Hindawi
Complexity
Volume 2023, Article ID 7143943, 16 pages
https://doi.org/10.1155/2023/7143943

Research Article

Chess Position Evaluation Using Radial Basis Function Neural Networks

Dimitrios Kagkas, Despina Karamichailidou, and Alex Alexandridis

Department of Electrical and Electronic Engineering, University of West Attica, Ancient Olive Grove Campus, Thivon 250 and P. Ralli, Aigaleo 12244, Greece

Correspondence should be addressed to Alex Alexandridis; alexx@uniwa.gr

Received 15 November 2022; Revised 26 January 2023; Accepted 2 March 2023; Published 17 April 2023

Academic Editor: Lingzhong Guo

The game of chess is the most widely examined game in the field of artificial intelligence and machine learning. In this work, we propose a new 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 estimations of a properly trained neural network. Such an approach offers the benefit of having an estimation for the position evaluation in a matter of milliseconds, while the time needed by a chess engine may be several orders of magnitude longer. The proposed approach introduces models based on the radial basis function (RBF) neural network architecture trained with the fuzzy means algorithm, in conjunction with a novel set of input features; different methods of network training are also examined and compared, involving the multilayer perceptron (MLP) and convolutional neural network (CNN) architectures and a different set of input features. All methods were based upon a new dataset, which was developed in the context of this work, 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 dataset, which contained data from over 80,000 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 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.

1. Introduction

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 champions 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 sophisticated 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 use chess engines for their tournament preparation and overall training. 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 a tree search algorithm [2–4].

Many demonstrations of ideas involving the game of chess and machine learning currently exist. One work is based on the optimization of the handcrafted evaluation function using evolutionary algorithms for tuning the function parameters with a strategy named dynamic boundary strategy [5]. 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 [6]. Another example of the use of evolutionary algorithm, in conjunction with artificial neural networks, has been a program that learned to play the game by playing against itself [7].

The use of artificial neural networks has also been very common in researches in this domain. Neural networks are structures that resemble the human brain in its principle of operation [8]. 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.

In the case of DeepChess [3], a neural network is trained to evaluate positions using millions of games without including any other chess-related knowledge, such as the rules of the game. KnightCap also used a neural network along with custom-made attack and defence tables and piece weights [9]. Another case is NeuroChess [10], a chess-playing program that relies on neural networks and handcrafted features for evaluating positions, trained to predict game results. The use of neural networks is taken one step further by Giraffe [11], a chess program that acquired chess knowledge by self-play, which utilizes them not only to tune the evaluation function parameters but also to perform pattern recognition and feature extraction. Pattern recognition has also appeared in a research involving multilayer perceptrons (MLPs) and convolutional neural networks (CNNs) in order to evaluate positions by using look-ahead algorithms as little as possible [12]. Probably, the most impressive, however, is the famous AlphaZero [13] 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.

There are several other works that employ ANN to various chess modeling tasks. For example, in [2], the potential of CNN models in predicting moves in chess is examined; three layered CNN models are used to make move predictions, attesting the effectiveness of the proposed approach. In [14], various machine learning approaches such as regression and Naive Bayes classifiers are used for predicting chess game results. In [4], MLP and CNNs of predetermined sizes are tested in various chess modeling approaches, as for example chess position evaluations and moves classification; the experimental results showed the superiority of MLP models, despite chessboard representation methods, such as bitmap and algebraic input. In [15], a deep neural network paired with a look-ahead algorithm for evaluation function is presented. The proposed moves yielded by the proposed method were compared against those proposed by Stockfish; the model was able to predict moves of equal strength to those of Stockfish of over 80% in all the sample positions. In [16], an approach based on CNN models with a limited look ahead for chess game playing is proposed. The study found that the proposed method can effectively identify tactical patterns in games and yielded promising results when tested against the Stockfish computer engine.

Another machine learning paradigm that has met great success in chess is reinforcement learning (RL) [17]. The basic idea behind RL is that an agent finds the optimal solution of a problem through trial and error procedures and a reward and punishment framework; for example, in [18], a No-Go algorithm based on reinforcement learning using CNN to evaluate legal chess moves is proposed. In [19], a Chinese chess game algorithm is proposed based on reinforcement learning; a self-play learning model is constructed by combining deep CNNs and Monte Carlo search tree algorithm to simulate chess moves. In [20], an LSTM model is built examining various chessboard representation methods in order to check which one is more relevant for the neural network training process. In [21], a supervised agent is trained using game records, where the performance of the agent is increased through self-play with the on-policy reinforcement learning algorithm, without search and without making assumptions about the true game state.

