. n.d. Deeplearning4j. [online] Available at: <<https://deeplearning4j.org/>> [Accessed 1 December 2020].

Deepmind. n.d. AlphaGo: The story so far. [online] Available at: <<https://deepmind.com/research/case-studies/alphago-the-story-so-far>> [Accessed 13 November 2020].

Dlology.com. 2018. Quick Notes on How to choose Optimizer In Keras | DLology. [online] Available at: <<https://www.dlology.com/blog/quick-notes-on-how-to-choose-optimizer-in-keras/>> [Accessed 23 March 2021].

Docs.oracle.com. 1993. javax.swing (Java Platform SE 7). [online] Available at:

<<https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>> [Accessed 1 December 2020].

Donovan, L., 2019. wockradd/Blackjack. [online] GitHub. Available at:

<<https://github.com/wockradd/Blackjack>> [Accessed 5 May 2019].

En.wikipedia.org. n.d. Alpha–beta pruning - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning> [Accessed 11 January 2021].

En.wikipedia.org. n.d. Mechanical Turk - Wikipedia. [online] Available at:

<https://en.wikipedia.org/wiki/Mechanical_Turk> [Accessed 20 November 2020].

Encyclopedia Britannica. 2019. Checkers | game. [online] Available at:

<<https://www.britannica.com/topic/checkers>> [Accessed 15 November 2020].

Ensor, T., 2020. The AI renaissance: why it has taken off and where it is going. [online] Cambridge

Consultants. Available at: <<https://www.cambridgeconsultants.com/insights/whitepaper/ai-renaissance-why-it-has-taken-and-where-it-going>> [Accessed 24 October 2020].

Hartikka, L., 2017. A step-by-step guide to building a simple chess AI. [online] freeCodeCamp.org. Available

at: <<https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/>> [Accessed 9 October 2020].

Hyndman, R., 2010. How to choose the number of hidden layers and nodes in a feedforward neural

network?. [online] Cross Validated. Available at: <<https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw#:~:text=There%20is%20currently%20no%20theoretical,architectures%20with%20various%20hidden%20layers.>> [Accessed 11

January 2021].

jitpack. 2021. loloo64 / chesslib Download. [online] Available at: <<https://jitpack.io/p/loloo64/chesslib>>

[Accessed 13 November 2020].

Jupyter.org. n.d. Project Jupyter. [online] Available at: <<https://jupyter.org/>> [Accessed 15 February 2021].

Knudsen, F., 2020. Deep Blue | Down the Rabbit Hole. [online] Youtube.com. Available at: <<https://www.youtube.com/watch?v=HwF229U2ba8>> [Accessed 30 October 2020].

Komendantskaya, E., 2021. Data mining lecture.

Pathguy.com. n.d. Cheskers. [online] Available at: <<http://www.pathguy.com/chess/Cheskers.html>> [Accessed 2 October 2020].

Porter, B., Zyl, J. and Lamy, O., 2021. Maven – Welcome to Apache Maven. [online] Maven.apache.org. Available at: <<https://maven.apache.org/>> [Accessed 23 March 2021].

Russell, S. and Norvig, P., 2010. Artificial intelligence. 3rd ed.

Silver, D., Hubert, T., Schrittwieser, J. and Hassabis, D., 2018. AlphaZero: Shedding new light on the grand games of chess, shogi and Go. [online] Deepmind. Available at: <<https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go>> [Accessed 15 November 2020].

Soltis, A., 2020. Chess | game. [online] Encyclopedia Britannica. Available at: <<https://www.britannica.com/topic/chess>> [Accessed 15 November 2020].

Springboard Blog. 2020. Best language for Machine Learning | SpringboardIN Blog. [online] Available at: <<https://in.springboard.com/blog/best-language-for-machine-learning/>> [Accessed 13 November 2020].

Stadnicki, S., 2012. How to utilize miniMax algorithm in Checkers game. [online] Game Development Stack Exchange. Available at: <<https://gamedev.stackexchange.com/questions/31166/how-to-utilize-minimax-algorithm-in-checkers-game>> [Accessed 11 January 2021].

Team, K., n.d. Keras: the Python deep learning API. [online] Keras.io. Available at: <<https://keras.io/>> [Accessed 1 December 2020].

TensorFlow. n.d. TensorFlow. [online] Available at: <<https://www.tensorflow.org/>> [Accessed 1 December 2020].

Witten, I. and Frank, E., 2011. Data mining. 3rd ed. San Francisco, Calif.: Morgan Kaufmann.

Yourturnmyturn.com. n.d. YourTurnMyTurn.com: checkers rules. [online] Available at: <<https://www.yourturnmyturn.com/rules/checkers.php>> [Accessed 9 October 2020].

Taraev, A., 2013. cheskers. [online] Youtube.com. Available at: <<https://www.youtube.com/watch?v=HNIjl4vJI0s>> [Accessed 2 October 2020].

2013. Chess pieces [image] Available at: <https://commons.wikimedia.org/wiki/Category:SVG_chess_pieces> [Accessed 20 November 2020].

