**Chapter 1 Introduction**

**1.1 Introduction**

A fundamental approach to artificial intelligence has been to capture human expertise in the form of programmed rules of behavior that, when executed on a computer, emulate the behavior of the human expert. Simulating the symptoms of intelligent behavior rather than the mechanisms that underline that behavior was a natural outgrowth of the pressure to design useful software applications. This approach, however, presents significant limitation artificial intelligence programs are restricted mainly to those problems that are in large part already solved. The challenge of creating machines that can learn from their own experience without significant human intervention has remained elusive.

A landmark for artificial intelligence was achieved in 1997 when Deep Blue defeated the human world champion (Campbell,Hsu,2002) .Computer chess programs continued to progress steadily beyond human level in the following two decades. These programs evaluate positions using features handcrafted by human grandmasters and carefully tuned weights, combined with a high-performance alpha-beta search that expands a vast search tree using a large number of clever heuristics and domain-specific adaptations. In the Methods we describe these augmentations, focusing on the 2016 Top Chess Engine Championship (TCEC) world-champion Stockfish other strong chess programs, including Deep Blue, use very similar architectures (levy, Monty, 2017).

During the last two decades machine readable information has increased exponentially and with that, so has the need to understand and use this information. A technique gaining more and more interest is Machine Learning, which pro-vides tools with which large quantities of data can be automatically analyzed (Hall, 1999) and thereby easing the burden of hand-programming growing volumes of increasingly complex information (Michalski et al., 2013). Machine Learning proved to be helpful in tasks which are difficult to program by hand, such as data mining, machine translation, speech recognition, face recognition and robot motion (Pierre Lison, 1996).

The theory of deep learning provides an account of how agents may optimize their control of an environment. While deep learning agents have achieved some successes in a variety of domains, it could only be used in ends where useful features could be handcrafted (Mnih et al., 2015). In more complex tasks, features cannot be handcrafted sufficiently, as is the case of tasks that are so deeply familiar to humans that it is hard to explain their behavior explicitly, driving a car for example (H. Tsoukas, 2005), or of tasks that are not yet fully examined, example given, just developed drones that should perform ips. To use the concept of deep learning successfully, without explicitly stating features and in situations approaching real-world complexity, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs and use these to generalize past experience to new situations (Mnih et al., 2015).

With the rise of Deep Learning, agents can now successfully learn policies from high-dimensional input data. This is tested by Deep Mind in the domain of classic Atari games, where the agent, when only receiving pixel data, surpasses the level of all previous algorithms and achieved a level comparable to professional human gamers (Mnih et al., 2015). Also, the game of Go, long viewed as one of the most challenging games for artificial intelligence, has been tackled with the use of deep learning, achieving a winning-rate of 99.8% against all other Go programs and defeating the human World Go Champion Lee Sedol (Silver et al., 2016).

For the game of chess, however, the current best engines are based on brute force calculation of a limited amount of moves, in combination with a handmade evaluation function tweaked both manually and automatically over several years (Lai, 2015). Being inspired by the trend of applying deep reinforcement learning to games, which Deep Mind started by achieving superhuman performance on Atari games, a natural question arises: could an agent, which would be able to form more complex evaluation functions, improve on the current chess engines, while being independent and not restricted by any human knowledge? As an attempt of answering this question Deep Chess has been created, an engine based on Deep Learning, which has learned to play the endgame of chess, specifically, the endgame of King and Pawn against King (KPK).

**1.2 Objectives**

To build a self learning agent using Deep Learning algorithms.

To evaluate positioning of chess element for finding or predicting best move.

**1.3 Justification of Study**

The most chess engine we have today mostly based on brute force method. So the complexity of those engines high and also those chess is static. Only Deep Learning approach can make a chess engine fast and efficient.

The uses of Minimax and Alpha beta pruning together is limited in game theory

There are many approaches to build artificial chess program. Due to limit in search depth, hardware limitation minimax and alpha beta pruning with deep learning approach used in this research

Gaming agent can be used in commercial purpose.

* 1. **Scope of Study**

To make a gaming agent with deep learning algorithm a person will need to know certain facts like better search algorithm in game tree and reduce the node from game tree.

**Chapter 2 Literature Overview**

**2.1 Background and Related Work**

For more than 50 years, the game of chess has served as a testing ground for efforts in artificial intelligence, both in terms of computers playing against other computers and computers playing against humans (lasker, 2017). During these five decades, the progress of chess programs in terms of their measured performance ratings has been steady. This progress, however, has not arisen mainly because of any real improvements in anything that might be described as “artificial intelligence.”

Instead, progress has come most directly from the increase in the speed of computer hardware [(Lida, Sakuta, Jeff, 2002)](#page8) and also straight-forward software optimization. Deep Blue, the famous computer program and hardware (32 computing nodes, each with eight very-large-scale integrated processors) that defeated Kasparov in 1997, evaluated 200 million alternative positions per second. By contrast, the computer that executed *Belle*, the first program to earn the title of U.S. master in 1983, was more than 100 000 times slower. Faster computing and optimized programming allows a chess program to evaluate chessboard positions further into the prospective future. Such a program can then choose current moves that can be expected to lead to better outcomes that might not be seen by a program running on a slower computer or with inefficient programming.

**2.1.1 Deep Learning**

Deep learning is a class of machine learning algorithms that refers to the number of layers through which the data is transform. (Deng, L.; Yu, D.2014)

Most machine learning models perform well only for approximating functions that are rest or second order. They are not capable of building up hierarchies of knowledge to model very complex high order features. As a result, in order to approximate complex functions where the input-output mapping is inherently hierarchical, the models would have to be made extremely large. Extremely large models not only require high computational power, but they also require very large training sets to prevent over fitting, due to the high degrees of freedom in the model. Unfortunately, computational power and availability of training examples are usually limiting factors in practical machine learning applications.

Deep learning is used across all industries for a number of different tasks. Commercial apps that use image recognition, open source platforms with consumer recommendation apps and medical research tools that explore the possibility of reusing drugs for new ailments are a few of the examples of deep learning incorporation.

To perform the classification directly from pixels, the system would have to essentially evolve to memorize images of cars and test input images for similarity with those images. Besides the problems mentioned above, these models are also unlikely to generalize well - given only red, blue and yellow cars in the training set, it's unlikely to be able to correctly identify green cars.

The most efficient and accurate way to classify those images is with a hierarchical approach, where the rest layer would identify low level features like gradients, corners and edges. The second layer can then use the output of the rest layer to identify shapes. Finally, the location of the shapes can be used to classify objects in the image. This is what humans naturally do and it's something that machine learning systems struggled with until recently.

A new class of algorithms capable of performing hierarchical knowledge extraction was popularized in 2007 by Hinton et [al.](#page38) while there are a few models that can be extended to perform deep learning, the vast majority of current applications use artificial neural networks.

