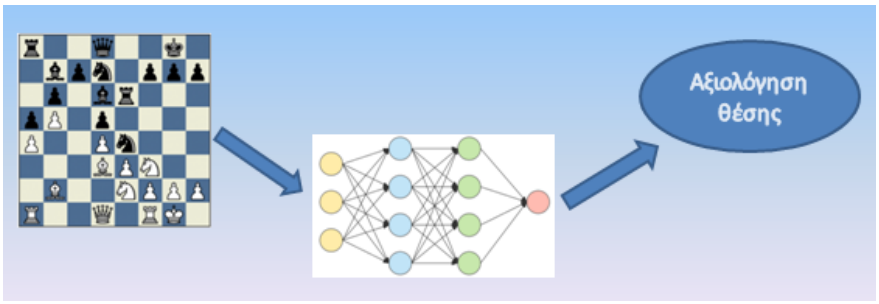
ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ & ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ

Πρόγραμμα Μεταπτυχιακών Σπουδών
«ΔΙΑΔΙΚΤΥΟ ΤΩΝ ΠΡΑΓΜΑΤΩΝ ΚΑΙ ΕΥΦΥΗ ΠΕΡΙΒΑΛΛΟΝΤΑ»

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αξιολόγηση σκακιστικών θέσεων με χρήση νευρωνικών δικτύων



Μεταπτυχιακός Φοιτητής: Δημήτριος Κάγκας, ΑΜ msciot18001
Επιβλέπων: Αλέξανδρος Αλεξανδρίδης, Καθηγητής

ΑΙΓΑΛΕΩ, ΙΟΥΛΙΟΣ 2021

UNIVERSITY OF WEST ATTICA

FACULTY OF ENGINEERING

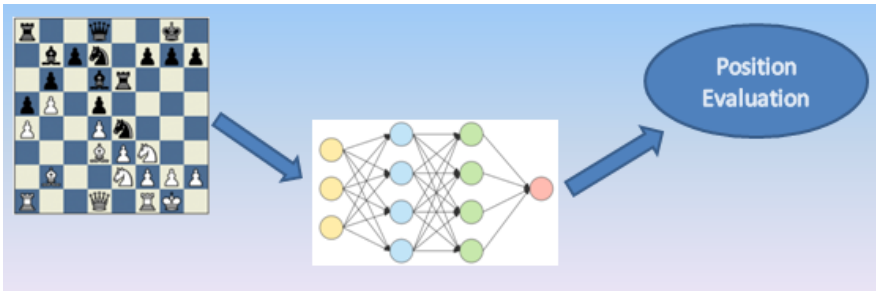
DEPARTMENT OF ELECTRICAL & ELECTRONIC ENGINEERING

Master of Science in

"INTERNET of THINGS AND INTELLIGENT ENVIRONMENTS"

MSc Thesis

Chess position evaluation using neural networks



Student: Kagkas, Dimitrios, Registration Number msciot18001

MSc Thesis Supervisor: Alexandridis, Alex, Professor

ATHENS-EGALEO, JULY 2021

Η Διπλωματική Εργασία έγινε αποδεκτή και βαθμολογήθηκε από την εξής τριμελή επιτροπή:

Αλεξανδρίδης Αλέξανδρος Καθηγητής (Επιβλέπων)	Φαμέλης Ιωάννης Καθηγητής (Μέλος)	Πατρικάκης Χαράλαμπος Καθηγητής (Μέλος)
 Digitally signed by Alexandros Alexandridis Date: 2021.07.22 23:53:06 +03'00'	 Digitally signed by Ioannis Famelis Date: 2021.07.22 18:31:49 +03'00'	 Digitally signed by Charalampos Patrikakis Date: 2021.07.22 23:44:16 +03'00'

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Ο κάτωθι υπογεγραμμένος Κάγκας Δημήτριος του Σπυρίδωνος, με αριθμό μητρώου msciot18001, φοιτητής του Πανεπιστημίου Δυτικής Αττικής της Σχολής ΜΗΧΑΝΙΚΩΝ του Τμήματος ΗΛΕΚΤΡΟΛΟΓΩΝ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ,

δηλώνω υπεύθυνα ότι:

«Είμαι συγγραφέας αυτής της διπλωματικής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος.
Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του διπλώματός μου».

Ο Δηλών

Κάγκας Δημήτριος

ΠΕΡΙΛΗΨΗ

Το σκάκι αποτελεί το πιο συχνά χρησιμοποιούμενο παιχνίδι στο πεδίο της τεχνητής νοημοσύνης και της μηχανικής μάθησης. Πολλές προσεγγίσεις έχουν προταθεί όπου οι υλοποιήσεις επιχειρούν να αντικαταστήσουν μέρη ή και ολόκληρη τη λειτουργικότητα των σκακιστικών μηχανών. Σε αυτή την εργασία προτείνουμε μία μέθοδο για την εύρεση της αξιολόγησης μιας σκακιστικής θέσης χωρίς να γίνεται χρήση αλγορίθμου αναζήτησης δέντρου και εξέταση κάθε πιθανής κίνησης ξεχωριστά, όπως θα έκανε μια σκακιστική μηχανή. Αντί για την επεξεργασία του δέντρου αναζήτησης ώστε να εξεταστεί η θέση σε βάθος αρκετών κινήσεων, προτείνουμε τη χρήση προβλέψεων ενός κατάλληλα εκπαιδευμένου νευρωνικού δικτύου, που εξασφαλίζει πολύ μεγαλύτερη ταχύτητα και μικρότερες απαιτήσεις σε επεξεργαστική ισχύ. Το βασικό πλεονέκτημα αυτής της προσέγγισης είναι η λήψη πρόβλεψης της αξιολόγησης μιας σκακιστικής θέσης σε χρόνους της τάξης των χιλιοστών του δευτερολέπτου ενώ μια σκακιστική μηχανή θα χρειαζόταν εώς και μερικά λεπτά για να επιτύχει το αντίστοιχο αποτέλεσμα. Η προτεινόμενη προσέγγιση περιλαμβάνει ένα νέο σύνολο χαρακτηριστικών ως μεταβλητές εισόδου, σε συνδυασμό με μοντέλα νευρωνικών δικτύων βασισμένα στην αρχιτεκτονική συνάρτησης ακτινικής βάσης (RBF) τα οποία εκπαιδεύονται με τον αλγόριθμο fuzzy means. Δύο διαφορετικές μέθοδοι εκπαίδευσης νευρωνικών δικτύων επίσης εξετάστηκαν και συγκρίθηκαν, οι οποίες αφορούν την αρχιτεκτονική perceptron πολλαπλών στιβάδων. Όλες οι μέθοδοι βασίστηκαν στο ίδιο σύνολο δεδομένων το οποίο προέκυψε μετά από επεξεργασία περισσότερων από 1500 παρτίδων παικτών κορυφαίου επιπέδου. Για το σκοπό αυτό αναπτύχθηκε μια εφαρμογή σε γλώσσα Java για την εξαγωγή συγκεκριμένων χαρακτηριστικών από την κάθε σκακιστική θέση στις παρτίδες αυτές, ώστε να γίνει η κατασκευή του συνόλου των δεδομένων εκπαίδευσης, το οποίο περιείχε δεδομένα από 81967 θέσεις. Πολλαπλά νευρωνικά δίκτυα εκπαιδεύτηκαν με σκοπό την εξέταση διαφορετικών εκδοχών της κάθε μεθόδου, σχετικά με την επιλογή μεταβλητών εισόδου καθώς και φιλτραρίσματος των δεδομένων. Τα αποτελέσματα της διαδικασίας έδειξαν πως η προσέγγιση με την αρχιτεκτονική RBF ήταν η καλύτερη σε απόδοση. Τα μοντέλα που παράχθηκαν με την προτεινόμενη προσέγγιση είναι κατάλληλα για ενσωμάτωση σε δομές λήψης αποφάσεων που βασίζονται σε μοντέλα, π.χ. σε μεθοδολογίες ‘model predictive control’ (MPC), οι οποίες μπορούν να γίνουν η βάση για την ανάπτυξη ενός ολοκληρωμένου σκακιστικού λογισμικού.

ΛΕΞΕΙΣ – ΚΛΕΙΔΙΑ: αλγόριθμος fuzzy means, αξιολόγηση σκακιστικής θέσης, νευρωνικά δίκτυα, σκακιστική μηχανή, συνάρτηση ακτινικής βάσης, perceptron πολλαπλών στιβάδων

ABSTRACT

The game of chess is the most widely examined game in the field of artificial intelligence and machine learning. There are many approaches where implementations attempt to substitute parts, or the whole functionality of a chess engine. In this Thesis we propose a method for obtaining the evaluation of a chess position without using tree search and examining each candidate move separately, like a chess engine does. Instead of exploring the search tree in order to look several moves ahead, we propose to use the much faster and less computationally demanding predictions of a properly trained neural network. Such an approach offers the benefit of having a prediction for the position evaluation in a matter of milliseconds, while a chess engine may need even minutes to achieve the same result. The proposed approach introduces a novel set of input features, in conjunction with models which are based on the radial basis function (RBF) neural network architecture and trained with the fuzzy means algorithm; two different methods of network training are also examined and compared, involving the multilayer perceptron (MLP) network architecture. All methods were based upon the same dataset which was derived by a collection of over 1500 top-level chess games. A Java application was developed for processing the games and extracting certain features from the arising positions in order to construct the training dataset, which contained data from 81967 positions. Various networks were trained and tested as we considered different variations of each method regarding input variable configurations and dataset filtering. Ultimately, the results indicated that the proposed approach using the RBF method was the best in performance. The models produced with the proposed approach are suitable for integration in model-based decision making frameworks, e.g. model predictive control (MPC) schemes, which could form the basis for a fully fledged chess playing software.

KEYWORDS: chess engine, chess position evaluation, fuzzy means, multilayer perceptron, neural networks, radial basis function

ACKNOWLEDGEMENTS

I would like to express my special thanks of gratitude my professor Dr. Alex Alexandridis for his guidance during our excellent collaboration. I would also like to thank Despoina Karamichailidou for her valuable contribution during the experimental process.

TABLE OF CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

SECTION 1

Introduction

In this Thesis we explore the possibility of utilizing the learning capabilities of neural networks in order to approximate the evaluation score of a chess position. This is typically achieved by a chess engine which assesses a position by performing a tree search on the possible continuations and applying min-max and alpha-beta pruning algorithms.

Chess is an immensely interesting and entertaining game. The intellectual challenge it offers resembles the one in puzzles and also involves strategic and tactical thinking which fascinates players and fans of the game throughout the ages. As computers evolved and the field of software engineering came to be, chess acquired a form of computer application in addition to its previous board game form. It was only a matter of time before the idea of a computer player was conceived and implemented and in the year 1996 the first 'Man vs Machine' match, between the chess world champion at the time, Garry Kasparov, and IBM's Deep Blue, took place [1].

Although Kasparov did win that match, the continuous improvement of computers in terms of processing power and the use of better and more suitable to the problem search algorithms have resulted in a huge rise of the capabilities of chess engines. Nowadays even top level grandmasters not only are no match for them but they in fact use them for their tournament preparation and overall training, and moreover, championships among engines are held. Advancements in the fields of artificial intelligence and machine learning have led to new ideas about engines like chess playing agents with no coding of any chess rules in them whatsoever and enhancements on the position evaluation algorithms with the aid of neural networks replacing for example the heuristic function used in the tree search.

Many demonstrations of ideas involving the game of chess and machine learning currently exist. One research on the optimization of the handcrafted evaluation function has demonstrated the use of evolutionary algorithms for tuning the function parameters with a strategy named 'dynamic boundary strategy' [2]. A similar approach on the subject has been to represent the individuals in the population as virtual players, each of them using different, fixed parameters for their evaluation function. These players compete for advancing to the next generation of the algorithm, not to each other but are ranked using real top-level games [3]. Another example of the use of evolutionary algorithm, in combination with neural networks, has been a program that learned to play the game by playing against itself [4].

The use of neural networks has also been very common in researches in this domain, as in the case of DeepChess, that used a neural network trained to evaluate positions using millions of games but no other chess-related knowledge including the rules of the game [5]. KnightCap also used a neural network along with custom made attack and defence tables and piece weights [6]. Another case is NeuroChess, a chess playing program that relies on neural networks and handcrafted features for evaluating positions, trained to predict the result of games [7]. The use of neural networks is taken one step further by the Giraffe, a chess program that acquired chess knowledge by self-play, that utilizes them not only to tune the evaluation function parameters but also to perform pattern recognition and feature extraction

[8]. Pattern recognition has also appeared in a research involving multilayer perceptrons and convolutional neural networks in order to evaluate positions by using look-ahead algorithms as little as possible [9]. Probably the most impressive, however, is the famous AlphaZero which, having no knowledge besides game rules and training solely by self-play for 24 hours, managed to defeat the world champion engines in the games of Chess, Shogi and Go [10].

Many of the aforementioned publications rely on neural networks for various purposes, focusing mainly on the architectures of convolutional neural networks (CNNs) and multilayer perceptrons (MLPs). A candidate architecture for an alternative approach could be the radial basis function networks (RBFs) [11] that present important advantages, especially over MLPs, in terms of structure, training process and optimization.

Our goal is to train a neural network in order to approximate the absolute evaluation score of a chess position, without performing the kind of search chess engines do. We intend to investigate and propose the architecture that achieves the maximum accuracy in predicting the evaluation of a chess position to an acceptable degree, but without looking into specific moves or exploring the search tree at all. Such a method would provide predictions for several moves ahead but in a much shorter timeframe than a chess engine would need to achieve the same result. The networks will use a proposed set of chess position features, derived from the ones utilized by the Stockfish engine. Also, they will have no domain specific knowledge, nor will any kind of assisting structures, like lookup tables, be used. The examined neural network architectures are MLPs and RBFs. The source of our training data is a collection of 1514 top-level chess games. All the necessary processing, including breaking the games down to separate positions, 81967 in number, extracting the proposed set of features from them and building a database to construct the training dataset, is done by a custom made Java application developed specifically for this purpose. All examined methods are primarily compared in terms of two basic indicators, the mean absolute error and the coefficient of determination, and secondarily in network training time. Moreover, the examination of each method consists of different variations in order to evaluate the effect of certain input variables in the training dataset and also of different filtering applied to the dataset itself.

The remaining of this Thesis is structured as follows: in the following section an introduction to chess engines is made. Necessary terminology as well as basic knowledge for communicating with them is provided, followed by an explanation of the position evaluation process. At the end of the section, the basic components that compose the evaluation heuristic function of the Stockfish engine are briefly described. Section 3 contains a theoretical presentation about neural networks, the concept of learning and the existing architectural patterns. More specific analysis is made for the multilayer perceptron and the radial basis function architectures along with descriptions of the most common training algorithms. In section 4 the entire experimental procedure is described in detail, including the formation of our proposed set of position features and the description of game processing and network training, results are presented and discussion is provided. Finally, section 5 is the conclusion of the Thesis followed by references.

SECTION 2

Chess Engines

Chess is a very popular game but an extremely complicated one as well. Learning how to play chess, regarding rules and gameplay, is easy enough, unlike the understanding of both simple and sophisticated techniques and patterns, and their underlying ideas, that make the difference in the level of skill among players. After mastering the mechanics of the game, chess players initially learn to execute routine sequences of moves, following simple ideas, in specific situations like a king and rook endgame, which mainly consist of very few pieces. Later they learn to recognize various patterns in a position regarding tactical motifs, like a double attack, and execute them, progressively calculating longer sequences of moves that lead to them and studying positions with more pieces where the candidate moves are many more. In a next stage, basic principles and strategies are introduced in the form of guidelines that have derived from deep analysis that the novice player does not need to have studied, such as the general opening principles, or a simple plan that applies to certain types of positions, like putting pressure on a weak pawn along an open file. As the skill level of the player increases, these principles and plans are brought together by broader strategic themes that appear in specific openings, or middlegame and endgame situations, and by understanding them, obeying to the appropriate principles or choosing the correct plan, becomes less dogmatic and more the output of a logical process. Advanced players take this process one step further, where they can produce their own long-term strategic plans based on the characteristics of any position, that dictate what short-term plans need to be executed. This is necessary when having to play a complicated position deep in the middlegame, when prior analysis is highly unlikely to have been done and needs to be calculated on the spot.