8 Appendices

8.1 Initial Code

8.1.1 Simple Keras Model

```
import tensorflow as tf
import numpy as np
from tensorflow import keras

#define our model
model = keras.Sequential()
#add one layer, takes 1 input, returns 1 output
model.add(keras.layers.Dense(1,input_shape=[1]))
#how we're training it, using stochastic gradient descent to improve and mean squared
error to figure out losses
model.compile(optimizer='sgd',loss='mean_squared_error')

#data
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])
y = np.array([-2.0, 1.0, 4.0, 7.0, 10.0, 13.0])

#train the model, x as input, y as output, 500 times
model.fit(x,y,epochs=100)

#save it
model.save("model.h5")
```

8.1.2 Initial GUI Code

```
import java.awt.*;
import javax.swing.*;

public class Display {
    private int SIZE = 500;
    private ImageIcon [][] images;

    private JFrame frame;
    private JPanel panel;
    private JButton [][] chessBoardSquares;

    public Display() {
        loadImages();
        initGui();
        defaultSetUp();
    }

    public void defaultSetUp() {
        //white set up
        chessBoardSquares[0][5].setIcon(images[0][0]);
        chessBoardSquares[1][6].setIcon(images[0][0]);
        chessBoardSquares[2][5].setIcon(images[0][0]);
        chessBoardSquares[3][6].setIcon(images[0][0]);
        chessBoardSquares[4][5].setIcon(images[0][0]);
        chessBoardSquares[5][6].setIcon(images[0][0]);
        chessBoardSquares[6][5].setIcon(images[0][0]);
        chessBoardSquares[7][6].setIcon(images[0][0]);
        chessBoardSquares[0][7].setIcon(images[0][2]);
        chessBoardSquares[2][7].setIcon(images[0][3]);
        chessBoardSquares[4][7].setIcon(images[0][3]);
        chessBoardSquares[6][7].setIcon(images[0][1]);

        //black set up
        chessBoardSquares[0][1].setIcon(images[1][0]);
        chessBoardSquares[1][2].setIcon(images[1][0]);
        chessBoardSquares[2][1].setIcon(images[1][0]);
        chessBoardSquares[3][2].setIcon(images[1][0]);
        chessBoardSquares[4][1].setIcon(images[1][0]);
        chessBoardSquares[5][2].setIcon(images[1][0]);
        chessBoardSquares[6][1].setIcon(images[1][0]);
        chessBoardSquares[7][2].setIcon(images[1][0]);
        chessBoardSquares[7][0].setIcon(images[1][2]);
        chessBoardSquares[5][0].setIcon(images[1][3]);
        chessBoardSquares[3][0].setIcon(images[1][3]);
        chessBoardSquares[1][0].setIcon(images[1][1]);

    }

    public void loadImages() {
        images = new ImageIcon[2][4];

        images[0][0] = new
ImageIcon(Game.class.getResource("/resources/whitePawn.png"));
        images[0][1] = new
ImageIcon(Game.class.getResource("/resources/whiteKnight.png"));
        images[0][2] = new
ImageIcon(Game.class.getResource("/resources/whiteBishop.png"));
        images[0][3] = new
ImageIcon(Game.class.getResource("/resources/whiteRook.png"));
        images[1][0] = new
ImageIcon(Game.class.getResource("/resources/blackPawn.png"));
        images[1][1] = new