Many of the aforementioned publications rely on neural networks for various purposes regarding the game of chess, focusing mainly on the architectures of CNNs and MLPs. While the contribution of these works is of great importance in the field of chess modeling, they are based on standard ANN architectures and well-known training algorithms that may come with certain limitations when applied to high-dimensional data sets. For instance, determining critical network parameters like the network size is implemented using tedious search procedures, based usually on exhaustive search both for hidden layers and number of nodes in each layer, which may be computational impractical; as a result, suboptimal network sizes may be selected. That means further improvement could be achieved by implementing different ANN architectures and novel training algorithms, so that an improved model in terms of modeling accuracy and computational efficiency is yielded.

A promising candidate architecture for an alternative modeling approach could involve the radial basis function networks (RBFs) [8]. RBF networks consist of a single hidden layer exhibiting several significant properties such as universal approximation, robustness, and good generalization, and as a result, they represent a very competitive choice, for example, over MLPs, in terms of structure, training process, and optimization for modeling nonlinear systems; it is because of their simple structure that RBF networks come with several important advantages regarding shorter training speed and increased modeling efficiency [22]. A recent work that employs RBF networks in chess modeling framework is presented in [23], where a simple RBF network is used to approximate Q-learning function used Chinese chess evaluation through a continues self-learning procedure. The results of the experimental work seem to be very promising; however, a more optimized RBF network would improve the performance. However, notably, and to the best of the authors' knowledge, RBF neural network architecture has not been used in evaluating positions within the standard game of chess.

To this end, in this work, the benefits of RBF networks are exploited in order to build a robust and effective model for evaluating chess positions at an acceptable computational cost. More specifically, our goal is to approximate the absolute evaluation score of a chess position based merely on neural network estimations, avoiding any kind of tree search that chess engines rely on [24]. The advantages of the RBF neural network architecture are exploited in order to achieve maximum accuracy in estimating the evaluation of a chess position to an acceptable degree, without looking into specific moves or exploring the tree search. The network center parameters are determined through the fuzzy means (FM) algorithm, which has been applied successfully to various applications offering models with increased accuracy [25], whereas the linear weights can be optimally determined through linear least squares. More specifically, the proposed approach offers (i) variable network sizes and flexible parameter selection, where the size of the model is determined based on the efficient FM algorithm, and (ii) a reduced number of algorithmic parameters, as the proposed approach uses only one parameter that needs to be tuned; thus, the method is able to handle large amount of data and multiple inputs that could affect the evaluation score of a chess position in shorter training times. Another important contribution of this work is the introduction of a new set of chess position features, free of domain specific knowledge, and any kind of assisting structures, such as lookup tables, derived from a custom-made Java application by processing thousand games in the Stockfish engine [26]. The proposed neural network model and the selected set of features provide estimations for several moves ahead in a much shorter timeframe than a chess engine would need to achieve the same result, eventually yielding a method able to produce highly accurate models for chess position evaluations.

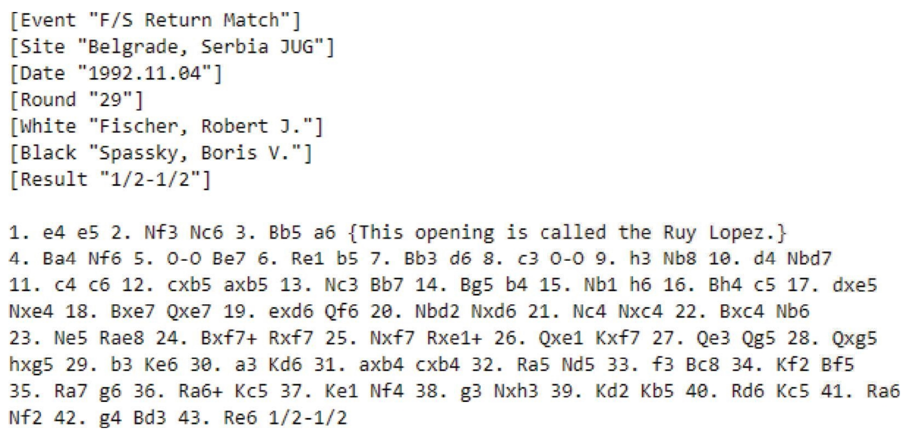
The rest of this paper is structured as follows: Section 2 gives an introduction to chess engines, the derivation of the proposed set of features as well as a description of the process for extracting the network training data. Section 3 gives a description of the RBF network architecture and the fuzzy means training algorithm. Section 4 describes the experimental procedure in detail, where the modeling results are presented and discussed. The paper concludes by outlining the advantages of the proposed approach and setting directions for future work.