In essence, human players learn to recognize the characteristics and patterns appearing in a position and, based on their abstract evaluation of them, judge which strategic plans are more appropriate. Taking this to a more general point of view, players take characteristics, possible strategic plans and their calculation of future sequences of moves that seem more important according to these plans into consideration and determine if a given position is equal or better for either side, and to what extent. The various characteristics of a position such as material, mobility, king safety or control of the centre, are obviously the source of any evaluation of the position.

In case of humans, these characteristics are considered in a vague, instinctive and non-quantified manner, that form an abstract evaluation mostly based on experience and judgment, which is usually loosely expressed like ‘White is slightly better’ or ‘Black is winning’. A computer, on the other hand, needs a more concrete and quantified definition of both the characteristics and the evaluation, in order to be able to execute any calculations and produce an outcome. Programs designed to execute such processes are called chess engines. Chess engines process a given position internally, in the form of some kind of representation, and are able not only to provide an evaluation of the position but also suggest continuations of moves that they consider the strongest. Popular chess engines include Stockfish, Komodo, Houdini and Rybka.

2.1 Chess Engine API

As mentioned above, in order for a chess engine to perform any kind of process on a chess position, it must be provided with some kind of representation of the position. Furthermore, chess software developed to encapsulate engines also needs a form of representation, not just of single positions but of whole games, which could be parsed and then analyzed. In addition to that, the software also needs to generate a representation of a chess game when a user produces one by making moves in its graphical interface. Many types of notations have been introduced in order to deal with these needs, the most popular being the Forsyth-Edwards notation (FEN), for the position representation, and the portable game notation (PGN), for the game representation, both specified in 1994 in the 'Portable Game Notation Specification and Implementation Guide' standard [12].

Moreover, the communication between a chess engine and any software, either being a wrapper software like a chess playing user interface or just a command line where the engine is run as a standalone application, needs to be defined, as any other communication, by a protocol. Most contemporary chess engines implement the 'universal chess interface' (UCI) communication protocol, released in 2000, which gradually replaced the older 'chess engine communication protocol' in popularity in the following years, although some engines still support that. It was designed by Stefan Meyer-Kahlen who was actually the author of a commercial chess engine called Shredder.

2.1.1 PGN

PGN is actually plain text in a standardized format, developed by Steven J. Edwards, readable by both humans and software, which is used to record moves and related data of a chess game [13]. It is divided into two segments, the former of which contains the game information. Each piece of game information is enclosed between square brackets, which are referred to as tags. As per standard, there are seven mandatory tags that must appear before any other optional tags, in a specific order: event, site, date, round, white, black, and result. The latter segment of the PGN format contains the movetext, i.e. the moves of the game, in 'standard algebraic notation (SAN). In order for this text is to be used by software, it is encapsulated in a file with .pgn extension. An example of a PGN can be seen in figure 1.

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 {This opening is called the Ruy Lopez.}
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

Figure 1: PGN example

2.1.2 FEN

FEN is used to describe a specific position of a game. Its purpose is to encapsulate all the information needed in order to resume a game from a given position. It was initially developed by David Forsyth and later Steven J. Edwards extended it so that it would be usable by chess software [14].

A FEN is a single line of text that consists of six fields separated by a space, in the following order:

- *Piece placement*: A sequence of characters that depicts the placement of the pieces on the chess board. Regarded from White's point of view, ranks are considered from the eighth to the first, each one expressed as a subsequence of characters representing the state of the squares on the rank, with respect to the order of the files on the chess board, A to H. Ranks are separated by forward slashes and pieces are represented with their respective letter, as described by the standard algebraic notation, in upper case letters for White and lower case letters for Black. Lastly, digits are used to indicate the number of consecutive unoccupied squares among pieces in a rank.
- *Active color*: A lower case letter, either 'w' or 'b' for White or Black respectively, indicating the side to move next.
- *Castling availability*: A string that describes the castling rights for both sides. This may contain the letters K, Q, k and q that correspond to the kingside and queenside castling for White and Black respectively, or a dash when any castling is no longer available. The relevant letter is removed of the string only when a move that permanently loses rights for the specific castling has been made. In positions where a castling is temporarily not allowed the relevant letter is not removed, as rights have not been permanently lost.
- *En passant target*: The square on which a pawn will arrive if it was to perform an en passant move on the next move, regardless of whether such a move is available. If no such square exists, i.e. no two-square pawn move was made in the previous move, a dash is used instead.
- *Halfmove clock*: The number of consecutive plies (moves counted separately for each side) where no capture has occurred and no pawn has moved.
- *Fullmove clock*: The sequence number of full moves. A full move is counted when both sides have made a move (ply).

According to the description above, the FEN for the initial position of a chess game would be:

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

where all the black pieces and pawns are on the eighth and seventh rank, the sixth to third rank contain only unoccupied squares, all the white pieces and pawns are on the first and second ranks, White is to move next, all castling rights stand, no en passant target is available, no half moves have been played and the first full move is being played. An example of a position after the moves 1.e4 e5 2.Nf3 Nc6 3.Bc4 can be seen in figure 2.

Figure 2: Position example

In this case, the FEN for the position would be:

r1bqkbnr/pppp1ppp/2n5/4p3/2B1P3/5N2/PPPP1PPP/RNBQK2R b KQkq - 3 3

2.1.3 UCI protocol

The UCI protocol specification [15], as described by Stefan Meyer-Kahlen, defines valid commands for communicating with a chess engine, and the expected responses sent from the engine back to the caller.

The necessary commands for basic communication with an engine would be the following:

- uci: Informs the engine that communication will be based on the UCI protocol. This must be the first command sent in a chess engine session.
- isready: Used for synchronization between the engine and the calling software. The engine will respond to this command if it is still alive and has finished any previously requested processing.
- setoption name [optionName] value [optionValue]: Sets the internal parameters of the engine. The available options and values depend on the engine version.
- position [fen]: Sets the internal board of the engine to the position described by the FEN string. The default keyword 'startpos' can be used instead of FEN to initialize the board at the starting position.
- go [settings]: Initiates the search process of the engine on the position that has been set at the internal board. Various optional search settings can be used with this command, the most important being 'depth x' that indicate searching for x plies (half-moves), 'movetime x' for searching for x milliseconds and 'infinite' for searching until the stop command has been received.
- stop: Stops the searching process.
- quit: Terminates the chess engine process.

The most important responses the engine sends back to the caller are:

- uciok: Acknowledges the communication via the UCI protocol.
- readyok: Indicates the engine is ready to receive commands, after it has been pinged with the isready command.
- bestmove: Reports the best move found for the specified position, as a result of the last search.

- info: The main response for providing search results. It consists of many components, the most important being ‘depth’ indicating the ply count at which the search is currently at, ‘currmove’ indicating the move that is currently processed, ‘pv’ reporting the best sequence of moves found so far, and ‘score’ indicating the evaluation of the position at that point of the search. Such a response is sent every time any of its components changes, during the execution of the search process.

Obviously, the most interesting engine output is the evaluation of the position, i.e. the score component of the ‘info’ response. This has four possible expressions depending on the specific position and the available time the engine had to process it. The main expression is ‘cp’ which is the evaluation in ‘centi-pawns’ from the point of view of the engine. This means that if a position with White to move had a positive evaluation, after the best move and when Black would be to move, the evaluation would be the same in value but negative, indicating that the side to move (black) is worse by that much. Typically, chess software wrapping a chess engine inverse the evaluation sign when Black is to move in order for the evaluation to always refer to White, resulting in the more intuitive output in which positive evaluations indicate that White is better by that value, and negative evaluations indicate Black is better by that value, eliminating the impact of the side to move on the display of the evaluation.

Also the term ‘centi-pawns’ needs some clarification, as it can be misleading. What this seems to imply is that each unit of evaluation corresponds to 1/100 of the value of a pawn, which would subsequently mean that an evaluation of +100 returned by the engine is translated to White being a pawn up (displayed as +1.00 at the user interface). This is not always the case, as it is a matter of implementation in each individual chess engine. In Stockfish for example, the unit of evaluation corresponds to 1/213 of the value of a pawn meaning the indication of Black being a pawn up in a position is in fact -213. It must also be noted that, in general, such an evaluation does not necessarily mean that Black actually has one pawn more than his opponent on the chess board but rather that, taking every aspect of the position into consideration, Black’s advantage is analogous to having one more pawn.

Apart from ‘cp’, another expression of the score component is ‘mate’. This substitutes ‘cp’ when the engine has found a mating sequence and indicates not an evaluation but the number of moves (full-moves in this case, not plies) in this sequence until the mate position occurs. The rest of the expressions are ‘lowerbound’ and ‘upperbound’ and are used by the engine when the searching process for the specific depth has been stopped before determining the actual ‘cp’ so the engine reports that the value is either an upper or a lower bound to the actual evaluation.

2.2 Evaluation

As mentioned above, the evaluation of a chess position is depicted as a signed value that represents which side is better and by how much. Although chess engines calculate this value as an integer amount of centi-pawns, whatever a centi-pawn stands for in the context of each engine, the wrapping interfaces transform it to a signed decimal number with two decimal places, normalized to the value of a pawn in the human perspective. The calculation of the evaluation value is a complicated process that each chess engine implements in its own way, but all of them share the same general approach - a tree search.

2.2.1 Search

In a practical sense, the goal of a chess engine is to provide the best move for a given position. In order to determine the best move, engines need to look ahead at various sequences of moves and evaluate the arising positions, since in general such a result cannot be concluded from the static information of a position [16]. The algorithms used for searching depend on the nature of the examined game. Chess is a two-player game that belongs to the family of zero-sum games with perfect information. In short, zero-sum characterizes a game where the amount a player is winning by is exactly equal to the amount the other player is losing by, or more generally, the total of gains and losses of all players is exactly zero. Perfect information means that any factor that affects the decision making process of a player is known at all times, i.e. in chess both players always know the location of all the pieces on the board. In these types of games, searching involves traversing a search tree (or game tree) using a 'min-maxing' algorithm.

A search tree is a directed graph of nodes, also called states or vertices, which are linked by directed edges, also called arcs or state transitions. In the case of chess, each node represents a position on the board and each edge represents the move that resulted to the respective position. The root node of the tree is the position under question; the one being evaluated and searched for the best move. Possible transpositions of a position, resulting in cycles in the search tree, are eliminated by the search algorithm and are not searched again. The leaf nodes represent mate or statemate positions or positions at a certain depth which are assigned an evaluation. The evaluation of the leaf nodes is calculated based on a function of various static characteristics of the respective position called 'features'. In order for the search to take place, the search tree needs to be constructed first so the engine starts from the given position (root node) and creates the next generation of nodes by making every possible move on its internal board structure. By repeating this process for every generated node, the tree gradually increases in depth. Obviously, it is not possible to construct the whole game tree except for positions that are close to the end of the game.

In games like chess the algorithm used for searching the tree is the MinMax algorithm. Specifically in chess, an enhancement of the MinMax algorithm is used called Alpha-Beta algorithm or Alpha-Beta pruning. The idea of min-maxing in a two-player game is that one side tries to achieve the highest possible evaluation (maximizing player) and the other side tries for the lowest possible evaluation (minimizing player) when it is their turn to play. This dictates that the flow of processing is actually opposite to the construction of the search tree, since the evaluation of every parent node depends on the evaluation of its children nodes. In essence, all the leaf nodes are evaluated and then each parent node is assigned the appropriate value (minimum or maximum) from its children, depending on whether they are at a minimizing or maximizing level of the tree. This process continues all the way up to the root node and the value assigned to it is the evaluation of the current position. The Alpha-Beta pruning enhancement suggests that, due to min-maxing logic, some branches of the tree can be ignored (pruned) if it is certain that they cannot provide any better value to their parent than the one it already has been assigned. For example, if a node that exists on a minimizing level of the tree has been assigned the value -37 from one of its previously processed children, and the currently processed child node has acquired the value of +25 so far, from traversing its own first child hierarchy, since this node exists on a maximizing level, the value of 25 is the lowest it can be. Subsequently, it is not possible to offer its minimizing parent any value lower than -37 that it already has, since 25 is as low as it can get, thus there is no point in processing the rest of its children and so these branches are pruned.

Following this concept, the engine applies the search algorithm iteratively to the continuously growing search tree that it constructs. Beginning from the current position, it generates all the first level children nodes (depth 1) and searches the tree with these being the leaf nodes. The evaluation is assigned to the root node and the search at depth 1 is concluded. It then generates the nodes at depth 2 and applies the search algorithm again with these nodes being the leaf nodes and so on. It is important to note that in every iteration any previous values assigned to upper level nodes are disregarded since they will acquire their values from processing the current state of the tree, with leaf nodes at one level deeper. The search continues in this manner until the desired depth is reached or the available time has expired.

When the search is terminated, a path can be formed connecting the root node to a leaf node, the evaluation of which is the one that has bubbled all the way up the tree. Since the edges between the nodes represent actual moves, this path is translated to the optimal sequence of moves as a continuation from the current position. In fact the engine constructs such a sequence for every depth level that finishes searching at. These sequences are not necessarily related especially for small depths, considering that the search is re-executed for every new depth level. What seemed as a good continuation when the position was examined at a depth of 2 to 4 might be refuted in a depth of 7.

So, what a chess engine actually returns as the evaluation of the current position is in fact the static evaluation of a future position that is reached after optimal moves are played for both sides. How far in the future in terms of moves this position is, largely depends on computational power and available time combined with the complexity of the arising positions as more computation time is needed to evaluate each of them. This means that the value that comes up is always an approximation of the actual evaluation of the position [17]. If it was possible to process any position to the very end, the outcome of the search would indicate win, draw or loss with absolute certainty. This is actually the case with positions closer to the end of the game; the engine returns the number of moves until mate instead of an evaluation, or the value of zero indicating a drawn position, provided that the best moves are played for both sides.

2.2.2 Features

While this is the common mechanism that chess engines use to evaluate a position, the static evaluation function, used to evaluate every leaf node position based on its features is what makes the difference between them. Every engine implementation has its own logic of what features are taken into consideration and how they are weighted, giving more or less importance to each one. Moreover, the evaluation function itself differs not only in its definition but also in the way that it is determined, which may be from chess experience and collective knowledge of top-level grand masters, processing of chess game databases, use of machine learning techniques or, most probably, a combination of these.

In any case, the value that is ultimately produced by the evaluation function is the result of a computation involving the position features. Features are value representations related to the actual pieces on the chess board as a means of quantifying aspects like their absolute or relative value, their location, mobility or their interaction with other pieces, either being as simple as a check or more complex, based on specific patterns. The actual method of calculating each one is also a matter of specific engine implementation, but a general rule is that specific weights are assigned based on the calculated values which differ according to the game phase. It should be pointed out that since each feature is calculated for each player, it is the difference between the respective counterparts that makes an impact on the total

evaluation. Depending on its nature, each feature can be considered as a component of a group that represents a more abstract chess concept. The most notable among such concepts are material, mobility, king safety, pawn structure and piece-specific patterns.

Material seems the most concrete and easily quantifiable concept. A predetermined value is assigned to each type of piece for its existence on the board, regardless of any positional aspects. Such a value is the absolute material value of a piece. The values are 1 for pawns, 3 for bishops and knights, 5 for rooks and 9 for queens. Some usually used values for kings are 10, 20, 200 and infinite as it is really a matter of personal opinion since both players can only have exactly one king so they both receive its value in their total and the difference is always zero. This is actually how a beginner human player would count material and determine equality for the position based on the difference.

When it comes to more experienced players and chess engines, a more subtle idea is also taken into account. Additionally to its absolute value, each piece is also assigned a relative value depending on its placement on the board. For human players this is more of a conceptual estimation, characterizing their pieces as 'good' (well-placed) or 'bad' (poorly-placed). The criteria for this estimation differ according to the phase of the game. For example, a rook on the opponent's seventh rank in the middlegame or an advanced pawn in the endgame would be considered good pieces, resulting in estimating their total value as much more than their absolute material value actually is. In essence, the relative value of a piece could be considered as a bonus or penalty to its absolute value. Engines implement this idea by utilizing specific structures known as piece-square tables. This is a collection of arrays that consists of an array for each piece type which matches each combination of ranks and files (squares) to a positive or negative score. These tables are usually created and tuned by humans, based on chess experience and knowledge and are most probably different for each phase of the game.