In the past few years, deep learning has refreshed the state-of-the-art results on a wide range of problems, often greatly surpassing the previous results. For example, in 2012, Krizhevsky et al. trained a deep network for the popular Image Net competition.

As another example, Google's Deep Mind recently published their results on training deep networks to play seven different arcade games, given only pixel intensities as input and increasing the score displayed as the objective. The networks they trained were able to surpass the performance of previous game-specific AIs on six of the seven games and exceeded human expert performance on three of them[.](#page38)

Besides computer vision and control tasks, deep learning has also been successfully applied to natural language processing, sentiment analysis, audio analysis and other fields where knowledge is inherently hierarchical.

In this research, we apply deep learning to chess. We use DL algorithm to evaluate positions, decide which branches to search and order moves.

**2.1.2 Deep Learning in chess**

In this section, the current state of chess computers is illustrated, followed by the problem of evaluation functions in chess. Thereafter, the achievements of chess engine are discussed, ending with the difference between Giraffe's and Deep Chess's approach

In 1997 World Chess Champion Garry Kasparov got defeated by IBM's Deep Blue. For the first time in the history of chess, computers proved to be stronger than humans. The strongest chess computers rely heavily on brute-force methods: calculating all possible combinations of moves to a certain depth (Hsu, 2002). How can a human searching 3 to 5 positions per second be as strong as a computer searching 200 million positions per second? Apparently, humans calculate much more effectively and rely on their intuition gained by experience. It is hard to de ne concrete rules to increase computer's search e effectiveness without overlooking strong moves (Lai, 2015).

A problem with most current chess engines is their evaluation functions, which assign scores to positions without calculating further. These functions contain most of the domain-specific knowledge in chess engines. The best chess engine at the moment, Stockfish, has an evaluation function which is designed with the help of many grandmasters and consists of more than 100 handcrafted features, slightly manipulated over the last few years (Lai, 2015).

As an attempt to solve the problem of evaluation functions in chess, Giraffe was engineered, a chess engine based on Deep Learning. Giraffe is the first successful attempt at using machine learning to create a chess evaluation function, with minimal hand-coded knowledge. It reached a level of International Master (Elo-rating of 2400) and achieved at least comparable positional understanding compared to the top engines of the world. This is quite remarkable, since the evaluation functions of the top engines have all been tuned both manually and automatically over several years and many of them have been worked on by human grandmasters (Lai, 2015). M. Lai states about Giraffe: Unlike most chess engines in existence today, Giraffe derives its playing strength not from being able to see very far ahead, but from being able to evaluate tricky positions accurately and understanding complicated positional concepts that are intuitive to humans, but have been elusive to chess engines for a long time."

The evaluation function is approximated using a 3-layer neural network, consisting of two hidden layers and one output layer.

**2.1.3 Conventional Chess Engines**

Although they differ in implementation, almost all chess engines which exist today (and all of the top contenders) implement largely the same algorithms. They are all based on the idea of the depth minimax algorithm first developed by John von Neumann in 1928 and adapted for the problem of chess by Claude E. Shannon in 1950[.](#page37)

**2.1.4 Minimax Algorithms**

The minimax algorithm is a simple recursive algorithm to score a position based on the assumption that the opponent thinks like we do and also wants to win the game.

In its simplest form -

**function minimax(node, depth, maximizingPlayer) is**

**if depth = 0 or node is a terminal node then**

**return the heuristic value of node**

**if maximizingPlayer then**

**value := −∞**

**for each child of node do**

**value := max(value, minimax(child, depth − 1, FALSE))**

**return value**

**else (\* minimizing player \*)**

**value := +∞**

**for each child of node do**

**value := min(value, minimax(child, depth − 1, TRUE))**

**return value**

Listing: Simple minimax

This algorithm works in theory and also in practice for simpler games like tic-tac-toe (game tree size of at most 9! or 362880). However, as it searches the entire game tree, it is impractical for games like chess, where the average branching factor is approximately 35, with an average game length of about 80 plies. The search tree size for chess is estimated to be about 1023-1046 without repetitions which is more than one hundred orders of magnitudes higher than what is computationally feasible using modern computers. Therefore, a chess engine must decide which parts of the game tree to explore (Knudsen, 2000).

The most common approach is a depth search, where it artificially limit how far ahead it will look and when it is at the end of the sub tree it want to search, it call a static evaluation function that assigns a score to the position by analyzing it statically (without looking ahead). A very simple evaluation function is to simply add up pieces of both sides, each multiplied by a constant (e.g. Q = 900, R = 500, B = 300, N = 300, P = 100). The evaluation function is very important and is one of the major areas of investigation for this research.

**Function minimax (position,depth)**

**{**

**if position is won for the moving side :**

**return 1**

**else if position is won for the non – moving side :**

**return - 1**

**else if position is drawn :**

**return 0**

**if depth == 0:**

**return evaluate(position)**

**best Score = -1**

**for each possible move mv :**

**subScore = - minimax(position.apply(mv),depth - 1)**

**if subScore > bestScore :**

**bestScore = subScore**

**return bestScore**

**}**

Listing: Depth-limited minimax

There are 3 changes –

* minimax() gets an additional parameter, depth, representing how deep we want to search this position
* if it is at depth 0, call evaluate and return the result
* if it is at depth > 0, recursively call minimax as before and pass in depth 1 as the depth for the sub-tree.

The search will now terminate in reasonable time, given a reasonable depth limit, but there is a very serious problem - the horizon effect.

In the algorithm above, we cut off all branches at the same distance from the root (called the horizon). This is dangerous because, for example, if in one branch, the last move happens to be Q\*P (queen captures pawn), we may score that position as a pretty good position, since we just won a pawn. However, what we don't see is that the pawn is defended by another pawn and the opponent will take our queen the next move. This is actually a very bad position to be in.

This problem cannot be solved by simply increasing the depth limit, because no matter how deep it is searching, there will always be a horizon. The solution is quiescent search. The idea is that once we get to depth == 0, instead of calling evaluate () and returning, we enter a special search mode that only expands certain types of moves and only call evaluate () when we get to a "quiet" and relatively stable position.

There is a trade to be made here - if we include too many moves, q-searches become too large and we won't be able to search many plies in normal search. If we include too few moves, we may suffer from reduced forms of the horizon effect.

Q-searches are usually not depth-limited and instead rely on the tree terminating. Trees will always terminate (usually reasonably quickly) since the number of possible captures is usually limited and tend to decrease as captures are made.

**2.1.5 Pruning**

Without introducing any heuristics and with no loss of information, this algorithm can be optimized by introducing a "window" for each call to minimax (). The idea is that if the true score is below the lower bound that means the caller already has a better move and therefore doesn't care about the exact value of this node (only that it is lower than the lower bound). Conversely, if the true score is higher than the upper bound that means the caller also doesn't care about the exact value, just the fact that it is higher than the upper bound. It may seem counter-intuitive at rest to have an upper bound but the reason for that is because chess is zero-sum and upper bound and lower bound switch places for each ply as we search deeper.