ImageIcon(Game.class.getResource("/resources/blackKnight.png"));
        images[1][2] = new
ImageIcon(Game.class.getResource("/resources/blackBishop.png"));
        images[1][3] = new
ImageIcon(Game.class.getResource("/resources/blackRook.png"));
    }
}
```



```

        images[0][3] = new
        ImageIcon(Game.class.getResource("/resources/whiteKing.png"));

        images[1][0] = new
        ImageIcon(Game.class.getResource("/resources/blackPawn.png"));
        images[1][1] = new
        ImageIcon(Game.class.getResource("/resources/blackKnight.png"));
        images[1][2] = new
        ImageIcon(Game.class.getResource("/resources/blackBishop.png"));
        images[1][3] = new
        ImageIcon(Game.class.getResource("/resources/blackKing.png"));

    }

    public void initGui() {
        frame = new JFrame("Cheskers");
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        chessBoardSquares = new JButton[8][8];

        for(int j=0 ; j<8 ; j++) {
            for(int i=0 ; i<8 ; i++) {
                chessBoardSquares[i][j] = new JButton();
                if((i+j)%2 == 0) {
                    chessBoardSquares[i]
[j].setBackground(Color.WHITE);
                }else {
                    chessBoardSquares[i]
[j].setBackground(Color.LIGHT_GRAY);
                }
            }
        }

        panel = new JPanel(new GridLayout(8, 8));
        for(int j=0 ; j<8 ; j++) {
            for(int i=0 ; i<8 ; i++) {
                panel.add(chessBoardSquares[i][j]);
            }
        }

        frame.add(panel);
        frame.setSize(SIZE, SIZE);
    }
}

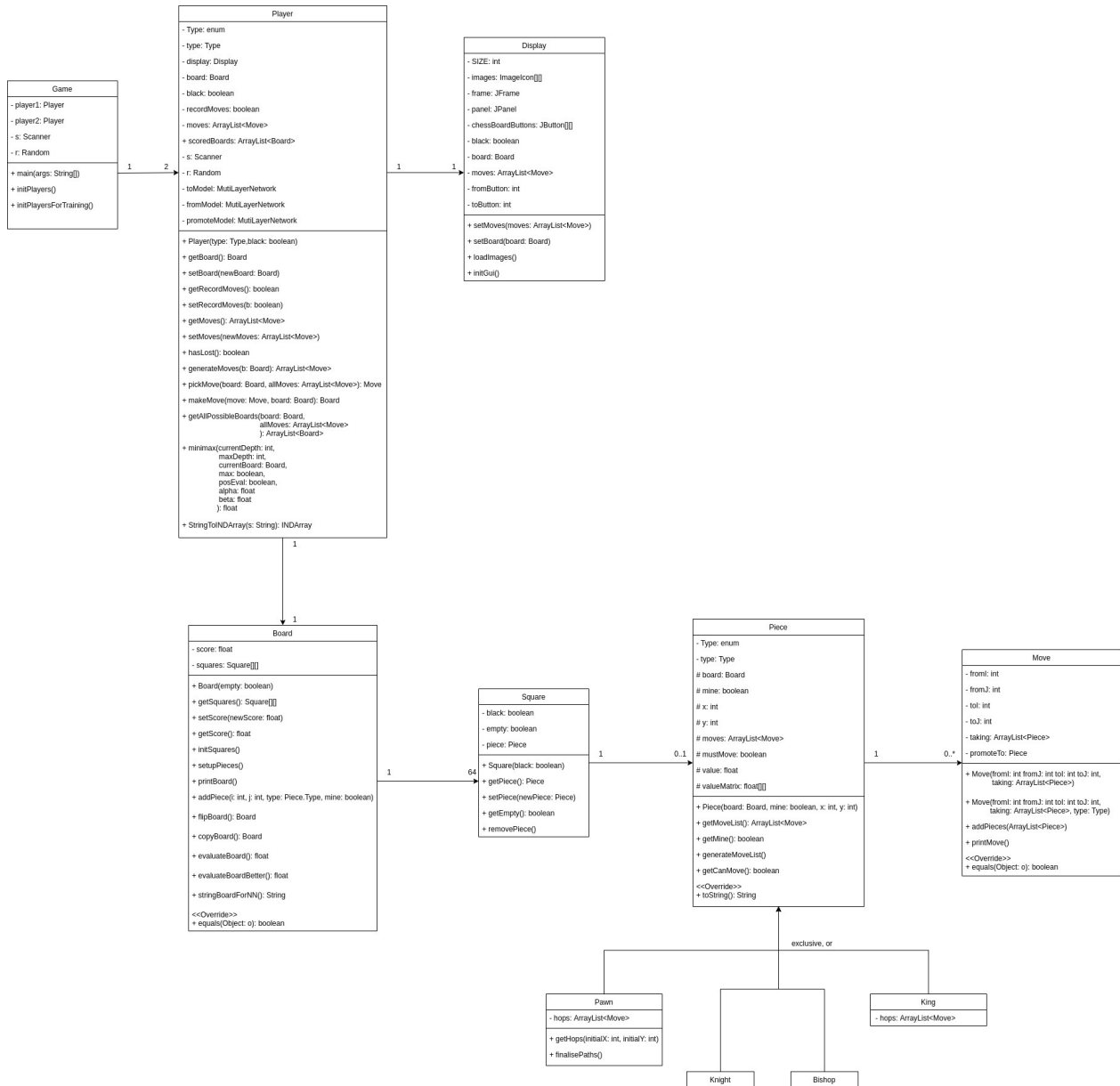
public class Game {

    public static void main(String[] args) {
        Display d = new Display();
    }
}

```

8.2 Diagrams and Tables

8.2.1 Game Class Diagram



8.2.2 Win Rate Table

mm1	100
mm2	99
mm3	99
mm4	100
mm5	100
mm6	98
nn1	37
nn2	35

8.2.3 Time Taken Table

random	0
mm1	134
mm2	130
mm3	579
mm4	1122
mm5	2976
mm6	3613
nn1	30
nn2	28

8.2.4 Time Taken With and Without Pruning Table

	without pruning	with pruning
mm1	424	134
mm3	5292	579
mm5	1174310	2976

8.2.5 Boards Evaluated

	without pruning	with pruning
mm1	6128	290
mm3	322363	5060
mm5	38958954	85530

8.3 Apache 2.0 License

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the

Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution

notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. **Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licenssor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licenssor regarding such Contributions.
6. **Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licenssor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. **Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licenssor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. **Limitation of Liability.** In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. **Accepting Warranty or Additional Liability.** While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.