Mobility is related to the available legal moves for each side. This could be simply considered as the sum of the legal moves for each player and it is definitely valid for a human player to do so. In case of chess engines however, the calculation is a little more sophisticated. More specifically, each piece type is given a bonus or penalty based the number of legal moves at its disposal, with the assistance of structures similar to piece-square tables that contain this correspondence of values. These structures are also hand-crafted and vary among engine implementations and they also differ according the phases of the game. The sum of the bonuses/penalties for all the pieces on the board for each side is the value that is actually used as mobility. Different implementations apply various rules to what they count as legal moves like excluding moves of pinned pieces or including moves to squares that friendly pieces occupy, i.e. supporting them, despite not being technically legal. On top of these rules, most engines only consider what they call 'safe mobility' in which the moves to squares where the moving piece is in danger are excluded from the total of legal moves for a piece. As this can be an expensive calculation, in some cases a more simplistic method is used like excluding squares controlled by enemy pieces of less absolute value or just by enemy pawns.

King safety is the most complex of all the evaluation concepts. It is composed from many features regarding specific aspects like the setup of the defending pawns, castling rights, the placement and distance of defending and attacking pieces near the king, checks and threats, weak squares around the king and a lot more. The actual components used to measure king safety are not the same across engines but they more or less follow these patterns. Ultimately, a weighted total of all the different features provide the main evaluation function with a value for king safety. As is the case for all features, the weight values have to be predefined in some way and definitely vary according to the game phase.

Pawn structure is another important factor in the position evaluation. By definition, pawn structure a term used to describe the position of all pawns regardless of the placement of other pieces but including the placement of other pawns. Features in this category represent the state of each individual pawn like doubled, isolated, backward, blocked, connected, passed etc. In a more generalized sense though, there are some features that consider more pieces relatively to the examined pawn in a specific manner like setups that make it a candidate passed pawn or a weak pawn.

Piece-specific patterns include a wide variety of features regarding the placement of pieces in specific circumstances, relatively to other pieces and pawn structure. Such patterns are outposts, which are squares that cannot be attacked by enemy pawns, rooks on open or semi-open files, or on the same file as the enemy queen or attacking the enemy king area, bishops on long diagonals, minor pieces behind friendly pawns, trapped pieces, pieces near their own king aiding in its protection, pawns on the same square color as a bishop, x-ray attacks, which are attacks on squares that would occur if an interfering piece was removed, weak pieces, hanging pieces, safe pieces and pawns, and many more. These features are measures by number of occurrences on the board and weighted accordingly.

2.2.3 Stockfish Features

Our proposed set of features is derived from the implementation of the Stockfish chess engine. Stockfish is an open source engine, licensed under the GPL v3.0 and compatible with the UCI protocol. It was developed by Tord Romstad, Marco Costalba, Joona Kiiski and Gary Linscott and it is the strongest chess engine in the world as of 2018 [18].

Table 1 briefly presents the feature components and table 2 presents the function components of the tree search heuristic function that Stockfish relies on, as described in the 'Stockfish evaluation guide' [19]. It is important to notice that Stockfish does not necessarily use every feature mentioned in table 1 individually, but rather combines or aggregates them. Some of the features in the table are the actual aggregations of other features, or their calculation depends on the values of others. Also, Stockfish calculates most features for both sides, except when the feature is not side specific by its nature, but ultimately uses the difference of the two counterparts.

It should be noted that the chess board segment that consists of the square occupied by the king along with the eight squares surrounding it, with the exclusion of the squares defended twice by pawns, is referred to as 'king ring'.

Table 1: Stockfish evaluation feature components

Material	
Non pawn material	Weighted value of (non-pawn) pieces in the middlegame
Piece value mg	Weighted value of pawns and pieces in the middlegame
Piece value eg	Weighted value of pawns and pieces in the endgame
PSQT mg	Piece square table bonuses in the middlegame

PSQT eg	Piece square table bonuses in the endgame
Mobility	
Mobility	Number of attacked squares in mobility area
Mobility Area	Subset of all legal moves based on certain conditions
Mobility mg	Mobility-related bonus in the middlegame
Mobility eg	Mobility-related bonus in the endgame
King	
Pawnless flank	Penalty for king being in a pawnless flank
Strength square	Bonus/Penalty for squares being a king shelter
Storm square	Bonus/Penalty for enemy pawn stormed squares
Shelter strength	Bonus for shelter of the king position
Shelter storm	Penalty for the king position being pawn stormed
King pawn distance	Minimum distance of king to friendly pawns
Check	Number of squares where a check can be given on the next move, without counting the queen as blocker
Safe check	Number of safe squares on which a check can be given on the next move
King attackers count	Number of pieces that attack the enemy king ring
King attackers weight	Sum of weight of the pieces that attack the enemy king ring
King attacks	Number of attacks on squares adjacent to enemy king
Weak squares	Number of attacked squares defended only once by the enemy king or queen
Weak bonus	Number of weak squares in enemy king ring
Unsafe checks	Number of unsafe squares on which a check can be given on the next move
Knight defender	Number of squares around the king defended by a knight
Endgame shelter	Compensating endgame penalty for shelter storm

Blockers for king	Number of pieces currently blocking checks
Flank attack	Penalty for squares in the king flank that are attacked once and twice
Flank defence	Number of defended squares in the king flank
King danger	Cumulative bonus/penalty involving various king attack, defence, storm and shelter components
King mg	King-related bonus/penalty in the middlegame
King eg	King-related bonus/penalty in the endgame
Space	
Space Area	Number of safe squares on files c to f and ranks 2-4 for White or 5-7 for Black
Space	Weighted bonus for Space Area
Initiative (also Winnable)	
Initiative	Bonus/Penalty for initiative aspects (e.g. passed pawns, piece infiltration, flank majorities etc)
Initiative total mg	Weighted bonus/penalty for Initiative in the middlegame
Initiative total eg	Weighted bonus/penalty for Initiative in the endgame
Imbalance	
Imbalance	Weighted material value of pawns and pieces as an imbalance component for each side
Bishop pair	Bonus for possessing the pair of bishops
Imbalance total	Weighted bonus/penalty for material imbalance
Pieces	
Outpost	Number of outposts occupied by minor pieces
Outpost square	Number of outpost squares regardless of occupancy
Reachable outpost	Number of minor pieces that can occupy an outpost in the next move
Minor behind pawn	Number of minor pieces exactly behind a pawn
Bishop pawns	Bonus for pawns being on the same diagonal as a bishop

Rook on file	Bonus for rooks being on open and semi-open files
Trapped rook	Penalty for rooks trapped by their king
Weak queen	Penalty for pin or discovered attack on the queen
King protector	Bonus/Penalty for minor pieces based on the distance from their king
Long diagonal bishop	Bonus for bishops that are on a long diagonal and control both central squares
Outpost total	Bonus/penalty for outposts
Rook on queen file	Bonus for rooks being on a file any queen is on
Bishop x-ray pawns	Number of enemy pawns x-ray attacked by bishops
Rook on king ring	Number of rooks x-ray attacking the enemy king ring
Queen infiltration	Bonus for queen occupying a weak square in the enemy side
Pieces mg	Piece-related bonus/penalty in the middlegame
Pieces eg	Piece-related bonus/penalty in the endgame
Pawns	
Isolated	Number of pawns not having any friendly pawns on adjacent files
Opposed	Number of pawns having enemy pawns on the same file
Phalanx	Number of pawns having a friendly pawn on same rank and adjacent file
Supported	Number of times pawns are supported by other pawns
Backward	Number of pawns that are behind friendly pawns on adjacent files and cannot advance safely
Doubled	Number of unsupported pawns in front of a friendly pawn on the same file
Connected	Number of Phalanx or Supported pawns
Connected bonus	Bonus/Penalty for pawn aspects (e.g. connected, opposed, blocked etc)
Weak unopposed pawn	Number of not Opposed pawns that are either Isolated or Backward
Weak lever	Number of not Supported pawns twice attacked by enemy pawns

Blocked	Bonus for blocked pawns on the ranks 5-6 for White or 2-3 for Black
Doubled isolated	Number of pawns that are Doubled and Isolated and are blocked by an Isolated enemy pawn
Pawns mg	Pawn-related bonus/penalty in the middlegame
Pawns eg	Pawn-related bonus/penalty in the endgame
Passed Pawns	
Candidate passed	Number of pawns that are passed or candidate passed
King proximity	Bonus/Penalty for king distance from enemy passed pawns
Passed block	Bonus/Penalty for Candidate passed pawns in enemy side regarding the ease of their advance.
Passed file	Bonus for Candidate passed pawns based on its file
Passed rank	Bonus for Candidate passed pawns based on its rank
Passed leverable	Number of Candidate passed pawns excluding the ones with no pawn lever
Passed mg	Passed pawn-related bonus/penalty in the middlegame
Passed eg	Passed pawn-related bonus/penalty in the endgame
Attack	
Knight attack	Number of squares attacked by friendly knights
Bishop x-ray attack	Number of squares attacked by friendly bishops, only including x-ray attacks through enemy queen
Rook x-ray attack	Number of squares attacked by friendly rooks, including x-ray attacks through enemy queen and friendly rook
Queen attack	Number of squares attacked by friendly queen
Pawn attack	Number of squares attacked by friendly pawns
King attack	Number of squares attacked by friendly king
Attack	Number of squares attacked by friendly pieces and pawns
Queen diagonal attack	Number of squares diagonally attacked by friendly queen
Pinned	Number of pinned pieces and pawns

Threats	
Safe pawn	Number of pawns not attacked or adequately defended
Threat safe pawn	Number of non-pawn enemy pieces attacked by Safe pawns
Weak enemies	Number of enemy pieces and pawns attacked and not adequately defended
Minor threat	Number of enemy pieces and pawns attacked by a minor piece
Rook threat	Number of enemy pieces and pawns attacked by a rook
Hanging	Number of enemy pieces and pawns not defended or non-pawn pieces attacked twice
King threat	Number of enemy pieces and pawns attacked by king
Pawn push threat	Number of pawns that can be pushed and attack a piece safely
Slider on queen	Number of squares where the enemy queen can be safely threatened by a bishop or rook
Knight on queen	Number of squares where the enemy queen can be safely threatened by a knight
Restricted	Number of squares that are made unsafe for enemy pieces
Weak queen protector	Number of Weak pieces that are only supported by the queen
Threats mg	Threat-related bonus/penalty in the middlegame
Threats eg	Threat-related bonus/penalty in the endgame

Table 2: Stockfish evaluation functions and helpers

Main evaluation	General cumulative evaluation
Middlegame evaluation	Cumulative evaluation for the middlegame
Endgame evaluation	Cumulative evaluation for the endgame
Scale factor	Scaling coefficient for the endgame evaluation
Phase	Weight value based on the non-pawn material on the board, indicating the phase of the game (opening, middlegame, endgame)
Tempo	Bonus for having the turn to move

SECTION 3

Neural Networks

Neural networks are structures that resemble the human brain in its principle of operation. This means that their functionality is to store experiential knowledge acquired from data in their environment through a learning process and use it to respond in similar circumstances. More specifically, they are mathematical tools that can imitate the behavior of a system, that is approximate a function, without having any source of information about it other than its input and output.

The fundamental building block of a brain is a nerve cell or neuron, which is a cell that acts as the transmission channel for electrical signals. Each neuron communicates with other neurons through an interconnection point called synapse, which is a structure where the electrical signal causes the generation of a chemical substance that stimulates the receivers in parts of the next neuron called dendrites. Dendrites are the point where every neuron is connected with all of its previous neurons and the receiver stimulation at them generates the electrical signal in the neuron. Signals may be amplified or reduced with respect to the strength of the synapse known as synaptic weight. The magnitude of accumulated electrical signal on a neuron decides if the neuron will be activated and in case it does, a voltage spike is triggered and transmitted to the next neuron through its synapses and so on. The capabilities of a brain rely on the parallel structure and high connectivity of its neurons. The ability to learn derives from the modification of the synaptic weights so, in essence, acquired knowledge is stored in the synaptic weights.

Neural networks are parallel distributed processors composed of a number of processing units that, as a reference to the brain, are also called neurons. A neuron is a mathematical model, implemented as a hardware or software structure, the main components of which are a set of synapses, an accumulator and an activation function [20]. A visual representation of a neuron model is shown in figure 3.

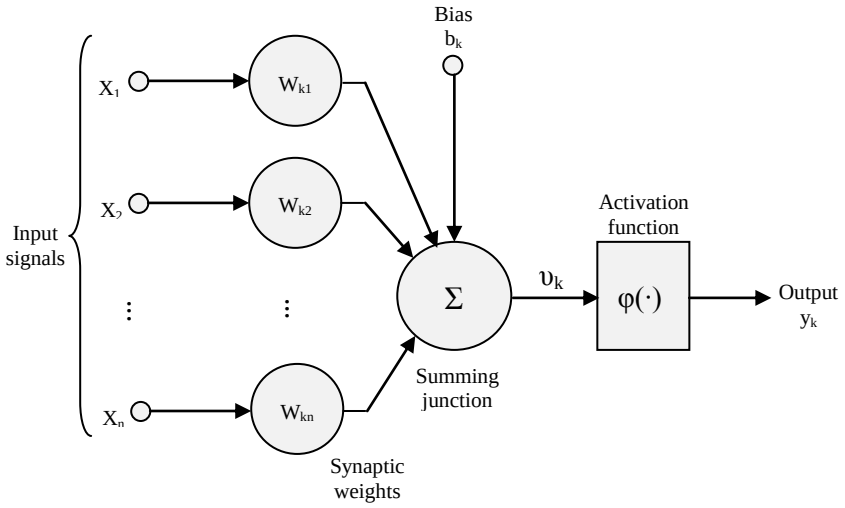


Figure 3: Neuron mathematical model

Each one of the synapses is actually a signal input x_j with an amplifier that represents the synaptic weight w_{kj}, with the index j referring to the individual synapse and the index k to the individual neuron. The weights can take positive as well as negative values. The accumulator, or summing junction, sums the weighted signals from all the synapses and also adds an optional external bias b_k which acts as an offset for the activation function, practically affecting the activation threshold. It should be noted that the indexing of the input signals x_j is one-based since the bias b_k can be considered as the signal at index zero and could as well be assigned a synaptic weight w_{k0} in order to be grouped along with the inputs. The activation function transforms the accumulated signal to a bounded output. More specifically, three types of function are most commonly used. One is the Threshold function, which evaluates to a specific value after a threshold t and to another before that. A usual case is the utilization of the Heaviside function or step function where the output values are 1 and 0 at a threshold of 0. Another type is the linear function which is similar except that two thresholds are defined, where under the lower one the output is 0 and over the higher one the output is one, while in between the output is a linear function of the input. The third and most frequently used type is the sigmoid function, an s-shaped function, an example of which being the logistic function described by equation 3.1:

$$\varphi(v) = \frac{1}{1 + e^{-\alpha v}} \tag{3.1}$$

where α is called the slope parameter. For very high values of α, the function approximates the Heaviside function but sigmoid functions are preferred because unlike the Heaviside they are differentiable.

The neuron model can be expressed by the equations 3.2 and 3.3:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{3.2}$$

$$y_k = \varphi(u_k + b_k) \tag{3.3}$$

where m stands for the amount of synapses of the neuron k, u_k is the sum of the weighted signals from the synapses, $\varphi(\cdot)$ is the activation function and y_k is the output signal.

3.1 Learning

The profound asset of a neural network is its ability to learn from its environment and improve its performance. This is achieved via a process called learning or training according to which the networks modify their free parameters, i.e. the synaptic weights and bias of its neurons, in order to adjust to the data provided by its environmental context. Learning or training algorithm is a chain of activities that solve this learning problem by computing the updated values of the parameters based on the data presented to the network.

There is a variety of ways the adjustment of the network parameters can be preformed, each of them defining a different type of learning. These different types dictate different rules during the learning process. The most notable types are briefly mentioned below.

- Error-correction learning is the type of learning where the synaptic weights of neurons are modified based on an error signal that is calculated as the difference between the desired output value and the value the network outputs for the given input with its current parameter configuration. The parameter modifications gradually bring the current output closer to the desired output. This evaluation is made actually not by minimizing the error signal itself but rather by minimizing what is called an index of performance (i.e. a cost function), for instance a function of the error signal named ‘instantaneous value of the error energy’ and defined as

$$E(n) = \frac{1}{2} e_k^2(n) \tag{3.4}$$

where $e_k(n)$ denotes the error signal of the neuron k for the nth vector of input.