This optimization is called - pruning[,](#page37) where and are the lower and upper bounds. Since and are horrible choices of variable names, we will refer to them as lower bound and upper bound in the rest of this paper but still refer to the algorithm as - for consistency with existing literature.

**function alphabeta(node, depth, α, β, maximizingPlayer) is**

**if depth = 0 then return the value of node**

**if maximizingPlayer then**

**value := −∞**

**for each child of node do**

**value := max(value, alphabeta(child, depth − 1, α, β, FALSE))**

**α := max(α, value)**

**if α ≥ β then**

**break (\* β cut-off \*)**

**return value**

**else**

**value := +∞**

**for each child of node do**

**value := min(value, alphabeta(child, depth − 1, α, β, TRUE))**

**β := min(β, value)**

**if α ≥ β then**

**break (\* α cut-off \*)**

**return value**

Listing: Alpha Beta Pruning

A detailed analysis of - is omitted here for brevity, but there is one very significant implication - that the order moves are searched is extremely important.

In standard minimax, move ordering (the order in which we explore nodes) is irrelevant because all nodes have to be visited exactly once. With - pruning, we only explore nodes that can potentially be useful and we can stop searching a node as soon as we prove that the result will be outside the window. That means, if we always expand the best nodes rest, we will not have to examine as many nodes as if we had sub-optimal ordering.

It is obviously impossible to always have optimal ordering - since if there is a way to generate an optimal ordering for any given position, there would be no need to search at all. However, by ensuring that move ordering is close to optimal in most cases using heuristics, we can make the search more efficient.

In conventional chess engines, there are a few common heuristics to improve move ordering -

* If we had previously searched the same position, the move that ended up being the best is probably still the best (even if the previous search was done to a shallower depth).
* If a move proved to be good in a sibling node, there is a good chance it will be good for the node under consideration as well.
* Capturing high-valued pieces with low-valued pieces is usually good and capturing defended low-valued pieces with high-valued pieces is usually bad.
* Queen promotions are usually good.

In this research, we will investigate using machine learning to assist the engine in ordering moves.

Depth-limited minimax () with pruning and q-search form the backbone of virtually all existing chess engines. There are many further optimizations that can be done and have been done to further improve the search. Those optimizations will not be covered in this paper because they are not directly relevant for this research. However, many of them have been implemented in the chess engine created for this research.

**2.1.6 Evaluation**

The evaluation function is a very important part of a chess engine and almost all improvements in playing strength among the top engines nowadays come from improvements in their respective evaluation functions. The job of the evaluation functions is to assign scores to positions statically (without looking ahead). Evaluation functions contain most of the domain-specific knowledge designed into chess engines.

In this research, we will develop an evaluation function based on a machine learning approach inspired from Stockfish[,](#page37) an open source chess engine that is currently the strongest chess engine in the world. Examining an existing state-of-art evaluation function can help us design an effective feature representation for our machine learning effort.

Evaluation function consists of 9 parts -

**Material:** Each piece on the board gets a score for its existence. Synergetic effects are also taken into account (for example, having both bishops gives a bonus higher than two times the material value of a bishop) and polynomial regression is used to model more complex interactions.

**Piece-Square Tables:** Each piece gets a bonus or penalty depending on where they are, independent of where other pieces are. This evaluation term encourages the engine to advance pawns and develop knights and bishops for example.

**Pawn Structure:** The position is scanned for passed, isolated, opposed, backward, unsupported and lever pawns. Bonuses and penalties are assigned to each feature. These features are all local, involving 2-3 adjacent pawns.

**Piece-specific Evaluation:** Each piece is evaluated individually, using piece-type-specific features. For example, bishops and knights get a bonus for being in a "pawn outpost", rooks get a bonus for being on an open le, semi-open le, or the same rank as enemy pawns.

**Mobility:** Pieces get bonuses for how many possible moves they have. In Stockfish's implementation, squares controlled by opponent pieces of lesser value are not counted. For example, for bishop mobility, squares controlled by enemy pawns are not included and for queen mobility, squares controlled by bishops, knights, or rooks are not included. Each piece type has a different set of mobility bonuses.

**King Safety:** Bonuses and penalties are given depending on number and proximity of attackers, completeness of pawn shelter and castling rights.

**Threat:** Undefended pieces are given penalties; defended pieces are given bonuses depending on piece type and defender type.

**Space:** Bonuses are given for having "safe" empty squares on a player's side.

**Draw-ish-ness:** Certain material combinations often result in draws, so in these cases the evaluation is scaled toward 0.

It is quite a complicated function with a lot of hand-coded knowledge. Most engines don't have evaluation functions that are nearly as extensive, because it is difficult to tune such a high number of parameters by hand.

**2.2 Related Work**

In the eighteen century, the idea of creating a chess playing machine arose with The Turk, a fake chess automaton that became famous before it was exposed as a fake. It was basically a mechanical illusion created by hiding a person with high chess knowledge inside the machine who would operate it. Only in 1912 did a real automaton appeared under the name of “El Ajedrecista”. It used electromagnets under the board and automatically played a three chess piece endgame by moving a King and a Rook, against the human opponent (Rune Djurhuus.2017). In the late 1940s, the field of mechanical chess research faded as first computers appeared. Since then, computerized chess became an important research subject: if a computer could play chess, then other problems that require human intelligence could also be solved. In less than fifty years, chess programming evolved to the point that a chess engine (Deep Blue) could beat the best human player in the world at that time (Kasparov) (Hsu.2002).

Most of the nowadays chess applications use a chess engine as the “thinking” part of the program and a Graphical User Interface (GUI) as its interface. One of the most popular and used GUIs is Tim Mann’s freeware Win board. This interface allows users to visualize and interact with a virtual chessboard by using a mouse. Still there are some problems with this kind of interfaces. Namely, to operate them we need pull away from the original experience in which we face a real chessboard and can change the state of the game simply by moving a piece on the board. To address this limitation some research researches have used Tangible User Interfaces (TUI) in its design. TUIs “are a growing space of interfaces in which physical objects play a central role as both physical representations and controls for digital information”. A broad discussion of TUI has been provided by Ulmer and Ishii.

The Chess interface to computerized chess is an example of a chess research project that uses a TUI. It consists in a robotic arm with pincers that is used to move the pieces around the chess board. It can be used to play against a computer or a geographically remote opponent. Chess has a robotic arm and thus has the advantage of making the computer’s move, but has the disadvantages of doing it very slowly, being expensive and invading a lot of space. A low cost homemade solution, the Chess Box system is built by using a wooden box to contain a webcam, a mirror and two inexpensive glass chess sets. The mirror is used to reflect the image of the chessboard back to the camera which tracks the pieces. However this system suffers from vision system’s limitations such as poor light conditions and moves are not always recognized.