2. Chess Dataset Creation

2.1. Chess Engines. Chess engines evaluate a given position internally in the form of some kind of representation. Some of the most popular chess engines are Stockfish, Komodo, Houdini, and Rybka [27]. Chess engines in general are able not only to provide an evaluation of the position but also to suggest continuations of moves that they consider the strongest. As for the representation of the position, there are many formats to describe a specific position of a game, most popular of which are the "Portable Game Notation" (PGN) and the "Forsyth-Edwards Notation" (FEN). PGN is actually plain text in a standardized format, developed by Steven J. Edwards, and readable by both humans and software, which is used to record moves and related data of a chess game. An example of a PGN can be seen in Figure 1. It is divided into two segments, the former of which contains the game information. The latter segment of the PGN format contains the move text, i.e., the moves of the game, in "Standard Algebraic Notation" (SAN). On the other hand, 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. Figure 2 depicts an example of FEN representation which, as it can be seen, it is a single line of text that consists of a specific number of fields, including piece placement, active color, and other important information, separated by spaces.

Chess engines search the best move by looking ahead at various sequences of moves and evaluating the arising positions, since in general such a result cannot be concluded from the static information of a position. The calculation of the evaluation value that represents which side is better and by how much 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. In the case of chess, which is a two-player game that belongs to the family of zero-sum, the algorithm usually applied 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 [28].