- Memory-based learning is used for classification problems and makes use of training data, or previous results of operation in general, by storing correspondences of input data to desired output value in a memory. When evaluating a new input, the output is determined by the 'local neighborhood' of that input in the memory store by applying specific rules as the 'nearest neighbor' rule which is adopting the output value of the stored input with the minimum distance from the input under question. An improvement to this rule is the '*k*-nearest neighbor' rule where the *k* nearest stored data points are taken into account and the most frequent output value is adopted. This technique can eliminate errors due to outlier points, unlike the single 'nearest neighbor' rule.
- Hebbian Learning is named after the neuropsychologist Hebb who expressed a learning rule in 1949. Loosely explained the rule states that the weight of a synapse should be increased if the two neurons this synapse connects tend to be activated at the same time, while in the opposite case the weight should be lessened or even the synapse should be eliminated at all.
- Competitive Learning differs from other types of learning in that only one neuron is activated at a time, unlike other cases where many neurons of a network may be active, hence the term 'competitive'. Accordingly this type of strategy is named 'winner-takes-all'. This type of learning is suitable for detecting statistical features in the input dataset. An important prerequisite is that the neuron used must be identical in everything but their synaptic weights which have to be assigned randomly in order to behave differently for various inputs.
- Boltzmann Learning, named after Ludwig Boltzmann, is based on statistical mechanics and dictates a stochastic algorithm for learning. Neural networks trained by this type of algorithm are called Boltzmann machines. Neurons of such a network function in an on/off fashion and an energy function is defined that depends on the state and synaptic weights of each neuron. During the training process the state of randomly selected neurons is flipped with a probability that depends on the difference in the energy function value that this particular flip inflicts. Training is considered complete when the network reaches what is called 'thermal equilibrium'.

There are two major approaches to learning, also called learning paradigms, that are distinguished by the existence of a 'teacher', therefore referred to as 'learning with a teacher' or 'supervised learning' and 'learning without a teacher'. In the former case, an entity called teacher or supervisor has full knowledge of the environment, e.g. of a system that the neural networks attempts to emulate. This implies that for any possible input to the system, the teacher is able to provide the respective desired output to the network. In fact, what is actually provided to the network is the error signal, which categorizes this learning paradigm as a form of the error-correction learning type mentioned above. Taking the various input data and their respective error signals into account the parameters are adjusted in an iterative manner resulting in a state when the network can acceptably emulate the teacher rather than the actual system. In this final state, the knowledge possessed by the teacher is stored to the free parameters of the network. Such a learning approach aims to discover the relationship between the input and output data and is usually used for function approximation and pattern recognition problems. Figure 4 depicts a block diagram of the supervised learning process:

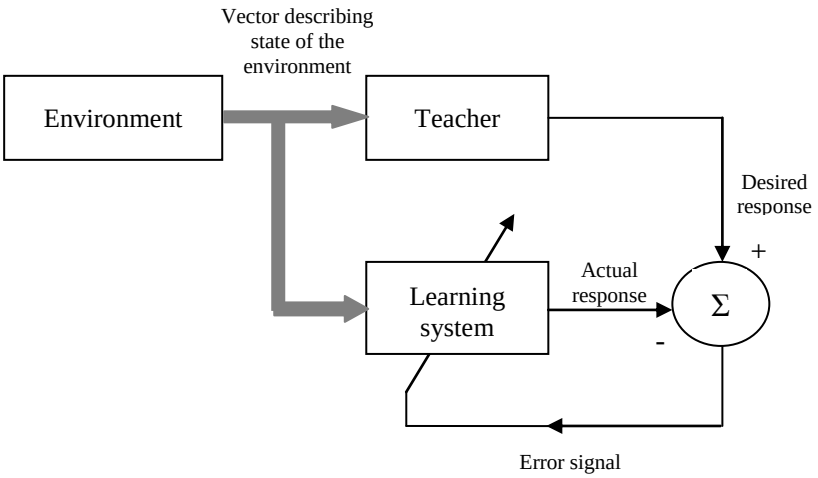


Figure 4: Supervised learning block diagram

The point where the training is considered as complete may be determined by statistical measures as the mean square error or the sum of squared errors. Such error functions can be expressed as multidimensional surfaces with the parameters of the network as coordinates. During the learning process, the appropriate parameter adjustments in order to successfully emulate the behavior of the teacher are translated to the operating point continuously moving towards a minimum point of the surface. This is accomplished by using the gradient of the surface which is a vector computed at any current operating point of the network and points towards the steepest descent. In this fashion the parameters are adjusted in order to reach a minimum point as soon as possible but without any guarantee that it will be a global minimum of the surface; it may as well be a local minimum.

The ‘learning without a teacher’ approach, as the name implies, does not include a teacher like in the supervised learning. In other words, there is no means of providing the desired output value to the neural network. This approach can be sub-categorized into two categories, ‘unsupervised learning’ and ‘reinforcement learning’. In unsupervised learning, the objective is not for the network to relate input to output data but to discover patterns among the input data. Training is based on maximizing an objective function or minimizing a loss function. Each neuron tries to adjust its parameters to describe input data most optimally with respect to that function. When the network functions on its own, after the training, usually the strategy ‘winner-takes-all’ is applied when the only activated neuron is the one that scores best based on the function the network was trained with for the given input. This kind of learning falls under the category of competitive learning and is used for clustering problems. In reinforcement learning, despite the fact that there is no teacher to provide the network with the desired values, a continuous interaction with the environment does exist through which the network is rewarded or punished for its actions. This is the role of a critic. After an interaction of the network with the environment some new input data are produced and the critic computes a measure of performance and transforms it into a reinforcement signal. This signal is fed to the network and causes synaptic weight configurations that produce good results to be positively reinforced and those with bad results to be negatively reinforced. This kind of learning is used for dynamic programming and adaptive automated control problems.

3.2 Architectures

A neuron can adjust its synaptic weights to store knowledge via a learning process. No matter how sophisticated this process is, a single neuron has limited capabilities. In order to have significant results in complicated problems, many neurons need to be put together so that the capabilities of learning and storing knowledge are greater, hence a network of neurons or neural network. In a network, neurons are organized in layers. In all types of networks there is an input layer of nodes that provides the input signals, which is not taken into account as an actual layer in any case since no computation is performed in it. The types of neurons used, the manner in which they are connected, as well as their arrangement are the factors that define the architecture of the network. This architecture determines the type of learning process that can be applied to the network. There are many different neural network architectures; some of them are specialized to the solution of specific types of problems while others are suitable for a variety of problems.

There three kinds of neural network architectures, distinguished by two main characteristics; the existence of hidden layers and the existence of feedback loops. The first type of networks includes these that consist of only one layer of neurons with no feedback loops. These are called single-layer feed-forward networks. In this type the input layer provides the signals to the only existent computational layer, which is the output layer of the network. As mentioned, in a feed-forward network the signal is only transmitted from the input to the output and no part of the output is cycled back as input. Secondly, there are networks that also not include a feedback loop in their structure but have more than one layer. Such a network is called a multilayer feed-forward network. The layers other than the input and output layer are called hidden layers and their function is to enable the network to compute statistics of higher order which is useful when the input layer becomes larger, i.e. the input variables are more. In other words, the existence of more synapses raises the learning capabilities of the network. When every neuron of the network is connected all neurons of the next layer, the network is characterized as fully connected. If this is not the case, the network is said to be partially connected. These networks are used in function approximation problems as well as in pattern recognition problems [21]. A visual representation of a fully connected multilayer feed-forward network is presented in figure 5.

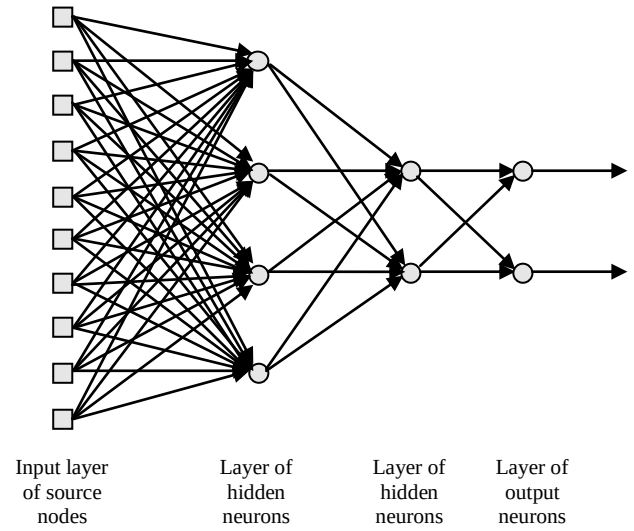


Figure 5: A fully connected multilayer feed-forward network

The third type of networks is the recurrent neural networks. In this case at least one feedback loop does exist and they could be either single-layer or multilayer. Also, feedback may or may not be provided by a neuron to itself (self-feedback) and hidden layers may also provide feedback and even do only that, without feeding any signal forward.

3.3 MLP Networks

One of the early implementations of a neural network was proposed in 1958 by Rosenblatt. His idea, the perceptron, is the plainest version of a neural network as it is only composed by a sole neuron which obviously classifies it under the single-layer feed-forward architecture. The proposed model was the first occurrence of a network to be trained according to the supervised learning paradigm. The structure of the neuron, as described earlier, includes a vector of inputs with synaptic weights and an additional bias. The activation function is a threshold function with upper and lower output values at +1 and -1 respectively and a threshold at 0, called the signum function. Since there are only two eligible output values, a perceptron can only be used for problems of classification between two classes.

However, there is one more constraint in using a perceptron for classification; the two target classes need to be linearly separable. This means that, in order for the classification to be effective, it is mandatory for a linear boundary to exist, separating the n-dimensional inputs on an n-dimensional space, i.e. a straight line in a two-dimensional plane, a plane in three-dimensional space and, in general, a hyperplane in n-dimensional space. This boundary, called decision boundary, is the means by which a perceptron can perform the classification. According to the general network structure, the output of the summing junction, and subsequently the input to the activation function, is expressed by equation 3.5.

$$u = \sum_{i=1}^{m} w_i x_i + b \qquad (3.5)$$

Since the activation function threshold is 0, the equation 3.6

$$\sum_{i=1}^{m} w_i x_i + b = 0 \qquad (3.6)$$

defines a hyperplane in an *m*-dimensional space which constitutes the decision boundary, where *m* denotes the dimensions of the input vector *x*. This boundary divides the input space in two regions that correspond to the two classes of the classification problem.

Viewing this fact from the training perspective, the synaptic weights and bias of the perceptron need to be defined in such a way that the resulting boundary is positioned properly in order to separate the training input data so that they correspond to the appropriate target classes. The learning process of the perceptron is of the error-correction type so the training algorithm uses the instantaneous value of error energy as a criterion in its iterative process of adjusting the synaptic weights. More specifically, the objective of minimizing this function is achieved via a learning rule called Widrow-Hoff rule or delta rule. This rule dictates that the modification of a synaptic weight performed on the nth iteration of the algorithm (i.e. the *n*th input vector) is expressed by equation 3.7.

$$\Delta w_{kj}(n) = \eta e_k(n) x_j(n) \qquad (3.7)$$

where η stands for the learning rate which is a positive value that defines in what proportion the product of the error signal and the respective component of the input vector *x* affects the modification of the corresponding synaptic weight. The updated weights are produced by adding the respective modification to the value the weight had in the previous iteration. It should be noted that the learning rate is a value that characterizes the learning process and remains the same across all modification calculations.

The external bias is incorporated into the set of the synaptic weights by considering it a weight itself that is always stimulated by an input of unity. In the beginning of training, the weights (and bias) are initialized with random value between 0 and 1. The training is terminated when the network current output value for all training inputs matches the desired output value for that input, essentially when the calculated modifications for all synaptic weights is zero. If the classes are indeed linearly separable, the training algorithm of the perceptron is proven to always converge according to the perceptron convergence theorem. It is worth mentioning that even though the perceptron is able to solve a classification problem with only two target classes, since it can produce only one decision boundary, the idea can be generalized and many perceptrons can be used in single-layer architecture in order to expand the capabilities of the network to solving classification problems involving more linearly separable classes by combining the decision boundaries each of the them can produce.

Nevertheless, many classes are inherently not linearly separable. Moreover, a whole other division of problems, function approximation dictates, in its generality, that the ability to emulate non-linear functions is necessary. This fact led to the development of neural networks of the multilayer feed-forward architecture, trained via supervised learning, where the existence of hidden layers provides such capabilities. The perceptron model is used for each neuron of these networks with a difference in the form of the activation function where the threshold function is substituted by a function that introduces a smooth non-linearity, along with the benefits of differentiability. In most cases, the sigmoid logistic function is used with unity as a slope parameter value.

Such networks constitute the most well known and commonly used forms of feed-forward neural networks; the multilayer perceptrons or MLPs. The main features of an MLP network, i.e. non-linearity, existence of hidden layers and high connectivity, along with the characterizing feature of every neural network, which is the ability of learning through training, are the source of the processing power it possesses. Interestingly enough, these features are the very reason why MLP networks are very complex to analyze on a theoretical basis and their learning processes very difficult to depict visually.

According to Kolmogorov's theorem, given enough neurons and the appropriate choice of synaptic weights, an MLP network with just one hidden layer is able to approximate any mathematical function. This fact led to characterizing the multilayer perceptrons as 'universal approximators' and subsequently means that the behavior of any unknown system can be emulated by an MLP. However, Kolmogorov's theorem only provides the aspect of mathematical theory and does not take into account aspects like the complexity of implementation or learning time optimization of the resulting network. Furthermore, there is no guarantee that a training algorithm capable of computing the values of the synaptic weights for approximating any unknown function actually exists.

Training an MLP utilizes the same core idea as the single perceptron. It is based on the delta rule and the minimization of a cumulative function of the error signal, which may be the generalization of the instantaneous value of the error energy of a neuron for all the neurons on the output layer, since an error signal can only be directly calculated on an output layer, called the instantaneous value of the total error energy defined by equation 3.8.

$$E(n) = \frac{1}{2} \sum_{k=1}^{K} e_k^2 (n) \qquad (3.8)$$

where K is the number of neurons in the output layer and $e_k(n)$ denotes the error signal of the neuron k for the nth input vector. Since the input and output of the network during training are the predefined input and output training data, this error function is actually a function only of the synaptic weights of the network. This implies that such a training process can be regarded as an optimization problem of minimizing this function by adjusting the values of the weights.

Assuming that every synaptic weight of the neurons in hidden layers was unmodifiable, or the hidden layers did not exist for that matter, and the output layer consisted only of a single neuron, the problem would be reduced to the training of a single perceptron and would be solved using the delta rule as described earlier. However, neither the output layer may necessarily include only one neuron nor the hidden layers would be absent in an MLP training problem. All the existing neurons, either output or hidden, contribute to the cumulative error function of the network. The arising problem is the discrimination between neurons that inflict a large portion of error to the outcome, and thus their synaptic weights should be greatly modified, and neurons that have already approached appropriate values for their weights. This is known as the credit assignment problem. An elegant solution to this problem and to the training of an MLP entirely is provided by the back propagation algorithm.

3.3.1 Back Propagation Algorithm

The back propagation algorithm, also referred to as error back propagation or back-prop, is a supervised form of learning that utilizes the error-correction rule. The main feature of the back propagation algorithm is that in order for the synaptic weights to be adjusted, two passes of signals through the network take place, one in the forward direction and one in the backward direction. In the former case, a training input vector is provided to the network so that an output according to the current synaptic weight configuration is produced. This is said to be the function signal as this is the way the network would normally work. At this point, no adjustments are made to the weights. In the latter case, this output along with the desired output defined in the training data, form the error signal which is propagated in the backward (output-to-input) direction through the network, and adjustments to the synaptic weights are performed in the error-correcting rule approach, i.e. the minimization of a cost function and more specifically the instantaneous value of the total error energy. Figure 6 dispays the direction of the signals.

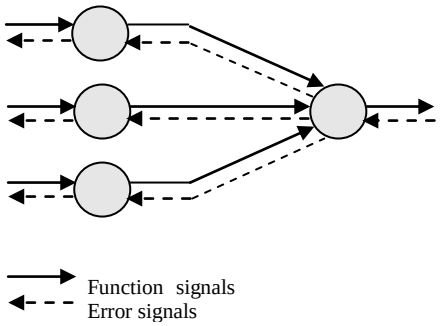


Figure 6: The directions of the input and error signals

These signal ‘roundtrips’ and weight adjustments are performed for each input vector in the training dataset. The processing of the whole training dataset in this fashion is called an epoch of training.