**CHAPTER 3 METHODOLOGY**

**3.1 Methodology**

In this research, we used five phases to explain the procedure. These phases are planning, requirement analysis, proposed algorithm, design & development, testing and results.

**Planning**

**Requirement Analysis**

**Design & Development**

**Testing & Result**

**Proposed Algorithm**

Figure 3.1: Flowchart of Methodology

**3.1.1 Planning**

Planning is the most important part beyond any successful work because there is a guideline exists for each and every phases of planning. During reading journals, it has been found that there are so much algorithms to play chess game smoothly. Therefore, the plan was to modify existing algorithm and supporting techniques to make system more accurate.

**3.1.2 Requirement Analysis**

There are two types of requirement for our research work–

**Software Requirement:**

Language: Python

Environment: pyCharm 1.4

Operating System: Windows or MAC or Ubuntu

External Algorithm: Minimax, Alphabeta pruning, evaluation function

**Hardware Requirements:**

Laptop or Desktop with high configuration

**3.1.3 Proposed Algorithms**

Algorithm starts with taking input of chess board. Then, it will generate all legal moves and will evaluate every move with the help of evaluation function. After generating all moves it will apply every single move in chess board and will gain feedback from reward function. Final move will be based on the best score it gained from reward function.

**3.1.4 Design & Development**

After planning proposed model of Deep Chess, the some paper pencil works has been done and figured out the complete process of search and pruning technique. The design of flowchart, system design also done carefully. Then the process mathematically proven. With the success of theoretical justification, the developing the model with the programming languages started.

**3.1.5 Testing & Result**

After the development of the required algorithm, Deep Chess was tested with some fixed data. They also compared the following algorithm with the other DL algorithm like Monte Carlo with that fixed data. Finally, the algorithm put expecting result. As well as, it provides best moves against opponent

**3.2 Justification of Methodology**

As there are lots of methodologies to do research work, this methodology was chosen for flexibility. As this is a software (algorithm) design and development related work, there are only a few phases of work. Therefore, a customized and simple methodology used to perform the work.

**3.3 System Design:**

Start

Start playing

Training process

Learn game rules

**Repeat for**

**Error**

**Free results**

Figure 3.2: system design of chess engine

**3.4 Description of system**

After starting Deep Chess program it will start learning game rules any special move from preset data. Then it will start training process by playing itself. The training process is complex and time consuming. Normally it takes 4-6 hours to train itself. Training process starts with generally all legal moves for current situation. Then it will chess a best move with the help of minimax, alphabet pruning and board evaluation function. After getting best move it will apply the move and will get reward (10 for winning, 0 for lost, 1for any move). The training process will continue up to end of game. Thus it will gain it own brain to play chess without human interrupt. After training process Deep Chess is ready to play with random user.

**3.5 Model of a Traditional chess engine**

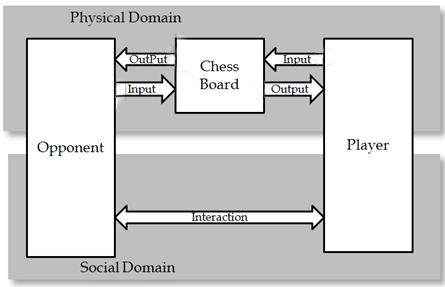
****

Figure 3.3:Design model for a tradition chess game

Playing chess in its original form with a real chessboard and against a real opponent share the social advantages, physical controls and information offered by most of board games. The design model for a regular chess game between two players. The human-to-human interaction present in a traditional chess game has elements from the physical and social domains as integral parts of the gaming experience. An example of social interaction in a chess game happens when we try to understand our opponent’s mind by, for instance, detecting that false signal he just made that expresses a terrible mistake with the intention of bluffing. Both players are in the physical domain where they interact with a physical chessboard. Since the chessboard is shared by both players, when a player inputs a new move that move becomes the output of his adversary.

**3.6 Model for a single player computerized chess game**

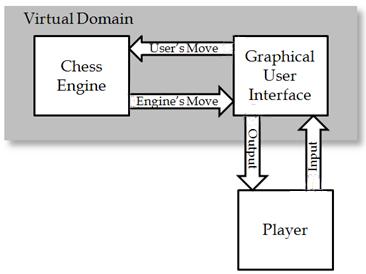
****

Figure 3.4:Design model for most of single player computerized chess games

The human player belongs to the physical domain. However, since there is no interaction in that domain in this particular model, we chose not to represent it. The virtual domain, on the other hand, is where all the interaction takes place.

The term opponent does not appear in this model because even though there is a chess engine that thinks about the best move to make, the embodiment of such opponent is usually not physically or virtually represented.

The lack of social interaction between opposing players is one of the main disadvantages of computerized chess. On the other hand, the digital version of chess allows us to play at any time with an opponent of custom difficulty. Also, computerized chess allows new features that were only possible by consulting chess instructors or chess books such as move advices or consult database of famous games.

**3.7 Deep Chess model**

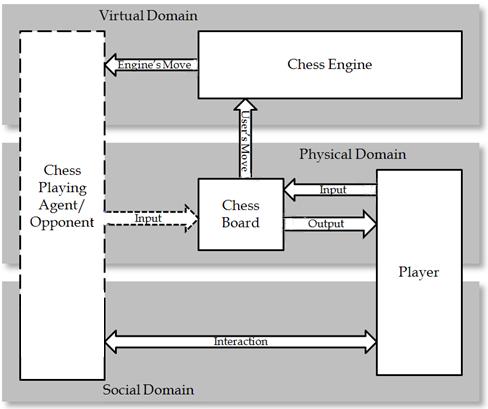
****

Figure 3.5: Deep Chess model

Our model uses a social agent that interacts with the user of a computer augmented chess game. An agent is an artificial software system situated in a computerized environment, which senses that environment and acts on it, over time, achieving its goals (Beal, Martin, 2001). The idea is to simulate the social interaction between two human players by having a human-agent relationship. For instance, while waiting for the opponent’s move, the player can try to interpret the agent’s mind, or even look at its defeated expression when it loses. The chess playing agent/opponent appears in a dashed line across the virtual and physical domains because it can be physically (e.g., a robot) or virtually embodied (e.g., a graphical avatar).

When comparing the virtual domain of this model to the one of computerized chess, it loses the GUI to display the virtual chessboard since now we have a chess TUI in the physical domain to replace it. While using a tangible user interface, we can input the user’s move into the virtual domain simply by moving a piece in the chessboard. The agent may have the capability of moving its pieces with, for example, a mechanical arm. If so, he plays the move, otherwise the agent must ask the player to make its move.