Every chess 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. 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. 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



```
[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.

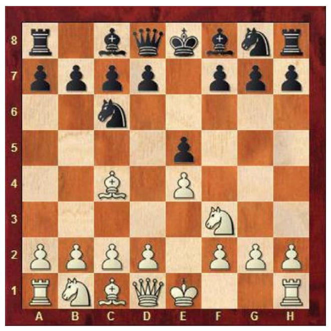

r1bqkbnr/pppp1ppp/2n5/4p3/2B1P3/5N2/PPPP1PPP/RNBQK2R b KQkq -3 3

Figure 2: FEN example.

position, provided that the best moves are played for both sides.

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, mainly between middle game (mg) and endgame (eg). It should be pointed out that since each feature is calculated for each player, it is the difference between the respective counterparts that make 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. 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. 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 (PSQT). 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 handcrafted 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 including moves that are not technically legal.

King safety is the most complex of all the evaluation concepts. It is composed from many features regarding specific setups of defending and attacking pieces relative to the position of the king. The actual components used to measure king safety are not the same across engines, but they more or less follow the same 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 is a term used to describe the position of all pawns regardless of the placement of other pieces but including the relative placement of other pawns. Features in this category represent the state of each individual pawn such as doubled, isolated, and blocked. In a more generalized sense though, there are some features that consider more pieces relatively to the examined pawn in specific manners like setups that make it 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. These features are measured by number of occurrences on the board and weighted accordingly.

2.2. Input Feature Selection. In this work, the proposed set of features is derived using the Stockfish chess engine. Stockfish [26] is an open-source engine, licensed under the GPL v3.0 and compatible with the UCI protocol. The UCI protocol specification [29], 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. Stockfish was developed by Tord Romstad, Marco Costalba, Joona Kiiski, and Gary Linscott, and it is one of the strongest chess engines in the world as of 2018. Stockfish relies on certain feature and function components in order to evaluate chess positions using a heuristic function. These components are briefly presented in Tables 1 and 2.

2.3. Data Extraction. 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 variables need to be determined. Two conceptually different groups of input variables were chosen. The first group consists of a certain set of static features of a chess position that describes the position at its current state, and indeed, the only required knowledge for deriving them is the current arrangement on the board, e.g., in the form of a FEN. 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 requires 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 the SQL database. The operation of the game processing application is displayed as a flow diagram in Figure 3. First, 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.

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 (https://www.chessgames.com/, https://www.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 where the database increased incrementally, and networks were trained using the respective datasets, gradually improving performance.

As mentioned, the chess engine that was used is 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 same features that Stockfish relies on. What Stockfish considers as a single feature may actually be composed of some other subcalculations 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, 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. 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 the 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, respectively, 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 relates to the resulting evaluation that

Table 1: Stockfish evaluation feature components.

Feature	Description	Components
Material	Weighted values and bonuses related to pieces and pawns, and their position on the board	Nonpawn material, PSQT mg, PSQT eg, piece value mg, piece value eg
Mobility	Values and bonuses related to the available squares pieces can reach	Mobility, mobility area, mobility mg, mobility eg
King	Bonuses and penalties related to king placement such as position relative to friendly and enemy pieces, pawn shelters, pawn storms, weak squares, attacks, and available checks	King pawn distance, blockers for king, knight defender, king attacks, king attackers count, king attackers weight, pawnless flank, flank attack, flank defence, storm square, shelter storm, shelter strength, endgame shelter, strength square, weak squares, weak bonus, king danger, check, safe check, unsafe checks, king mg, king eg
Space	Weighted bonus for safe squares on own side of the board	Space, space area
Initiative (winnable)	Bonus or penalty for initiative aspects such as passed pawns, piece infiltration, and flank majorities	Initiative, initiative total eg, initiative total mg
Imbalance	Weighted values and bonus or penalty for material differences such as the bishop pair	Imbalance, imbalance total, bishop pair