As mentioned, the synaptic weight modifications are calculated in an error-correction rule manner. The error function used as a measure of performance can be depicted as a surface in a multi-dimensional space with the various synaptic weights as coordinates. On this error surface the current configuration of the network defines the operating point and the synaptic weight modification is performed according to the gradient of the error surface, which

indicates the rate of change of the error function. The correction rule is the steepest descent of the error function so, in essence, the weights must be modified in such a way that the gradient is negatively maximum. This is expressed by the partial derivative of the error function with respect to the respective synaptic weight, which is the rate of change of the error function when the specific weight is modified. Combining these, a generalized form of the delta rule can be derived, defined by equation 3.9.

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} \quad (3.9)$$

where $w_{ji}(n)$ is the weight of the synapses connecting the *i* and *j* neurons for the *n*th vector of input, η is the learning rate and the minus sign indicates the descent of the gradient.

By applying the chain rule of calculus on the partial derivative, it is proven that the delta rule is expressed by equation 3.10.

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \quad (3.10)$$

where $\delta_j(n)$ denotes the local gradient for the neuron *j* and $y_i(n)$ is the output signal of the previous neuron or, equivalently, the input signal of neuron *j*. This expression of the rule shows that the weight modification is proportionate to the local gradient for the neuron *j*. At this point, two specific cases are considered for the neuron *j* according to its location in the network; if it belongs on the output layer or if it belongs to a hidden layer. In the former case, equation 3.11 defines the local gradient.

$$\delta_j(n) = e_j(n) \varphi_j'\left(u_j(n)\right) \quad (3.11)$$

This equation shows that the local gradient of an output neuron is equal to the product of the error signal and the derivative of the activation function of that neuron. Therefore, it is the calculation of the error signal at the output of the neuron that leads to the computation of the adjustments of the synaptic weights. For the case at hand the calculation of the error signal is trivial, since the neuron is provided with the desired output by the training data, and subsequently the computation of the local gradient and the weight modifications is also straightforward.

In the case of a hidden layer neuron, computations are more complicated due to the credit assignment problem. The aforementioned expression of the delta rule remains valid but the computation of the local gradient needs to also take the credit assignment into account. It is proven that the local gradient of the hidden neuron *j* is defined as follows.

$$\delta_j(n) = \varphi_j'\left(u_j(n)\right) \sum_k \delta_k(n) w_{kj}(n) \quad (3.12)$$

where *k* denotes a connected neuron in the next layer, $\delta_k(n)$ is the local gradient of the k neuron and $w_{kj}(n)$ are the synaptic weights involved in this connection. Comparing equation 3.12 to 3.11 (the respective local gradient of the output neuron), two observations can be made. In both cases the local gradient depends on the derivative of the activation function, which points out the structural contribution of any neuron to this calculation. Secondly, in terms of computation factors, it is the error signal of a hidden neuron that is actually expressed as a function of computations regarding not the neuron itself but the neurons in the following layer. What equation 3.12 demonstrates is that the local gradient of a hidden neuron is a function of the weighted sum of the local gradients of the next layer neurons, either hidden or output, that the examined neuron is connected with. This highlights the back-propagating nature of the algorithm, since computation results regarding the local gradients of neurons located closer to the output layer are prerequisites for those located further from it. It is this approach, in fact, that solves the credit assignment problem since the weight

adjustments of a neuron depend on the already adjusted synaptic weights of neurons that it affects.

Training is performed in a sequence of epochs, which is the processing of the entire training dataset as mentioned earlier. Usually the procedure includes presenting the network with the components of the training dataset, i.e. the input vectors, in a random order in each epoch in order to make the search of the synaptic weights more stochastic. The presentation of input vectors during an epoch and, more importantly, the weight adjustments can actually be executed in two methods; sequentially or batch. The sequential presentation of inputs, also called online training, has already been implicitly described. The input vectors are processed one by one, all the aforementioned computations are performed and the weights and biases are adjusted before moving on to the next. After all the vectors in the training dataset have been processed, training continues to the next epoch if necessary. It should be noted that whatever manner of presentation is used is maintained throughout the training. In other words, the manner of presentation characterizes the training itself and cannot differ across the epochs.

On the contrary, in batch training the weights are modified only once for all the available training data, at the end of the epoch. For this purpose, a cost function that uses the whole training set is needed, rather than one component vector of it. Such a function is the average square error defined by equation 3.13.

$$E_{av} = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in C} e_j^2(n) \qquad (3.13)$$

where the instantaneous value of the error energy is summed for all neurons in the output layer and all input vectors in the training set, and averaged by the amount N of these input vectors. According to this definition the modification of the synaptic weights take the form of equation 3.14.

$$\Delta w_{ji} = -\frac{\eta}{N}\sum_{n=1}^{N} e_j(n)\frac{\partial e_j(n)}{\partial w_{ji}} \qquad (3.14)$$

This is the adjustment made to the synaptic weight connecting the neurons *i* and *j* at the end of each epoch, with a learning rate η. Both methods have their merits but ultimately, in practice, the sequential method is more commonly used because its implementation is simpler and has proven to produce effective solutions to complex problems.

In a theoretical level, the back propagation algorithm does not converge in its generality so there are no established theoretical criteria for a terminating condition. Some practical criteria could be the stabilization of the synaptic weights, a maximum number of epochs or the convergence of the rate of change of the index of performance, e.g. the error function, under an acceptable threshold, which means that from a point forward the training epoch does not considerably improve the error so there is no point in continuing. It is possible though that after fulfilling a termination condition, the weight configuration not to be optimal. This is due to the nature of the steepest descent method that could trap the algorithm to a local minimum of the error surface because if such a point is reached no weight adjustments that move the operation point to a higher error level would be allowed. This implies that finding the global minimum of the error surface largely depends on the initial position of the operation point, i.e. the initial values of the synaptic weights, which are randomly selected.

The back propagation algorithm provides an efficient way of training multilayer perceptrons, regarding the computational complexity aspect. Even though it is does not guarantee the best solution for any possible problem where an MLP is used (as a universal approximator), it has managed to ease the major concern regarding the high complexity of the MLP networks.

3.3.2 Levenberg-Marquardt Algorithm

A variant of the back propagation algorithm would be the Levenberg-Marquardt algorithm, which is an algorithm for solving multivariate non-linear least square problems, and the most popular algorithm for training MLP networks [22]. It was developed by Kenneth Levenberg in 1944 and again, independently, by Donald Marquardt in 1963, bearing both their names and it is generally used in software tools for curve fitting problems. Matlab in particular used an improved version of the Levenberg-Marquardt algorithm, enhanced with a search algorithm for faster convergence [23].

The algorithm combines the steepest descent method with the Gauss-Newton method in an adaptive fashion. When the currently selected parameters have not yet approached the region of the optimal solution the algorithm acts like the steepest descent method, moving slowly but steadily towards the right direction. On the contrary, when the parameter selection does approach the desired values the algorithm behaves like the Gauss-Newton method which is faster and more accurate in the region near the optimal solution. This combination of methods is faster to converge that if these methods were individually used.

The decision of whether the current solution is far or close to the region of the optimal solution in this regard is made based on a non-negative algorithmic factor λ, which is called the damping parameter. Large values of the damping parameter results in a gradient descent operation while smaller values lead to Gauss-Newton. The parameter is initially set to a large value in order to made steady steps towards the optimal solution. As the current solution improves the λ parameter is decreased. If a specific adjustment results in a worst solution, the λ parameter is again increased [24].

3.4 RBF Networks

An alternative feed-forward neural network implementation was proposed by Broomhead and Lowe in 1988. This implementation utilizes radial basis functions as activation functions and the resulting networks are hence called radial basis function, or RBF, networks. RBF networks are used for solving problems of function approximation, classification and system control.

A radial basis function is a real function, the values of which depend only on a distance metric, typically the Euclidian distance, between the function input and a fixed point in the respective space, called center. In other words, radial basis functions are functions with radial symmetry. In more formal terms, a radial basis function is any function that satisfies the equation 3.15, where *c* denotes the center:

$$\varphi(x) = \varphi(\|x - c\|) \tag{3.15}$$

In some cases radial basis functions are used as groups to compose a basis in a vector space, i.e. a set of linearly independent functions that can produce any element of the space uniquely as a finite linear combination of them.

In a function approximation problem using radial basis functions, the computed function is a sum of radial basis functions, each having its own center and its contribution to the sum varies according to a weight coefficient. It is proven that by summing enough radial basis functions, it is possible to approximate any function. Equation 3.16 presents the general form of such a computation:

$$y(x) = \sum_{i=1}^{N} w_i \varphi(\|x - x_i\|) \tag{3.16}$$

where x_i is the center of the i-th function and w_i is the respective weight. This function form is what an RBF network structure actually implements.

RBF networks have a structural distinctiveness in comparison to the general feed-forward network model, which is that they only have a single hidden layer of neurons and also the output layers consists only of summing junctions. In addition to this, and as already mentioned, the activation function of each neuron in the hidden layer is a radial basis function, most commonly a Gaussian function. Since any function can be approximated if enough radial basis functions are summed, RBF networks are, like MLPs, universal approximators given the appropriate amount of neurons. A visual representation of an RBF network with *N* input variables, *M* hidden nodes and three output variables is presented in figure 7.

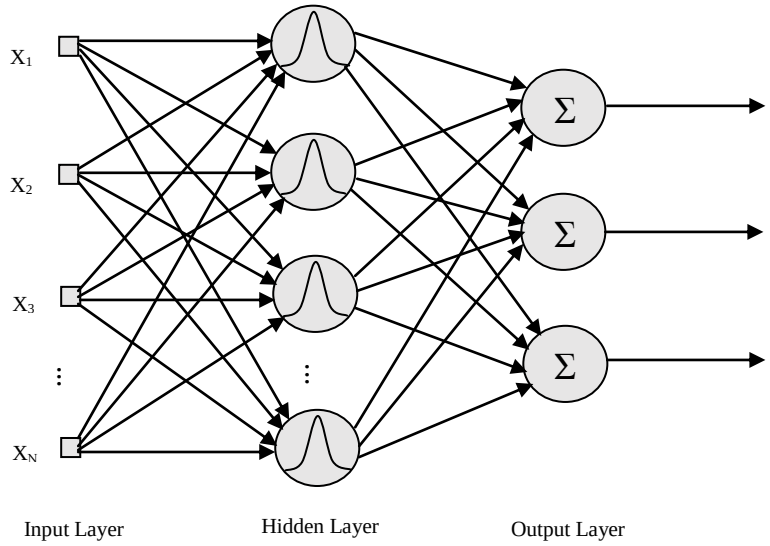


Figure 7: An RBF network representation

The existence of a radial basis function implies another difference between RBFs and typical feed-forward networks which is that the knowledge in an RBF is stored not only in the synaptic weights of neurons but also in the parameters of the activation function, i.e. the radial basis function centers and in case of variable width functions, like Gaussian functions, the curve width. As a result, this fact needs to be incorporated in the network training process. In other words, training an RBF network consists of computing not only the synaptic weights but also the coordinates of the centers, the width of the curves, if needed, and also the number of neurons on the hidden layer [25].

This process could be performed in an MLP way, using a method like steepest descent. However, the particular nature of RBF networks allows for an alternative approach, specifically suited for its structure. Training of an RBF can be broken down into two phases: the computation of the centers and widths of the radial basis functions and the computation of the synaptic weights. , Some training algorithms also include the computation of the optimal hidden layer size in their process. For those that do not, the size is manually determined and then can be optimized by repeating the training in trial-and error fashion.

Computing the function centers and widths is the more challenging of the two phases. It is usually handled as a clustering problem, thus a clustering algorithm is utilized and depends only on the input data. Such an algorithm is the *k*-means, an unsupervised clustering algorithm which was actually the popular approach in studying RBF networks in their early days. Nevertheless, this algorithm has two major disadvantages that led to the proposal of many alternatives in the following years. These are the slow algorithm convergence, being an

iterative process, and the fact that the aforementioned trial-and-error is needed to determine the number of neurons on the hidden layer.

After determining the hidden layer size and the function centers and widths in the first phase, the second phase is a much simpler task. Since the desired output is a weighted sum of the single hidden layer output, or in other words there is linear relationship between the hidden layer output and the network output, the synaptic weights can easily be calculated by the use of linear regression. A typical approach is using the linear least squares method, in matrix form, as displayed by equation 3.17.

$$w^T = Y^T \cdot Z \cdot (Z^T \cdot Z)^{-1} \qquad (3.17)$$

where Z denotes a matrix that contains the outputs of the hidden layer for every input and Y denotes a vector that contains the desired output, i.e. the target values.

3.4.1 Fuzzy Means Algorithm

One popular training algorithm for RBF networks is the fuzzy means (FM) algorithm [26]. This algorithm improves upon the *k*-means algorithm in that it incorporates the determination of the hidden layer size and demands much less execution time since it is not iterative regarding the input data. The main idea of the algorithm is the partition of each dimension of the input into triangular fuzzy sets, hence creating a grid of multidimensional subspaces in the input space, and then the selection of certain nodes of the grid as radial basis function centers for the hidden layer of the network based on a specific criterion.

More specifically, considering a case where the RBF network to be trained accepts N input variables, or in other words the input space has N dimensions, every dimension is partitioned into the same number M of triangular fuzzy sets. Given that the input is normalized, these sets also have the same width and can generally be described as follows.

$$A_m = \{a_m, \delta a\}, \qquad \text{s} = 1, ..., \text{M} \qquad (3.18)$$

where A_m denotes the m-th fuzzy set, α_m denotes the center element of the fuzzy set A_m and $\delta\alpha$ is the half of the width of a fuzzy set. Figure 8 depicts such a partition, using five fuzzy sets and considering a two dimensional input for the sake of visualization.

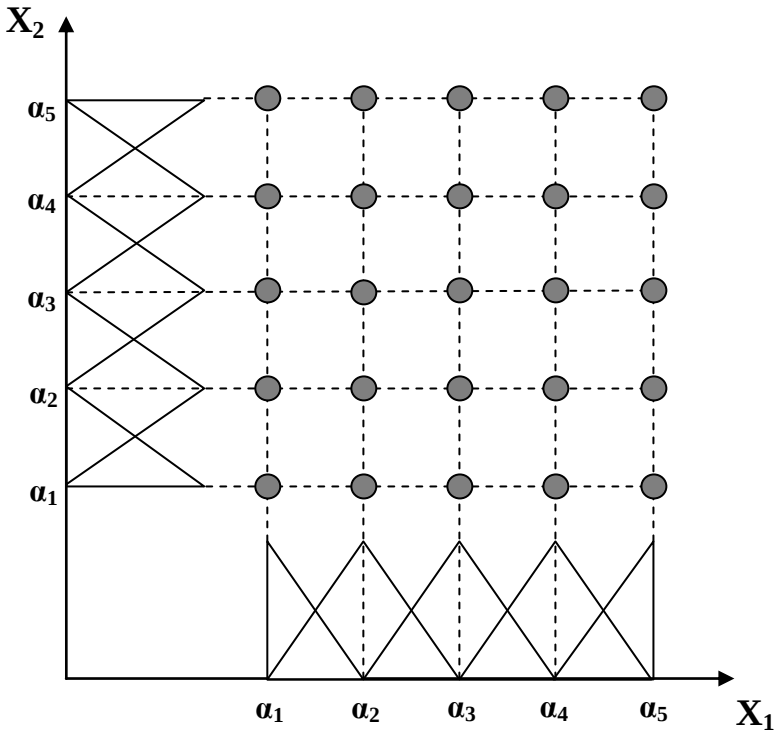


Figure 8: A fuzzy partition on a two dimensional space

Partitioning the input space in this manner produces *S* multidimensional fuzzy subspaces, through the combination of all the fuzzy sets in every dimension, equal to the amount of the used fuzzy sets to the power of the input dimensions. Each fuzzy subspace is denoted as A^s, where $s = 1, ..., S$, and its center element a^s is defined as a vector containing the center elements of the respective fuzzy set in each dimension. In the above particular scenario, 5 fuzzy sets are used for each of the two input dimensions so 25 subspaces are produced. These subspaces are the possible RBF centers for neurons on the hidden layer. An appropriate selection of these subspaces as RBF centers is the goal of the FM algorithm in order to construct the hidden layer in such a way that the network is both as minimal as possible and suited for the input data provided to it.