**Chapter 4 Project Description and Methods**

Each chessboard position was represented by a vector of length 64, with each component in the vector corresponding to an available position on the board. Components in the vector could take on values from, where 0 represented an empty square and the variables,,,, and  represented material values for pawns, knights, bishops, rooks and the queen and king respectively. The chess engine assigned a material value to kings even though the king cannot actually be captured during a match. The sign of the value indicated whether or not the piece in question belonged to the player (positive) or the opponent (negative). A player’s move was determined by evaluating the presumed quality of potential future positions.

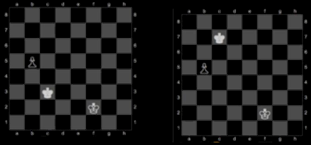
**4.1King Pawn King Positions**

Since the whole game of chess is very complex, the endgame has been chosen as a start to apply Deep Reinforcement Learning to chess. The endgame of chess is reached, when most of the pieces are taken. Specifically, the endgame of King and Pawn versus King (KPK) has been chosen to train Deep Chess on. In this subsection, the possible positions occurring in this specific endgame are sketched, together with the theory of won and drawn positions.

KPK positions are the situations where all pieces are taken, except for one pawn, resulting in positions with two kings and one side having a pawn.

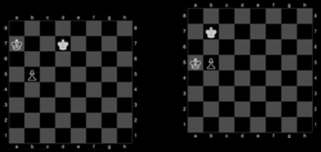
From white’s point of view, KPK positions could be both theoretically won or drawn. Since a position with only two kings is drawn, these positions can never lead to a loss. The strategy for the winning side is to promote the pawn; in other words, to move the pawn up to the 8th rank where it may promote to a queen. It could be said that whenever white is able to promote the pawn, white wins. Winning a game with an extra queen is basic and therefore, is left out of this research.

The borderline between a theoretically won and a drawn position can be quite narrow. However, these positions can be classified in three difficulty degrees, which have different winning strategies. Firstly, when the pawn is out of reach for the black king, the strategy is to move the pawn up the board before the black king can catch up. Secondly, when the black king is able to take the pawn, the white king should support the pawn in its way to promotion. Thirdly, when the black king is nearly able to keep the white king from supporting its pawn to promotion, the white king should shield away the black king to enter the second situation. For training Deep Chess, a random position generator has been created, which places all three pieces at a random location on the board.



(a) This position is won, This position is drawn,  
the black will not be able the black king will take   
to catch up with the pawn. the pawn in few moves.

(the pawn is out of reach)



(b) This position is won; This position is drawn,  
 since the white king since the black king is able   
 supports its pawn to keep the white king from

pawn up its way moving into a position

to promotion

Figure 4.1: The won and drawn position of king.

**4.2 State Representation**

In this section, the state representation is discussed such that it can be used to apply Deep Learning methods on. It addresses the Markov state, multi-channels and zero-averaging.

For Deep Learning methods to apply, Markov States should be created. A Markov State is a state that is sufficient to formulate a winning strategy only based on the information captured by that state. Chess is a perfect in-formation game, which means no information is hidden to both players. This means that solely the current chess position is enough to tell the best move, resulting in every possible chess position being a Markov state.

To represent a chess position of KPK such that it can be fed to the Q-network, an 8 x 8 x 3 matrix has been created with numbers corresponding to the location of the pieces. Every piece has its own channel within the representation to avoid stating any implicit ordering between the different pieces, Also, the average of each channel has been made zero, since for some non-linearity’s, such as a sigmoid function, the gradient is on its steepest around zero and therefore, the loss function would be optimized faster and would avoid saturated neurons.



Figure 4.2: Stalemate position (4 x 4)

|  |  |  |  |
| --- | --- | --- | --- |
| [15/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16] |
| [15/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16] |
| [15/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16 |
| -1/16 | -1/16 | -1/16 | -1/16] |

Table 4.1: The state representation 4 x 4

The agent would also have to get the number of moves left as an input, would the agent be able to take the 50-move rule into account. Therefore, the 50-move rule is discarded while training, meaning that the game goes on until another terminal state is reached. While training, the game cannot result in a repetition of moves, since the agent will always have a chance of 0.1 of performing a random move. This ensures that the game knishes in a niter number of moves, a requirement of a Markov Decision process.

**4.3 Reward Function**

In this section, the rewards of the agent are discussed, together with the handling of illegal moves.

For the agent to learn while playing games, it is necessary that it receives feedback from the environment, which is given in the form of rewards. The agent learns to predict the cumulative future reward for each possible action in each state, by minimizing the loss function discussed in Deep Learning, For reaching a winning state it gets a positive reward, for reaching a drawn state it gets a negative reward (since in white's point of view this is actually losing). If the agent would play the black side too, drawing could be given zero rewards; however, this is outside of the achievements of this research. To encourage Deep Chess to find a quick solution, performing a move is also given a small negative reward

|  |  |
| --- | --- |
| Win | +10 |
| Draw | -10 |
| Illegal/lost | -10 |
| Any | -1 |

Table 4.2: Reward function

To restrict Deep Chess from performing illegal moves, the agent is given the same negative reward as drawing when its outputted move is an illegal one. This way the agent learns to avoid illegal moves. While testing, however, the agent should do a move (if not in stalemate) even when the outputted move is illegal. In this case, the agent performs a random move instead. However, this should not happen if the agent has sufficiently learned.

**4.4 Evaluation Methods**

In this section the evaluation of Deep Chess is discussed, together with the theoretical best results achievable and ending with the division of positions into different levels.

In section King Pawn King, we have seen that KPK endgames can result in either a win or a draw. Therefore, it is necessary to separate Deep Chess’s performance in winning and drawing positions, since Deep Chess is only expected to win in a winning position and is not expected to win in a drawn position. Therefore, as an evaluation method, the number of wins in winning positions and the number of wins in drawing positions have been tracked separately.

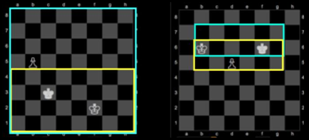
To know which random starting position theoretically results in a draw or a win, a connection with table base Gaviota was created. Gaviota is a table base including all possible chess endgames with less than 6 pieces, for which each specific endgame's theoretical result is stated.

The theoretical best results Deep Chess could achieve playing against a perfect opponent would be to win all winning positions and draw all drawn positions. Playing against Sunfish, however,

Deep Chess could also strive to win in drawn games, since Sunfish does not play perfectly, especially when the maximum number of searchable nodes is low.

Another intuitive evaluation method is to track the number of moves it takes Deep Chess to win. However, in different starting positions Deep Chess is aiming for different optimal lengths. Therefore, the amount of moves to win should be tracked per level of starting positions.

Since the difference between the difficulties of starting positions is quite large in KPK endgames, its test environment has been set up with 3 different levels of starting positions: easy, intermediate and hard positions. In easy positions, the white pawn is out of reach for the black king, which means it is only required to move the pawn up the board to win, see the theory of winning such a position in section King Pawn King Positions. In intermediate positions, the white king is in front of its pawn and the black king next to it. Therefore, it is required to support the pawn with the king. In hard positions, both the white king and the black king are in front of the pawn, which requires knowledge of keeping the black king from controlling the promotion square. In Figure, boxes are drawn that state where the kings can be randomly placed according to the classification of the different starting positions.



Easy starting position Intermediate starting positions



Figure 4.3: Hard starting positions

As justification for this classification, the number of theoretically won starting positions has been compared to the amount of theoretically drawn starting positions, for each of the different levels. In the easy positions 91 % of the positions are won, in the intermediate positions 77% of the positions are won and in the hard positions 53 % of the positions are won, whenever a winning position is more similar to a drawn position, its winning strategy requires more subtle movements and therefore increases in difficulty. Therefore, the ratio between won and drawn positions serves as a justification for the classification of different levels of starting positions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | All | Easy | medium | Hard |
| End game | 73% | 91% | 77% | 53% |
|  |  |  |  |  |

Table 4.3: Winning position rates in the different levels.

To track the performance during training, the following numbers are plotted: the average loss per move, the highest Q value per state and the amount of wins in winning positions per winning position.

**Chapter 5 Analysis of Results**

**5.1 Results**

We divided the data in 2 sections - 1) Starting move 2) End game move.

* **Starting move:** There are many starting move in chess. Like king Indian defense, two knights opening etc. Mainly that especial move is one kind of trap. So once must need to know about them. Likely other chess engine Deep chess also select e4 as best move and the second best move is n3.
* **End game move:** The most important part of game is end game. For this we considered some technique like pawn promotion, not letting opponent to promote his pawn, King safety etc.

|  |  |
| --- | --- |
| Move type | Results |
| Starting move | 81% |
| End game move | 73% |

Table 5.1: Results for different move type

The results based on specific situation. This may vary for different position.

To test the system we run the program through a test data. For positional test results we divided our possible outcomes in three formats-

**Best move:** These results indicate that the move is perfect for current chess position.

**Blunders:** These results means your move may not perfect but it’s not a mistake or worst move. There should have better move than that.

**Inaccurate:** Not good move.

After testing the program irretrievably for 1500 different chess position we found the below result

|  |  |
| --- | --- |
| Position number | Result |
| Best | 771 |
| Blunders | 328 |
| Inaccurate | 401 |

Table 5.2: Results for different chess position

The data which was tested downloaded from internet .We do not claim the 100% accuracy of tested data.

**5.2 Playing with Sunfish**

After testing Deep Chess in different chess position it was mandatory to test Deep Chess with standard

opponent like another chess engine. For this we used Sunfish (an open source chess engine).

Playing with Sunfish for Deep Chess was hard because Deep Chess wasn’t that much ready or fast like Sunfish was.

Here is the 10 game results played between Deep Chess and Sunfish.

**Game 1**

1.e4 e6 2.d4 d5 3.Nd2 c5 4.exd5 Qxd5 5.dxc5 Nf6 6.Ngf3 Qxc5 7.Bd3 Qc7 8. O-O Nbd7 9.a4 Nc5 10.Bc4 Be7 11.a5 O-O 12.Qe2 b6 13.a6 Bd7 14.Nd4 Rac8 15. Nb5 Qb8 16.b3 Bxb5 17.Bxb5 Nd5 18.Bb2 Bf6 19.Bxf6 Nxf6 20.b4 Ncd7 21.Nf3 Qc7 22.c4 Nb8 23.Rfd1 Qe7 24.Rab1 Ne8 25.Bxe8 Rfxe8 26.b5 Qc7 27.Rbc1 Red8 28.g3 h6 29.Ne5 Nd7 30.Nc6 Re8 31.Rd2 Nc5 32.Rcd1 e5 33.Qg4 Re6 34.Rd8+ Rxd8 35.Rxd8+ Kh7 36.Ra8 Rf6 37.Rc8 Qd6 38.Rd8 Qc7 39.Qc8

Deep Chess won

**Game 2**

1. e4 e6 2.d4 d5 3.Nd2 Nf6 4.e5 Nfd7 5.c3 c5 6.f4 Nc6 7.Ndf3 Qb6 8.a3 Be7 9.h4 a6 10.h5 cxd4 11.cxd4 Qc7 12.Bd3 b5 13.Ne2 f5 14.Kf2 Nb6 15.Qg1 Bd7 16.Bd2 O-O-O 17.b3 Kb7 18.Qe1 Ka7 19.Nc3 b4 20.axb4 Nxb4 21.Qe2 Qb7 22.Nb5+ Bxb5 23.Bxb5 Ra8 24.Bxb4 Bxb4 25.Bxa6 Qe7 26.Qb5 Ba3 27.Ra2 Qb4 28.

Rha1

Deep Chess won

**Game 3**

1.e4 e6 2.d4 d5 3.Nd2 h6 4.Ngf3 Nf6 5.Bd3 c5 6.exd5 exd5 7.Bb5+ Bd7 8.Qe2+Be7 9.dxc5 O-O 10.O-O Re8 11.Qd3 Bxc5 12.Nb3 Bb6 13.Nbd4 Nc6 14.c3 a6 15.Nxc6 bxc6 16.Ba4 c5 17.Bxd7 Qxd7 18.c4 Qg4 19.h3 dxc4 20.Qd6 Qe6 21.Qxe6 Rxe6 22.Nd2 c3 23.bxc3 Nd5 24.c4 Nf4 25.Nb1 Ne2+ 26.Kh1 Rd8 27.Bd2 Rd4 28.Re1 Rxc4 29.Be3 Nd4 30.Nd2 Rc3 31.Rac1 Rxc1 32.Rxc1 Ba7 33.Nb3 Re5 34.Rc4 Ne6 35.Ra4 Rd5 36.Rxa6 Bb8 37.g3 Be5 38.Ra5 Rd1+ 39.Kg2 Bd4

Match Draw.

**Game 4**

1. e4 e6 2.d4 d5 3.Nd2 Nf6 4.e5 Nfd7 5.Bd3 c5 6.c3 Nc6 7.Ne2 cxd4 8.cxd4 f69.exf6 Nxf6 10.O-O Bd6 11.Nf3 O-O 12.Bg5 Qb6 13.Nc3 Bd7 14.a3 h6 15.Bh4 Kh8 16.Bc2 g5 17.Bg3 Ne7 18.Bxd6 Qxd6 19.Ne5 Be8 20.f4 Bh5 21.Qd2 g4 22.f5Kg7 23.fxe6 Qxe6 24.Qd3 Ne4 25.Nxe4 dxe4 26.Qxe4 Bg6 27.Nxg6 Qxe4 28.Bxe4 Nxg6 29.Rac1 Rxf1+ 30.Kxf1 Rf8+ 31.Ke2 Rf7 32.Ke3 Rd7 33.Rc8 Ne7 34.Rb8 b635.Rb7 Nf5+ 36.Kf4 Rxb7 37.Bxb7 Nxd4 38.Kxg4 Nb5 39.Kf4 Kf6 40.Bd5 Nc7 41.