The criterion based on which the algorithm determines the selection of the centers is called a membership function and indicates to what degree each specific input vector belongs to a fuzzy subspace A^s. Equation 3.19 presents a general definition of the membership function.

$$\mu_{A^s}\big(x(k)\big) = \begin{cases} 1 - d_r^s\big(x(k)\big), & if\ d_r^s\big(x(k)\big) \le 1 \\ 0, & otherwise \end{cases} \qquad (3.19)$$

where $d_r^s(x(k))$ is a distance function between the input data vector *x(k)* and the fuzzy subspace A^s. This distance function is represented as a hyper-surface on the *N*-dimensional space of the input and constitutes a means of discriminating input vectors that get any degree of membership to a fuzzy subspace A^s from other vectors that do not. Since every dimension of the input space is partitioned using the same number of fuzzy sets (symmetric partition), the distance function hyper-surface is actually a hyper-sphere which can be defined by equation 3.20.

$$d_r^s\big(x(k)\big) = \sqrt{\frac{\sum_{i=1}^{N}\big(\alpha_i^s - x_i(k)\big)}{\sqrt{N}\delta\alpha}} \qquad (3.20)$$

where N denotes the dimensionality of the input space, α^s_i is the component of the fuzzy subspace center element in the i-th dimension (i.e. the respective fuzzy set center element) and $x_i(k)$ is the input vector component in the i-th dimension of the k-th input vector. Based on the above, the algorithm processes the input data and determines a selection of the fuzzy subspaces so that every input vector receives nonzero membership in at least one of them. As mentioned, this is a non-iterative procedure as the input dataset is processed only once and so short computational times are achieved even in cases of large datasets.

More specifically, for every vector in the input dataset the algorithm checks if it is located outside from all the hyper-spheres already defined by the previously selected RBF centers (of course none exist during the processing of the first vector). If this condition is satisfied, a new RBF center necessary in order to cover the current vector, otherwise it has already been covered by previous centers and so no more action is needed. This logic ensures that no input vector can be left uncovered. The algorithm then calculates the value of the membership function for every fuzzy set in every dimension for the current vector and the sets scoring the maximum value in each dimension assemble the fuzzy subspace that fits it the best, thus defining the new RBF center.

After the processing of the whole input dataset, a selection of RBF centers that compose the hidden layer of the network has been constructed. This selection of centers, as the description of the algorithm process shows, only has one parameter that affects its final form, which is the number of fuzzy sets that all input dimensions are partitioned by. This implies that an optimization process can easily be applied in order to produce an RBF network with optimal configuration.

SECTION 4

Experiment, Results and Discussion

As mentioned in the introduction, the objective of this Thesis is to investigate an approach in evaluating chess positions without performing a deep tree search look-ahead as a chess engine does. Instead, we attempt to achieve this by the utilization of the learning capabilities of a neural network. Both the architectures of MLP and RBF networks have been tested in order to compare their performance on the task at hand.

The goal of the experiment is to train a neural network in order to predict the evaluation score of a chess position at a high depth, which would take a chess engine a greater amount of time to calculate. This prediction is based on our proposed set of static features of the position as well as three low depth evaluation variables, at different depths. The existence of these evaluation variables in the training dataset provides the network with some dynamic knowledge of the position. Networks have been trained with three variants of the training dataset, regarding these evaluation variables, in order to examine their impact on the final outcome. The different variants were constructed by including all evaluation variables, by including only the two lesser-depth ones and by including none of them, respectively.

4.1 Experiment Implementation

A prerequisite for the network training is of course the creation of the training dataset. The output of the network, as already mentioned, will be the position evaluation score and in order to complete the formation of the training dataset, the input vector needs to be determined. Two conceptually different groups of input variables were chosen. The first group consists of our proposed set of features of a chess position as described later in this section. These are static features that describe the position at its current state and indeed the only required knowledge for deriving them is the current arrangement on the board, in the form of a FEN most probably. The second group of inputs refers to the dynamic nature of the position. These are actually evaluations of the position from the chess engine but at much less depth of search than the evaluation score we are trying to predict. These values act as a supplement to the feature inputs so that the temporary circumstances on the board, like the absence of a piece in a position occurring in-between a piece exchange sequence, are not misinterpreted by the network as equivalent to positions were such circumstances are permanent, e.g. a position with actual material imbalance, as would the sole consideration of the static features indicate.

The creation of such a dataset required the implementation of a processing application that would extract the necessary information from a large number of chess positions and store it in a database in order to be used in the procedure that would execute the training of the neural network. Such an application was developed using the Java programming language and the database was created and managed, via select and insert queries in the Java code, using SQL. The outline of the operation of the application is the following:

- A collection of chess games provided to the application in the form of PGN files are processed one by one.

- Each PGN file is processed in order to extract the move sequence of the particular game.
- An internal board representation is created at the initial position and the specific moves are performed on the board one by one.
- The current position FEN is checked for existence in the database. If it is found, the process proceeds with the next move.
- The features are computed for the current position.
- The phase of the current position is checked. If it is not middlegame, the position is ignored and the process proceeds with the next move.
- The current position is analyzed by a chess engine and the evaluation scores, both those used as supplementary inputs and the actual target, are calculated.
- The features and evaluation scores along with the corresponding FEN of the current position are stored in the database.
- Arising positions are processed sequentially in this manner until the current game is over and the procedure is repeated until every PGN has been processed.

The operation of the game processing application is displayed as a flow diagram in figure 9.

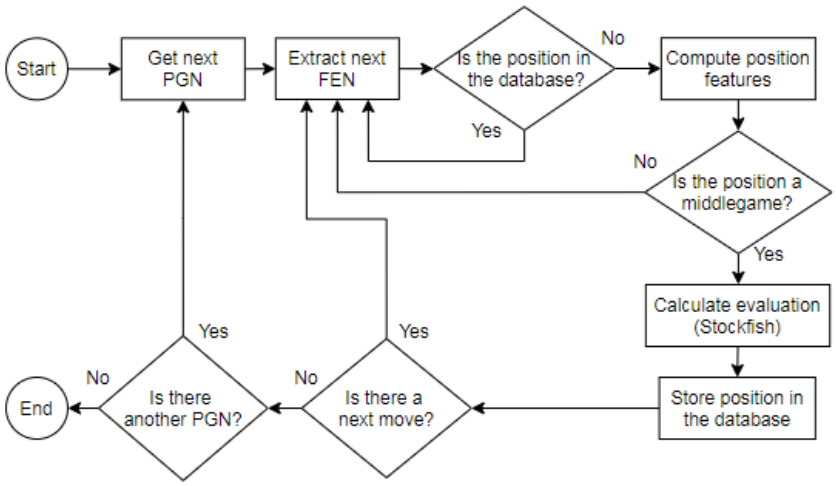


Figure 9: Game processing application flow diagram

Elaborating a bit more on the aforementioned procedure, a large collection of games of top-players (1514 in number) was retrieved from online chess databases, like chessgames.com and 365chess.com, and was provided to the application. It should be noted that this number of games was not collected and processed in a single run but rather in an iterative process were the database increased incrementally and networks were trained using the respective datasets, gradually improving performance.

As mentioned, the chess engine that was used was the open-source Stockfish engine, version 10. This led to the decision of deriving our proposed set of position features used as training inputs to the neural network from the features the Stockfish relies on. In fact, as was briefly described in section 2, what Stockfish considers as a single feature may actually be composed of some other sub-calculations of simpler patterns. Moreover, while these simpler patterns are calculated separately for the black and white side, it is in most cases the difference between the two respective values that is used in the calculation. In addition to these, as displayed in table 2, Stockfish calculates a few intermediate, cumulative values and helper values that finally combine, along with the features, into one master value (referred to as ‘main

evaluation') that is actually the static evaluation of the position. Every calculated value involved in this process, regardless of being simple or complex, a meaningful feature or an intermediate value, is described in the online documentation of the engine and was also briefly presented, in section 2. The documentation provides snippets of code in the Javascript programming language about each one of them which, although not directly usable in our case, have provided an insight about the manner each particular value is calculated and from this knowledge came the implementation in Java code for our processing application.

Regarding the network training dataset, our approach in constructing our proposed set of features, even though many are just linear combinations of others, was to consider the values of every feature and function described in tables 1 and 2, as well as the respective values for either side, as a separate position feature. This aims to maximize the possibility of the neural network discovering patterns in the feature dataset that relate to the resulting evaluation that may be overlooked in case of only using the aggregate values.This resulted in a total of 194 input values.

Another important decision that has been made is to perform the training of the network with data extracted only from chess middlegame positions. This is because the middlegame is the most complicated phase of a chess game where strategy is formed, there are usually not all but also not too few pieces on the board and therefore the network has the best opportunity to detect patterns that emerge from the features describing these positions and associate them with the evaluation. Apart from that, the other two phases of the game, the opening and the endgame, are usually handled by the engine using other means in addition to the tree search. In the former case the vast opening theory that already exists in chess literature is utilized by the engines with what are called 'opening books' that may guide the search accordingly. In the latter case, the endgame, there is a very good chance that no tree search is performed at all and an endgame tablebase is typically used instead. An endgame table base is an enormous database that contains every possible chess endgame position from a point on, already analyzed to the end, providing the outcome, the number of moves to reach it with best play in cases of a win or loss, and the best move in the position. As from the year 2012, tablebases contain every endgame position that includes up to seven pieces [27]. In this sense, the concept of an evaluation score is no longer valid as it is delegated to a value of zero in case of a draw, or the number of moves to reach a win or loss in the respective cases.

Stockfish, and hence our processing application, distinguishes among the different phases of the game using the phase parameter which is also used in the main evaluation formula. The phase parameter depends on the non-pawn material on the board and takes values from 128 to 0, where 128 corresponds to the opening, 0 to the endgame and any value in between to the middlegame. As mentioned, a position is analyzed by the engine and ultimately stored in the database only in case it is a middlegame position.

After a position has been identified as a middlegame position, it is analyzed by the chess engine so that the supplementary low depth evaluations as well as the target evaluation score can be extracted. Stockfish is allowed to analyze a position for exactly 165 seconds (2.75 minutes) and reach as much depth as possible in this timeframe. Of course, since the time is fixed, the achieved depth depends on the processing power of the computer the engine runs on, which in this case uses an Intel Core i7-4779 quad-core at 3.40 GHz and 8GB of RAM, and the configuration of the engine itself. The important points of the configuration in our case, set using the 'setoption' command as described in section 2, include occupying 3 computational threads per engine instance, a hash table of 1024MB and using analysis mode with a multiple PV of two, which means analyzing the two best lines instead of one in order to prevent the engine from possibly overlooking a better branch in the search tree due to its default behavior of applying more pruning on the non-best branches.

Under these circumstances, and depending on the complexity of each position, the engine reaches at a depth of around 28 plies, i.e. half-moves (individual moves played by either side), as opposed to what chess defines as a full move which consists of two plies, one by White and one by Black. Regarding the evaluations used as inputs, three scores were decided to be used, at depths of 2, 4 and 8 plies, bringing the total length of the neural network input vector to 197. For these depth values the chess engine on the particular machine can provide evaluations in a matter of a few milliseconds, while the processing time needed to extract the 194 static features mentioned before is about 10 seconds. The time needed for the two types of database queries, for checking if a position already exists and storing it, take less than 10 milliseconds each. These bring the whole procedure of processing a position to a total of about 3 minutes. In order to produce more data in the same amount of time, two instances of the processing application where run in parallel (in different CPU threads), each one provided with its own (non-overlapping) set of games to process and also running its own instance of Stockfish, but interacting with the same table in the database which was the single point of data storage. The table would then be exported as a csv file whenever the data were needed for network training.

The creation and training of the MLP networks has been facilitated by using the Matlab software and specifically the neural toolbox that provides implementations of various architectures along with an API for training and using the network for predictions. On the contrary, RBF networks were created and trained by using custom Matlab scripts that were written for this purpose. The training data were imported from the csv file as a matrix which was manipulated accordingly to support the training of the network. Driver scripts were created in order to support the preparation of the dataset, the definition of the networks, the execution of the training procedure and the usage of the networks for making predictions and measuring indicators for the various cases. The feed-forward MLP network that was trained in all cases had a two-hidden-layer structure, the first containing 20 neurons and the second containing 10 neurons, determined after a trial-and-error process. A visual representation of the network, as provided by Matlab, is shown in figure 10.

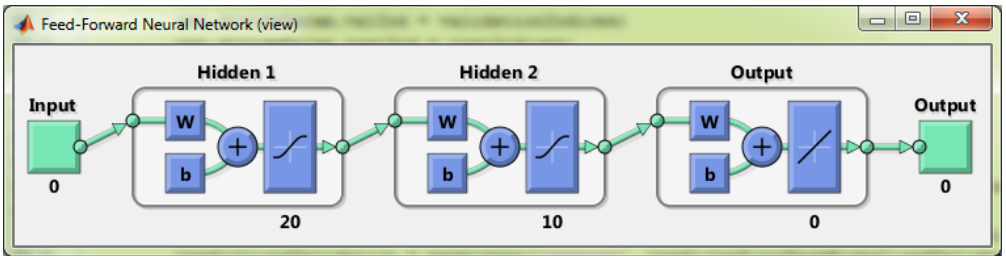


Figure 10: The trained MLP neural network

For the RBF network experiment we used thin plate spline (TPS) functions as activation functions and the network training was performed via the fuzzy means algorithm, also with the help of custom Matlab scripts. Equation 4.1 defines the thin plate spline function in RBF context.

$$\varphi(r) = r^2 log(r) \qquad (4.1)$$

As already mentioned, when a chess engine evaluates a position, there are two possible types of outcome, an evaluation score or the number of moves until mate, if such a sequence is found. While the processing of only middlegame positions lowers the possibility of the latter case, positions with mating sequences may still occur. In order for these cases to be distinguished, and not confuse a result of 2 meaning ‘mate in 2’ with an evaluation of +2,

before storing to the database, a very large value was assigned to such positions as evaluation score for the sake of uniformity. These data were then filtered out after importing the csv file to Matlab and were not part of the final training dataset since they do not correspond to an actual evaluation score. The resulting dataset in this case consists of 80425 middlegame positions.

In a second level of refinement, such filtering was alternatively performed for data having an evaluation score higher than 20. In a typical chess game, evaluation scores tend not to be very high since the game is usually more or less balanced. Values over 6, indicating an imbalance of two minor pieces, or even 9 that corresponds to the value of a queen are sparser than those closer to zero and the reason for this is that when the game is headed towards such circumstances, the losing player often resigns and these positions do not occur. This fact leads to the dataset containing much more data in the region nearer to zero and less as values get higher. The aforementioned filtering on values over 20 intends to examine the impact of this situation on the training of the neural network. The resulting dataset in this case consists of 79874 middlegame positions.

In addition to these, one supplementary experiment has been included. We have reproduced the methodology found in one of the mentioned related researches [9] using our own input data, in order to compare performances. This research presents two alternative board representations, referred to as 'bitmap' and 'algebraic', and uses these with an MLP, among others. We have taken their best performing case, the bitmap representation, and using our database of games, we produced the input dataset as described in their paper. Two MLP networks were trained accordingly, following the same filtering logic as outlined above.

Specifically, the bitmap representation analyzes the board in different layers, each of them regarding a specific piece. There are 6 different types of pieces for each side hence 12 layers. Each layer consists of 64 inputs that represent the state of the 64 squares of the board respectively. The value of an input may be 0 indicating that no piece is currently occupying the respective square, 1 if the square is occupied by a white piece of the type corresponding to the layer and -1 if it is occupied by a black piece of the type corresponding to the layer. This representation results to an array of inputs that contains 768 values.

4.2 Results & Discussion

In order to evaluate the performance of the networks, two indicators were chosen. The first is the mean absolute error (MAE) of the predicted position evaluation scores compared to the training targets, i.e. the error of the prediction, as defined by equation 4.2:

$$MAE = \frac{\sum_{i=1}^{n} |e_i|}{n} \quad (4.2)$$

The second indicator is the coefficient of determination (R^2) which is defined as follows:

$$R^2 = 1 - \frac{SSE}{SST}, SSE = \sum_{i=1}^{N} (y_i - y_i^*)^2, SST = \sum_{i=1}^{N} (y_i - y_i^-)^2 \quad (4.3)$$

where y_i is the actual target, y_i^* is the corresponding prediction and y_i^- is the mean value of the actual targets.