Ba2 a5 42.g4 Na6 43.h4 Nc5 44.g5+ Kg7 45.Bb1 hxg5+ 46.hxg5 b5 47.Ke5 b4 48.Bc2 Nd7+ 49.Kd6 bxa3 50.bxa3 Nf8 51.a4 Kf7 52.Bf5 Ng6 53.Bxg6+ Kxg6 54.Kc5

Deep Chess won.

**Game 5**

1. e4 e6 2.d4 d5 3.Nc3 Nf6 4.Bg5 Bb4 5.e5 h6 6.Bc1 Ne4 7.Qg4 g6 8.Ne2 c5 9.a3 Bxc3+ 10.Nxc3 Nxc3 11.bxc3 Qa5 12.Bd2 Qa4 13.Qf3 cxd4 14.Bd3 dxc3 15.Bxc3 b6 16.Bb4 Nc6 17.Qf6 Rg8 18.Bd6 Qa5+ 19.Ke2 Ba6 20.Rhc1 Qa4 21.Kf1 Rd8 22.Kg1 Bc4 23.Rab1 Qa6 24.Rb4 Qxa3 25.Rcb1 Qc3 26.Bxc4 dxc4 27.Qh4 Qxc2 28.R4b2 Qd3 29.h3 g5 30.Qg4 h5 31.Qxh5 g4 32.Re1 c3 33.Ra2 gxh3 34.g3Qd5

Sunfish won.

**Game 6**

1. e4 e6 2.d4 d5 3.Nd2 Nf6 4.e5 Nfd7 5.Bd3 c5 6.c3 Nc6 7.Ne2 cxd4 8.cxd4 f69.exf6 Nxf6 10.Nf3 Bd6 11.O-O Qc7 12.Nc3 a6 13.Bd2 O-O 14.Rc1 Bd7 15.Re1 Ng4 16.Bxh7+ Kh8 17.g3 Be8 18.Rxe6 Bh5 19.Nxd5 Qf7 20.Rxd6 Nxf2 21.Kxf2 Bxf3 22.Bg6 Qg8 23.Qxf3 Rxf3+ 24.Kxf3 Nxd4+ 25.Kg2 Qf8 26.Bf4 Ne2 27.Rd1 Rd8 28.Rxd8 Qxd8 29.Be3 Qd7 30.Bh5 Qb5 31.Bf3 Kg8 32.Kf2 Nxg3 33.hxg3 Qxb2+ 34.Rd2 Qa3 35.Nf4

Deep Chess won.

**Game 7**

1.e4 e6 2.d4 d5 3.Nd2 c5 4.Ngf3 Nc6 5.exd5 exd5 6.Bb5 Qe7+ 7.Be2 Qc7 8.O-ONf6 9.Re1 Be6 10.c3 cxd4 11.Nxd4 Nxd4 12.cxd4 Bd6 13.Bb5+ Nd7 14.Nf3 O-O 15.h3 Nf6 16.Bd3 Qb6 17.b3 Rae8 18.Bg5 Nd7 19.Bc2 h6 20.Qd3 f5 21.Bd2 Nf6 22.Ne5 Ne4 23.Bf4 g5 24.Bh2 f4 25.Nc4 Qc6 26.Nxd6 Qxd6 27.f3 Bf5 28.fxe4 dxe4 29.Qc4+ Be6 30.Qc5 Qxc5 31.dxc5 Bd5 32.Rad1 Bc6 33.b4 a6 34.a4 e3 35.Rd6 Rd8 36.Red1 Rxd6 37.cxd6 Rd8 38.Kf1 Kf7 39.Bg1 Ke6 40.Bb3+ Ke5 41.g3 Ke4 42.gxf4 gxf4 43.Bc2+ Kf3 44.Rd4 Kg3 45.Bf5 Bxa4 46.d7 Bb5+ 47.Ke1 Kg2 48.Bxe3 fxe3 49.Be6 Kg3 50.h4 b6 51.Re4 Bxd7 52.Rd4 Bxe6 53.Rxd8 a5 54.bxa5 bxa5 55.Ra8 Kxh4 56.Ke2 Kg4 57.Kxe3 h5 58.Rxa5 h4 59.Ra4+ Kg5 60.Kf3 h3 61.Kg3 Kf6

Match draw.

**Game 8**

1. e4 e6 2.d4 d5 3.Nc3 Bb4 4.e5 Ne7 5.Nf3 c5 6.dxc5 Bxc5 7.Bd3 Nec6 8.O-O Nd7 9.Bf4 f6 10.exf6 Qxf6 11.Bg5 Qf7 12.Be3 Be7 13.Ng5 Bxg5 14.Bxg5 O-O 15.f4 Nc5 16.Be2 Qc7 17.f5 Qb6 18.Kh1 d4 19.b4 dxc3 20.bxc5 Qxc5 21.Rf4 exf5 22.Bc4+ Kh8 23.Qh5 Ne5 24.Rh4 h6 25.Bxh6 Nxc4 26.Be3+

Deep chess won

**Game 9**

1.e4 e6 2.d4 d5 3.Nc3 Nf6 4.e5 Nfd7 5.f4 c5 6.Nf3 Nc6 7.Be3 Qb6 8.Na4 Qa5+9.c3 b6 10.a3 c4 11.b4 cxb3 12.Qxb3 Ba6 13.Nd2 Bxf1 14.Rxf1 Qa6 15.f5 Be7 16.fxe6 fxe6 17.Nb2 Na5 18.Qa4 b5 19.Qc2 Nc4 20.Nbxc4 bxc4 21.Ke2 Nb6 22.a4 Qa5 23.g3 Kd7 24.Rf7 g6 25.Bg5 Rae8 26.Qb2 Kd8 27.Rxe7 Rxe7 28.Qa3 Nc8 29.Rb1 Qc7 30.a5 Ke8 31.Bxe7 Qxe7 32.Qxe7+ Kxe7 33.Rb7+ Kd8 34.Nf3 h6 35.Rg7 g5 36.Rb7 Ke8 37.Ne1 Rg8 38.Nc2 h5 39.Nb4 h4 40.g4 h3 41.Na6 Kd8 42.Nc5 Rg6 43.Nd7 Rg7 44.Nc5 Rxb7 45.Nxb7+ Ke7 46.Kf3 Kf7 47.Kg3 Ke8 48.Kxh3 Ne7 49.Kg3 Ng6 50.Nd6+ Kd7 51.Nf7 Nf4 52.Kf3 Nh3 53.Kg3 Nf4 54.Nxg5 Ne2+ 55.Kf3 Nxc3 56.Ke3 Nd1+ 57.Ke2 Nc3+ 58.Kd2 Nb5 59.Nf3 Ke7 60.h4 Kf7 61.h5 Kg7 62.g5 a6 63.Ke3 Na7 64.g6 Nc6 65.Ng5 Nd8 66.Nh7 Nc6 67.Nf6 Ne7 68.Ne8+Kh6 69.g7 Nf5+ 70.Kd2 Nxg7 71.Nc7 Kxh5 72.Nxa6 Ne8 73.Nc5 Nc7 74.a6 Kg4