Regarding the MLP networks these indicators were measured as mean values over various iterations of network training and predictions. In these iterations a different randomization is applied on initializing the synaptic weights and biases of the network, possibly starting their approximation from a better position in order to find a better local (or the global) minimum of the error surface and resulting in a better performing configuration. On the contrary, different randomization does not affect the training algorithm of the RBF networks, other than the

random permutation of the dataset which is kept the same in all cases anyway, since the synaptic weights and biases are calculated through linear regression rather that approximated. Consequently no iterations were executed while training RBF networks.

No strict rule exists regarding the splitting of the input dataset to training, validation and testing data so a decision for 50% - 25% - 25% ratio was made. It should be noted that the permutation of the input dataset, in order to make each of the mentioned sets as random as possible, was always the same, independently from any randomization mentioned above, so that the same arrangement of data was kept in all iterations thus avoiding discrepancies in performance due to differences in the datasets.

The results are presented in matrix form depicting the two different cases of dataset filtering that are described above and the different configuration regarding the low-depth evaluation inputs, which are denoted as scenario 1, 2 and 3. Specifically, scenario 1 includes all three of the evaluation inputs (for depths 8, 4 and 2 as previously described), scenario 2 includes only the variables for depths 4 and 2, and scenario 3 does not include any of these variables at all. Each MLP performance indicator value presented in the matrices is the best case value accompanied by the mean value with its standard deviation over the different iterations in parenthesis. In the case of the RBF networks, the results presented do not include any mean and standard deviation since no iterations were executed and are supplemented by additional measurements, such as the number of selected RBF centers and fuzzy sets, which specifically help evaluate the performance of the RBF network.

4.2.1 RBF networks with proposed feature inputs

The results of the RBF networks trained with the input dataset consisting of the 194 variables of the proposed set of features plus the 0 to 3 evaluation inputs according to each scenario, for both cases of dataset filtering, are presented in tables 3 and 4. These tables also include the computation time for network training for each scenario. It should be noted that this computation time is the time for training the best network after performing structure optimization, i.e. the training of several networks with different structures for the hidden layer, and selecting the best one.

Table 3: Indicators of RBFs using the proposed features with mating evaluation filtering

Scenario	MAE		R^2		Fuzzy Sets	No. of nodes	Time (s)
	Testing	Validation	Testing	Validation			
1	0.76	0.76	0.50	0.52	25	38219	7304
2	0.83	0.84	0.43	0.45	21	38229	12448
3	0.85	0.85	0.41	0.43	21	38238	5745

Table 4: Indicators of RBFs using the proposed features with over-20 evaluation filtering

Scenario	MAE		R^2		Fuzzy Sets	No. of nodes	Time (s)
	Testing	Validation	Testing	Validation			
1	0.38	0.38	0.78	0.80	19	37983	5344
2	0.44	0.43	0.72	0.73	23	37995	5809
3	0.45	0.44	0.69	0.70	25	37970	8638

We can see that within both filtering cases, the first scenario, that contains all three supplementary evaluation input variables always performs better than the rest, and also scenario 2 that contains only the two lower depth variables performs slightly better than scenario 3 that merely contains static features.

A graphic representation of target versus predicted values for the best performance in the test subset for the two dataset filtering cases can be seen in figures 11 and 12.

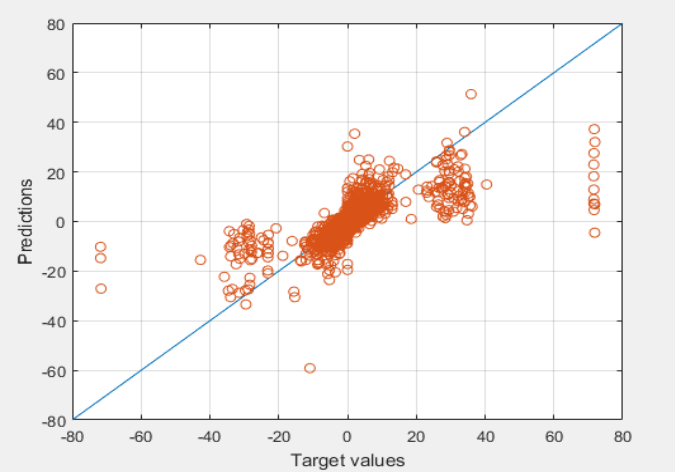


Figure 11: Target values vs Predictions of RBFs using the proposed features with mating evaluation filtering

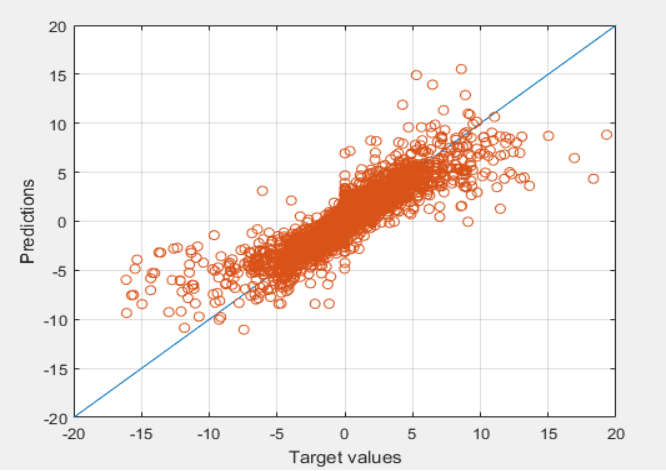


Figure 12: Target values vs Predictions of RBFs using the proposed features with over-20 evaluation filtering

Both figures depict the ideal network performance (blue line). The lack of uniformity in the input dataset can be easily observed in figure 11, where it can be seen that a big part of the data is gathered in the area around zero. Another observation at figure 12 is that as far as very large evaluation values are concerned, there seem to be some groups forming at value ranges of 20 to 40 and -20 to -40 and some extreme values around +-70. This phenomenon may be due to the fact that, when constructing the input data, the engine was given a fixed amount of time to process each position and such extraordinarily high evaluations can dramatically change when the engine reaches one more level of depth, or even if it happens to find an even better move sequence at the same depth. In other words, these may be cases where the engine has not managed to converge to an evaluation in the available amount of time. If more time was given to the engine, some of these evaluations would remain the same, some of those 30-like evaluations would spread to a larger range, and maybe a few of the evaluations at 70 could actually become 90 or 150. Of course it is not possible to determine the time frame the engine needs in order to converge, so possibilities for such phenomena, especially for irregular circumstances like evaluations of 30 or 70, will always exist. Other than these, the overall arrangement of the data points tends to follow the expected shape of the ideal model line, especially those of figure 12.

An observation in figure 11 regarding the performance of the network is that in the aforementioned cases where the target values are extremely large, almost every prediction is way off in terms of value approximation, possibly due to the inadequate quantity of data in these areas that results to the failing extrapolation the network attempts to perform. However, it is important to note that all predictions, except from two, are in the appropriate quadrant, meaning that winning side is still predicted correctly. Interpreting this fact in the context of playing the game, a position of actual evaluation of +30 that is mispredicted as +17 is still regarded as a clear win for the white side by the network.

Another interesting thing to notice, which exists in both diagrams but is not clearly visible due to the density of data, is the behavior of the network when the target value is exactly zero. A portion of the diagram displayed in figure 11, focused on the area around zero can be seen in figure 13.

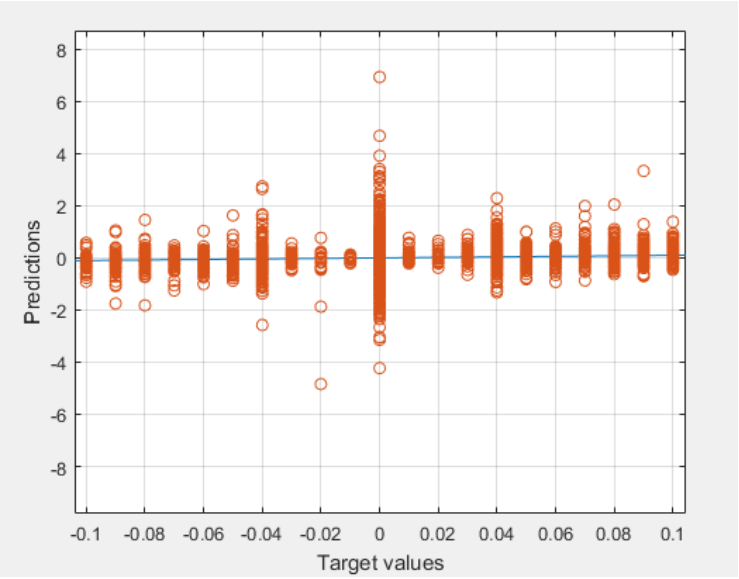


Figure 13: Network behavior around zero target value

As we can see, the network seems much more likely to make an erroneous prediction, ranging to large values for that matter, if the target value is exactly zero than in cases of values very close to zero.

As it has already been explained in section 2, an engine may give an arithmetic evaluation of a position, may detect a win or loss for the side to play (in which case it does not return an evaluation but the number of moves until mate), or it can detect that the position is a draw, which is indicated by returning the value of 0.00. Understanding what a draw indication means is actually a complicated matter, as there could be various reasons why that is, all grouped under the value of zero with no discrimination whatsoever. The most common cases for a draw are these of move repetition or perpetual check. These includes cases where there really is no other option in the position, like when there is insufficient material, or where the option of going for the repetition, or perpetual check is the best play for one or both sides, meaning that any other move sequence would result to a loss. Another case that is a bit trickier to comprehend is when the tree search reaches depths where the arising position can be found in an endgame tablebase. Endgame tablebases, mentioned in section 4, contain pre-analyzed positions containing up to six pieces, solved to the very end. This means that even if the current position is a middlegame position with many pieces on the board, when the tree search reaches a depth of 25 or so plies with best play, piece trades may have occurred so that the resulting position could be one of those pre-analyzed positions in the tablebase, already worked out to be a draw and consecutively evaluated as 0.00.

Many discussions have taken place in online forums about whether the value 0.00 is reserved only for special cases like forced draws or theoretical draws, or if such a value could also come up for positions that happen to be absolutely level and are truly evaluated to zero, without any clear conclusion on the matter. Others claim that the value of zero may also be interpreted not as absolute equality, but as the position being so complicated and unclear that the engine would need more time to search deeper in order to give a slight edge to either side.

However, from our neural network point of view, where no tree search is performed and inputs are related to evaluation values, this absolute zero evaluation is in a way a problematic situation. Specifically, assuming two similar looking positions where the one is a clear win for White, say +5, while the specific arrangement of pieces on the other allows a perpetual check,

thus evaluated as 0.00 by the engine, a neural network trained with the first one would inevitably evaluate the second accordingly, leading to the phenomenon observed in figure 13.

4.2.2 MLP networks with proposed feature inputs

The results of the MLP networks trained with the input dataset consisting of the 194 variables of the proposed set of features plus the 0 to 3 evaluation inputs according to each scenario, for both cases of dataset filtering, are presented in tables 5 and 6.

Table 5: Indicators of MLPs using the proposed features with mating evaluation filtering

Scenario	MAE		R^2	
	Testing	Validation	Testing	Validation
1	0.75 (0.82±0.04)	0.75 (0.81±0.04)	0.46 (0.42±0.02)	0.52 (0.48±0.03)
2	0.93 (1.00±0.06)	0.93 (1.00±0.06)	0.36 (0.30±0.04)	0.38 (0.33±0.03)
3	0.96 (1.05±0.04)	0.97 (1.04±0.04)	0.32 (0.26±0.03)	0.34 (0.30±0.03)

* Each cell contains the best performance in the respective dataset along with the mean value and standard deviation in parenthesis.

Table 6: Indicators of MLPs using the proposed features with over-20 evaluation filtering

Scenario	MAE		R^2	
	Testing	Validation	Testing	Validation
1	0.44 (0.47±0.03)	0.44 (0.46±0.03)	0.72 (0.70±0.02)	0.73 (0.71±0.02)
2	0.58 (0.60±0.02)	0.57 (0.60±0.02)	0.57 (0.54±0.03)	0.59 (0.56±0.03)
3	0.62 (0.64±0.02)	0.62 (0.64±0.02)	0.51 (0.48±0.02)	0.50 (0.48±0.02)

* Each cell contains the best performance in the respective dataset along with the mean value and standard deviation in parenthesis.

The computation for all iterations regarding this method took a rough 16 hours; the training time for each network was approximately 10.7 minutes (642 seconds).

A graphic representation of target versus predicted values for the best performance in the test subset for the two dataset filtering cases are depicted in figures 14 and 15.

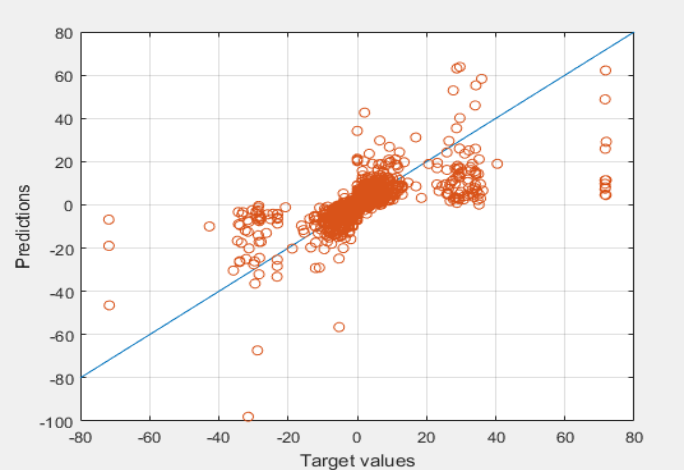


Figure 14: Target values vs Predictions of MLPs using the proposed features with mating evaluation filtering

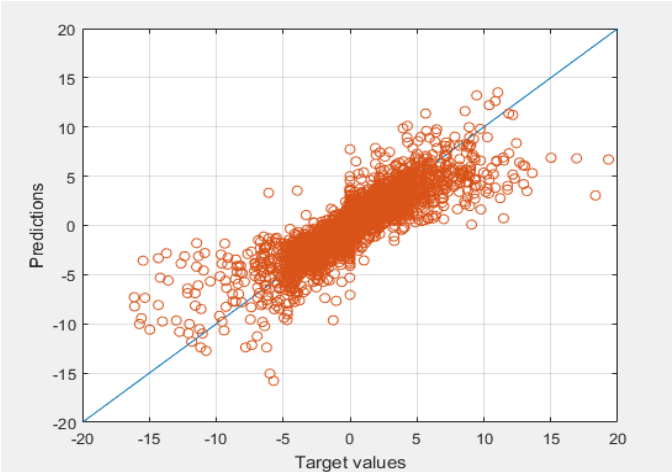


Figure 15: Target values vs Predictions of MLPs using the proposed features with over-20 evaluation filtering

These diagrams demonstrate the same pattern as the ones of the previous method in terms of the large value evaluation groups, the right quadrant placement of the mispredictions of large values, the proximity to the line of the ideal model, as well as the exact zero target value phenomenon.

4.2.3 MLP networks with bitmap representation inputs

The results of the MLP networks trained with the input dataset consisting of the 768 variables as dictated by the bitmap representation, again plus the 0 to 3 evaluation inputs according to each scenario, for both cases of dataset filtering, are presented in tables 7 and 8.

Table 7: Indicators of MLPs using bitmap representation with mating evaluation filtering

Scenario	MAE		R^2	
	Testing	Validation	Testing	Validation
1	0.70 (0.78±0.06)	0.71 (0.79±0.06)	0.50 (0.47±0.02)	0.50 (0.48±0.02)
2	0.85 (1.04±0.17)	0.85 (1.04±0.17)	0.40 (0.35±0.06)	0.42 (0.37±0.04)
3	0.94 (1.02±0.06)	0.95 (1.02±0.06)	0.35 (0.28±0.05)	0.35 (0.29±0.04)

* Each cell contains the best performance in the respective dataset along with the mean value and standard deviation in parenthesis.

Table 8: Indicators of MLPs using bitmap representation with over-20 evaluation filtering

Scenario	MAE		R^2	
	Testing	Validation	Testing	Validation
1	0.42 (0.45±0.03)	0.43 (0.45±0.03)	0.76 (0.74±0.02)	0.76 (0.73±0.02)
2	0.53 (0.56±0.02)	0.54 (0.56±0.02)	0.64 (0.60±0.02)	0.62 (0.60±0.01)
3	0.59 (0.61±0.02)	0.59 (0.61±0.02)	0.56 (0.54±0.01)	0.56 (0.54±0.01)

* Each cell contains the best performance in the respective dataset along with the mean value and standard deviation in parenthesis.

The computation time for the iterations for this method was approximately 79 hours, bringing the training time for each network at roughly 158 minutes (9480 seconds).

A visual representation of the desired values versus the predicted values for the best performance in the test subset for the two dataset filtering cases can be seen in figures 16 and 17.

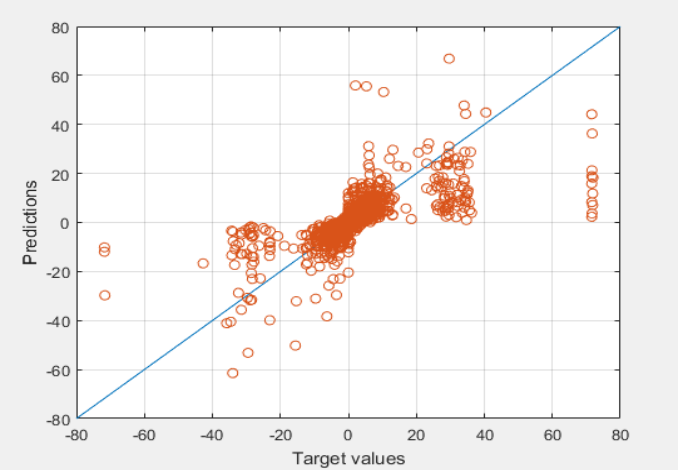


Figure 16: Target values vs Predictions of MLPs using bitmap representation inputs with mating evaluation filtering

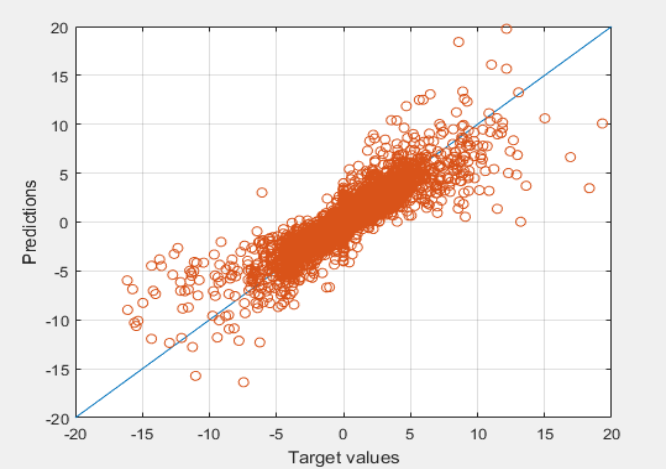


Figure 17: Target values vs Predictions of MLPs using bitmap representation inputs with over-20 evaluation filtering

Again, we notice the same grouping and correct quadrant effect for evaluations with large values and the general tendency of the data points to approximate the ideal blue line, mainly when the data set is filtered over 20 at figure 17. The particular situation with the exact zero target values is present in this experiment as well, since the source of its existence is not related to the type of inputs that are used, but in the actual engine evaluation.

4.2.4 Method comparison

In this section, the best performance results of the three examined methods for both cases of dataset filtering are compared and discussed. The aggregate results in tables 9 and 10 are only a repetition of the results already shown, presented together for easy comparison of the methods. Computational time for training has also been included as a comparison factor, but it should be noted that in the MLP cases it is a mean value over the iterations and has not been measured explicitly for each scenario.

Initially, observing the results of the over-20 evaluation filtering compared to those of mating evaluation filtering, it is obvious that the indicators are affected by the existence of larger

evaluations, which are sparser in the dataset for reasons explained in section 4. Errors in such evaluations, being larger in value, obviously have a greater impact on the mean absolute error and apparently the same goes for the coefficient of determination. This also indicates a discrepancy in the performance of the network, which was also observed in the respective diagrams in figures 11, 14 and 16, as in areas away from zero the network performance is poorer as it mainly has to perform extrapolation in order to give a prediction. Conversely, in areas near zero where data is denser, the necessary interpolation is much more likely to be efficient.

Table 9: Aggregate presentation of indicators with mating evaluation filtering

Scenario	Method	MAE		R^2		Training time (s)
		Testing	Validation	Testing	Validation	
1	RBF - proposed inputs	0.76	0.76	0.50	0.52	7304
	MLP - proposed inputs	0.75	0.75	0.46	0.52	642
	MLP - bitmap inputs	0.70	0.71	0.50	0.50	9480
2	RBF - proposed inputs	0.83	0.84	0.43	0.45	12448
	MLP - proposed inputs	0.93	0.93	0.36	0.38	642
	MLP - bitmap inputs	0.85	0.85	0.40	0.42	9480
3	RBF - proposed inputs	0.85	0.85	0.41	0.43	5745
	MLP - proposed inputs	0.96	0.97	0.32	0.34	642
	MLP - bitmap inputs	0.94	0.95	0.35	0.35	9480

Table 10: Aggregate presentation of indicators with over-20 evaluation filtering

Scenario	Method	MAE		R^2		Training time (s)
		Testing	Validation	Testing	Validation	
1	RBF - proposed inputs	0.38	0.38	0.78	0.80	5344
	MLP - proposed inputs	0.44	0.44	0.72	0.73	642
	MLP - bitmap inputs	0.42	0.43	0.76	0.76	9480
2	RBF - proposed inputs	0.44	0.43	0.72	0.73	5809
	MLP - proposed inputs	0.58	0.57	0.57	0.59	642
	MLP - bitmap inputs	0.53	0.54	0.64	0.62	9480
3	RBF - proposed inputs	0.45	0.44	0.69	0.70	8638
	MLP - proposed inputs	0.62	0.62	0.51	0.50	642
	MLP - bitmap inputs	0.59	0.59	0.56	0.56	9480

The method using RBF architecture along with our proposed feature set appears to be the overall best performer in all cases. Especially for the over-20 evaluation filtering, which is much better scoped in terms of playing the game, the MAE achieved by the RBF method is comparable to the advantage that chess engines typically give to the white side just for having the turn to play in the beginning of a game. Moreover, the fact that the performance of the RBF method does not improve that much by the existence of the supplementary evaluation inputs is certainly an asset, making the possibility of eliminating them, quite feasible for this method as opposed to the two MLP methods where this would have a significant impact as evidently demonstrated by scenario 3.

One minor exception to the above is the performance within mating evaluation filtering where the bitmap MLP network achieves a slightly better MAE than the RBF network, only in scenario 1. However, even then it does not surpass it regarding the R^2 indicator and the amount of time needed for training the network is significantly larger. A more interesting observation is that in scenario 3, where no low-depth evaluation input variables are provided, the RBF method performs more or less the same (even slightly better in respect to R^2) as the bitmap MLP does in scenario 2, where two low-depth evaluation variables assist its performance. Therefore, the RBF method is the one propose as the best performing and most promising approach.

Generally, we may notice that within both filtering approaches, all methods perform the best in scenario 1, when they are aided by all the supplementary low-depth evaluation inputs, and then better in scenario 2, when only the two lesser depth evaluations are included, than in scenario 3, when no such inputs are used. Interestingly and partially mentioned before, the three methods seem to be affected to different extends by these additional inputs with the RBF method being the least dependant on them and the bitmap MLP being the most improved in their presence. Also, the improvement of the RBF method in scenario 2 over scenario 3 seems almost insignificant as opposed to both the MLP methods.

Comparing the two MLP methods, the overall results of the bitmap method are generally a bit better than our proposed feature method. However, the training of such a network using 768 to 771 inputs in contrast with the one using 194 to 197 inputs takes about 15 times more computational time. We can observe that our proposed set of features considerably reduces computation time needed for network training over the bitmap representation approach, mostly due to the significantly lower amount of input variables.

Finally, it should be pointed out that although the training of a neural network using our proposed method requires some preparation and a considerate amount of time for training, the resulting model is able to provide evaluation predictions for several moves ahead without processing any search tree as a chess engine would do. This is the practical importance and actual motivation of this work as these predictions are obtained in negligible amounts of time, compensating for the minor deficit in accuracy. In fact, the time needed for a prediction has been measured as an average over 10000 predictions. The average time for obtaining a prediction using our trained model is 6.9 milliseconds. Although this is an impressive fact in its own right, there is a more useful aspect to it. Regardless of the amount of training time the model might need, but more importantly regardless of the amount of time that is given to the chess engine to analyze each position during the training dataset construction, the prediction time would practically remain the same. This implies that in a context where time is a limiting factor, as in competitive play, when a chess engine would have as much position analysis available as the time limit would allow, a software based on such a model could practically obtain evaluations based on the analysis time given during its training dataset construction, making the actual competitive time limit irrelevant.

SECTION 5

Conclusions – Future Work

In this Thesis we attempted to approach the subject of the evaluation of a chess position by utilizing the learning capabilities of neural networks. The primary goal was to substitute the process of tree search that chess engines perform when approximating a position evaluation with the predictions of a neural network based on the knowledge it can obtain through training. The benefits of such an approach would be that even though the predictions may not be as accurate as the ones of a chess engine, the sustained error is compensated by the gain of time, processing power and memory consumption that the chess engine needs in order to perform the tree search.

We considered and compared three methods, two of our own creation and a third one that had already been published [9]. Our considered methods involved the training of an RBF and an MLP network by using a proposed set of position features derived by the features described in the Stockfish documentation, which was the chess engine the networks competed against, as input variables. The already published method that we compare against also involved an MLP, but the input for its training was a board representation described in [9], called bitmap representation.

In order for the training input dataset to be created, over 1500 top-level games where collected and processed by a Java application of our own development. The various inputs for the networks where appropriately extracted from the arising positions depending on the respective method, and the target values for the network training where obtained by using the Stockfish 10 chess engine on each position for a specific amount of time. In order to provide a dynamic aspect of the chess position in the training dataset, we devised three more supplementary input variables with the actual evaluation of Stockfish in much lower depths of search than the one we would ultimately try to predict by using the neural network. The result of this whole procedure, described in detail in the respective section of the Thesis, was a database of about 80000 chess positions. The Matlab neural toolbox was utilized in order to facilitate the training of the involved MLP networks, while the RBF network training was handled by custom Maltab scripts written specifically for this purpose.

Three variations regarding the dataset, mentioned as scenarios, where examined for each method. In first scenario all three of the supplementary inputs were present, in the second scenario the input with the highest depth was removed and in the third one all three were removed. This gave us the opportunity to examine the impact that such inputs may have on the produced networks in the rivaling methods.

An issue that we had to tackle was that the dataset had an inherent uneven distribution of data in regions closer to zero and further from it, as described in more detail in section 4. This issue derives from the evaluation scores in high level chess games typically not being very high as when this imbalance begins to grow, or even is just foreseen, the losing player resigns and the game ends. A data-related experiment that we tried in order to approach this situation was to filter out target values above the value of 20 before training, in order to force a more even distribution of data in the domain that the network has to make predictions.

The results for every variation of each method were presented separately in both table and diagram form and remarkable observations were discussed. In addition, a table of comparative results was provided and the three methods were compared. It is our assessment that the method of training an RBF network with our proposed set of features as inputs, with or without the additional evaluation input variables, produces the best performance and deserves to be further investigated. Therefore it is the solution we propose as an approach in our case. Furthermore, our proposed set of features significantly decreases the network training time, compared to existing techniques. Our entire approach, i.e. to avoid exploring the search tree and rely on the network predictions instead, leads to obtaining an evaluation for several moves ahead (about 28 moves) in about 7 milliseconds on average.

Future work that may build upon this Thesis could include constructing a larger database of positions, and factoring in more hand-crafted supplementary input variables, in order to investigate to what extend could the MAE be minimized and the R^2 maximized. Also, the composition of the proposed feature set could be revisited and examine if the number of variables could be decreased in an effort to reduce training computation time even more. Another idea could be to apply a different training algorithm to the RBF networks, such as a non-symmetrical fuzzy means algorithm which has been proposed as an improved version of the symmetrical fuzzy means [25, 26] used in our case. In terms of improving performance, one may also consider addressing the 0.00 phenomenon, maybe by including the utilization of an actual endgame tablebase in order to distinguish such circumstances.

Besides moving towards improving the performance of the RBF network itself, another more ambitious course of action would be the development of a fully functional chess-playing application with our proposed RBF model in its core. Such an approach could utilize a control method called ‘model predictive control’ (MPC) [28, 29] as a decision making tool that would rely on our model in order to decide the next move.

References

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

[2] H. Nasreddine, H. S. Poh and G. Kendall, "Using an Evolutionary Algorithm for the Tuning of a Chess Evaluation Function Based on a Dynamic Boundary Strategy," *2006 IEEE Conference on Cybernetics and Intelligent Systems*, Bangkok, Thailand, 2006, pp. 1-6.

[3] E. Vázquez-Fernández, C. A. C. Coello and F. D. S. Troncoso, "An evolutionary algorithm for tuning a chess evaluation function," *2011 IEEE Congress of Evolutionary Computation (CEC)*, New Orleans, LA, USA, 2011, pp. 842-848.

[4] D. B. Fogel, T. J. Hays, S. L. Hahn and J. Quon, "A self-learning evolutionary chess program," in *Proceedings of the IEEE*, vol. 92, no. 12, pp. 1947-1954, Dec. 2004.

[5] Omid E. David, Nathan S. Netanyahu, and Lior Wolf, “DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess”, International Conference on Artificial Neural Networks (ICANN), Springer LNCS, Vol. 9887, pp. 88-96, Barcelona, Spain, 2016.

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

[7] Sebastian Thrun, “Learning to Play the Game of Chess”, NIPS'94: Proceedings of the 7th International Conference on Neural Information Processing Systems, January 1994, pp 1069–1076.

[8] Matthew Lai, “Giraffe: Using Deep Reinforcement Learning to Play Chess”, arXiv:1509.01549.

[9] G. H. Hardy, “Learning to Play Chess with Minimal Lookahead and Deep Value Neural Networks”, 2017.

[10] Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”, Science, 2018.

[11] Alexandridis, A., E. Chondrodima, N. Giannopoulos, H. Sarimveis, “A Fast and Efficient Method for Training Categorical Radial Basis Function Networks”, IEEE Transactions on Neural Networks and Learning Systems, 28(11) (2017), pp. 2831 - 2836.

[12] Standard: Portable Game Notation Specification and Implementation Guide,

https://ia802908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt

[13] Portable Game Notation, Wikipedia,

https://en.wikipedia.org/wiki/Portable_Game_Notation

[14] Forsyth-Edwards Notation, Wikipedia,

https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation

[15] UCI protocol, http://wbec-ridderkerk.nl/html/UCIProtocol.html

[16] Search, Chess Programming Wiki, https://www.chessprogramming.org/Search

[17] Evaluation, Chess Programming Wiki, https://www.chessprogramming.org/Evaluation

[18] Stockfish, Chess Programming Wiki, https://www.chessprogramming.org/Stockfish

[19] Stockfish Evaluation Guide, https://hxim.github.io/Stockfish-Evaluation-Guide/

[20] Simon Haykin (1999), “Neural Networks A Comprehensive Foundation”, Second Edition, Pearson Education Inc.

[21] Christopher M. Bishop (1995) “Neural Networks for Pattern Recognition”, Oxford University Press Inc.

[22] Levenberg–Marquardt algorithm, Wikipedia, https://en.wikipedia.org/wiki/Levenberg-Marquardt_algorithm

[23] Nazri Mohd Nawi, Abdullah Khan, M.Z. Rehman, “A New Levenberg Marquardt based Back Propagation Algorithm Trained with Cuckoo Search”, Procedia Technology, Volume 11, 2013, pp 18-23, ISSN 2212-0173.

[24] Gavin H., “The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems”, 2013.

[25] Alexandridis A., Chondrodima E. and Sarimveis, H., Cooperative learning for radial basis function networks using particle swarm optimization. Applied Soft Computing. 2016.

[26] Alexandridis A., Chondrodima E. and Sarimveis H., "Radial Basis Function Network Training Using a Nonsymmetric Partition of the Input Space and Particle Swarm Optimization," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 24, no. 2, pp. 219-230, Feb. 2013.

[27] Endgame tablebase, Wikipedia, https://en.wikipedia.org/wiki/Endgame_tablebase

[28] Stogiannos, M., A. Alexandridis, H. Sarimveis, “Model predictive control for systems with fast dynamics using inverse neural models”, ISA Transactions, 72 (2018), pp. 161-177

[29] Alexandridis, A., H. Sarimveis, K. Ninos, “RBF network training using a non-symmetric partition of the input space – Application to an MPC configuration”, Advances in Engineering Software, 42(10) (2011), pp. 830-837.