75. a7 Kf5 76.Ke3 Na8 77.Nd7 Kg5 78.Kd2 Kf5 79.Kc3 Ke4 80.Nc5+ Kf5 81.Kb4 Nc7 82.Ka4 Na8 83.Kb4 Nc7 84.Ka4 Na8 85.Kb5 c3 86.Kc6 c2 87.Nb3 Ke4 88.Kb7Kd3 89.Kxa8 Kc3 90.Nc1 Kb2 91.Kb7 Kxc1 92.a8=Q Kb2 93.Qc8 c1=Q 94.Qxc1+ Kxc1 95.Kc6 Kd2 96.Kd6 Kd3 97.Kxe6 Kxd4 98.Kf6

Deep Chess won.

**Game 10**

1. e4 e6 2.d4 d5 3.Nc3 Bb4 4.e5 Qd7 5.Qg4 Bf8 6.Nf3 b6 7.Bd2 Ba6 8.Bxa6 Nxa6 9.O-O Ne7 10.a4 Nf5 11.Ne2 h5 12.Qh3 O-O-O 13.Nh4 Nxh4 14.Qxh4 Be7 15.Bg5 Rde8 16.f4 Nb4 17.Rac1 Nc6 18.Rf3 Qd8 19.Bxe7 Nxe7 20.Qe1 Nf5 21.a5Kb7 22.Ra1 Qe7 23.Rfa3 Ra8 24.Qc3 c5 25.a6+ Kc7 26.Qf3 cxd4 27.c3 Qc5 28.Kh1 Kd7 29.Qd3 Rac8 30.cxd4 Qc4 31.Qd1 Qc2 32.Rc3 Qxd1+ 33.Rxd1 Rxc3 34.Nxc3 Rc8 35.Nb5 Rc2 36.Ra1 Rxb2 37.Nxa7 Nxd4 38.g3 Nf3

Sunfish won

Deep Chess won 6 matches

Draw 2 matches

Lost 2 matches

**Chapter 6 Conclusions**

With the rise of Deep Learning techniques and the recent successes in the game of Go, chess cannot stay behind in these developments. With respect to current chess engines being mostly based on brute-force methods with help from many grandmasters, We are confident that a new chess engine, independent of any restrictions of human knowledge, could have a good chance of improving on the current state-of-the-art concerning chess engines.

Some attempts have already been realized in applying Deep Learning methods to chess, however, Deep Chess is, to our knowledge, the rest engine without hand-crafted features that uses Deep Mind's endings. Deep Chess is based on a 4-layer deep neural network, receiving board positions as input and has trained by playing against itself. In winning positions it is able to win 73 % of the games against Sunfish. However, Deep Chess's performance is worse in hard positions and to compete with brute-force methods, it should eventually win 100 % of winning positions.

On the basis of the results of Deep Chess, We will provisionally conclude that it is possible to apply Deep Learning to the KPK endgame of chess. As future research, Deep Chess should be extended to all chess endgames and eventually the full game of chess, testing against current state-of-the-art chess engines. Only then, it could be confirmed that Deep Learning is a good alternative for the current brute-force methods.

**6.1 Limitations**

* Work perfectly on high configure computer only
* Evaluation function error in exception cases

**6.2 Future Works**

* Time management for different game mode
* Neural network based evaluation
* Improvement in search technique and evaluation function

**References**

Chris J. Maddison, Aja Huang, Ilya Sutskever and David Silver. Move evaluation in Go using deep convolutional neural networks. In International Conference on Learning Representations,2015.

Computer Shogi Association. Results of the 27th world computer shogi championship.

David N. L. Levy and Monty Newborn. How Computers Play Chess. Ishi Press, 2017.

D. E. Knuth and R. W Moore. An analysis of alphabeta pruning. Artificial Intelligence, 6(4):293–326, 1975.

Donald F. Beal and Martin C. Smith. Temporal difference learning for heuristic search and game playing. Inf. Sci., 122(1):3–21, 2000.

Donald F. Beal and Martin C. Smith. Temporal difference learning applied to game playing and the results of application to shogi. Theoretical Computer Science, 252(1–2):105–119, 2001.

Emanuel Lasker. Common Sense in Chess. Dover Publications, 1965.

Feng-hsiung Hsu. Behind Deep Blue: Building the Computer that Defeated the World Chess Champion. Princeton University Press, 2002.

Hiroyuki Iida, Makoto Sakuta and Jeff Rollason. Computer shogi. Artificial Intelligence, 134:121–144,2002.

J. Baxter, A. Tridgell and L. Weaver. Learning to play chess using temporal differences. Machine Learning, 40(3):243–263, 2000

John Knudsen. Essential Chess Quotations. iUniverse, 2000.

Kunihito Hoki and Tomoyuki Kaneko. Large-scale optimization for evaluation functions with minimax search. Journal of Artificial Intelligence Research (JAIR), 49:527–568, 2014.

Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. Master’s thesis, Imperial College London, 2015.

M. Campbell, A. J. Hoane and F. Hsu. Deep Blue. Artificial Intelligence, 134:57–83, 2002.

Norman P. Jouppi, Cliff Young, Nishant Patil, et al. In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA ’17, pages 1–12. ACM, 2017.

Oleg Arenz. Monte Carlo chess. Master’s thesis, Technische Universitat Darmstadt, 2012.

Omid E David, Nathan S Netanyahu and Lior Wolf. Deepchess: End-to-end deep neural network for automatic learning in chess. In International Conference on Artificial Neural Networks, pages 88–96. Springer, 2016.

Online chess games database, 365chess, 2017.

R. Coulom. Whole-history rating: A Bayesian rating system for players of time-varying strength. In International Conference on Computers and Games, pages 113–124, 2008.

Thomas Anthony, Zheng Tian and David Barber. Thinking fast and slow with deep learning and tree search. In Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA, pages 5366–5376, 2017.

Tomoyuki Kaneko and Kunihito Hoki. Analysis of evaluation-function learning by comparison of sibling nodes. In Advances in Computer Games - 13th International Conference, ACG 2011, Tilburg, The Netherlands, November 20-22, 2011, Revised Selected Papers, pages 158–169, 2011.

Victor Allis. Searching for Solutions in Games and Artificial Intelligence. PhD thesis, University of Limburg, Netherlands, 1994.