Pieces	Values and bonuses or penalties related to specific patterns of piece placement on the board	Outpost, outpost square, reachable outpost, outpost total, rook on file, long diagonal bishop, rook on queen file, weak queen, trapped rook, rook on king ring, queen infiltration, bishop pawns, bishop x-ray pawns, minor behind pawn, king protector, pieces mg, pieces eg
Pawns	Values and bonuses or penalties related to specific patterns of pawn placement on the board	Supported, phalanx, connected, connected bonus, opposed, isolated, doubled, doubled isolated, backward, blocked, weak unopposed pawn, weak lever, pawns mg, pawns eg
Passed pawns	Values and bonuses or penalties related to passed pawns placement on the board	Candidate passed, passed rank, passed file, passed leverable, passed block, king proximity, passed mg, passed eg
Attack	Values related to attacked squares and pieces	Pawn attack, knight attack, bishop x-ray attack, rook x-ray attack, queen attack, queen diagonal attack, king attack, pinned, attack
Threats	Values related to threatened and supported pieces and pawns	Hanging, restricted, weak enemies, safe pawn, threat safe pawn, pawn push threat, minor threat, rook threat, king threat, weak queen protection, slider on queen, knight on queen, threats mg, threats eg

Table 2: Stockfish evaluation functions and helpers.

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

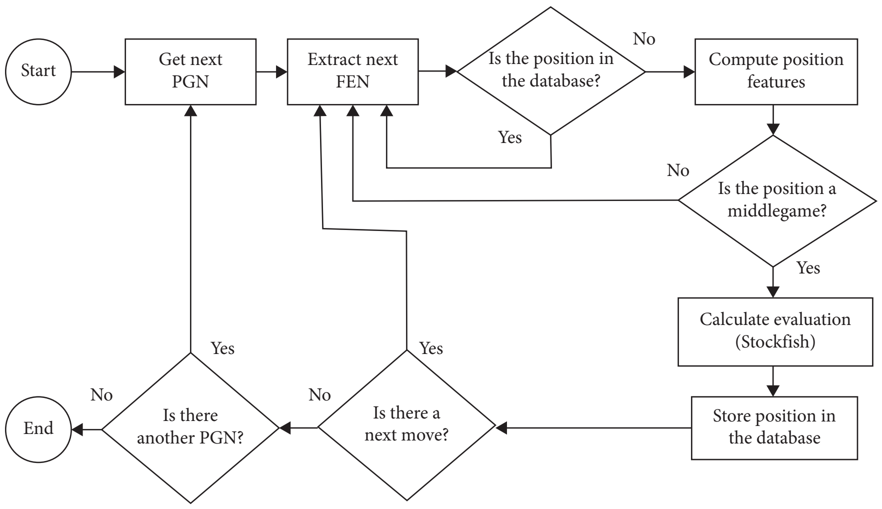

Start
Get next PGN
Extract next FEN
Is the position in the database?
No
Yes
Compute position features
Is the position a middlegame?
No
Yes
Calculate evaluation (Stockfish)
Store position in the database
Is there a next move?
Yes
No
Is there another PGN?
Yes
No
End

Figure 3: Game processing application flow diagram.

may be overlooked in case of only using the aggregate values. This resulted in a total of 194 input variables.

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, and 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 lose, and the best move in the position. As from the year 2012, tablebases contain every endgame position that includes up to seven pieces. 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 nonpawn 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 i7-4779quad-core at 3.40 GHz and 8 GB of RAM, and the configuration of the engine itself. The important points of the configuration in our case include occupying 3 computational threads per engine instance, a hash table of 1024 MB, 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 nonbest 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, takes 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 were 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.

3. Radial Basis Function Neural Networks

Radial basis function (RBF) networks [8] represent a special type of feed-forward neural networks that have been utilized successfully in a diverse range of applications from regression and classification problems [30, 31] to system control [32, 33]. Proposed in late 80s by Broomhead and Lowe, RBF neural networks consist of one hidden layer employing radial basis function as activation function and offer many important properties such as universal approximation, generalization, and robustness [22]. The simple structure of an RBF network allows for significant advantages regarding speed and efficiency when compared to other popular architectures like MLP networks [25].

A visual representation of an RBF network with N input variables, L hidden nodes, and one output variables is presented in Figure 4. Each node in the hidden layer implements a radial basis function characterized by a center vector $c_j \in \mathbb{R}^N$ $(j = 1, 2, \cdots, L)$, which is equal in dimension with the input layer, and, depending on the choice of the RBF, a width value. The structure of the RBF network entails a nonlinear mapping between the input and the hidden layer while the output layer is linearly attached to the hidden layer, where each connection is weighted by a value known as synaptic weight $w_j \in \mathbb{R}$ $(j = 1, 2, \cdots, L)$.

Given a set of input-output data,

$$(\mathbf{x}_i, y_i), \quad i = 1, 2, \cdots, K, \tag{1}$$

where $\mathbf{x}_i \in \mathbb{R}^N$, $y_i \in \mathbb{R}$, N is the number of input variables, K is the total number of examples, and L is the number of units in the hidden layer, and training an RBF network consists of computing the vector of synaptic weights, but also the number and location of the RBF centers and the corresponding widths where necessary [34] yielding an

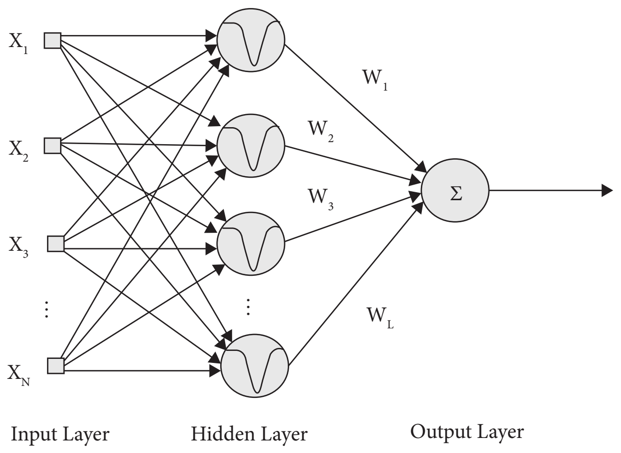


FIGURE 4: An RBF network representation.

approximating function $f\colon \mathbb{R}^N \longrightarrow \mathbb{R}$, so that each input yields a corresponding target output.

The nodes in the hidden layer implement an RBF function z_j $(j = 1, 2, \cdots, L)$, where for this work, the thin plate spline function is employed

$$z(v) = v^2 \log(v), \tag{2}$$

where v denotes the activity of a certain input vector to a certain RBF node. For instance, the activity of the l-th unit to the k-th input vector is defined as

$$v_l(\mathbf{x}(k)) = \left\|\mathbf{x}(k) - \mathbf{c}_l\right\|_2, \quad k = 1, 2, \cdots, K, \tag{3}$$

where $\mathbf{c}_l$ is the center vector of the l-th RBF unit and $\mathbf{x}(k)$ is the k-th input vector.

The output layer gives the RBF network evaluation, and the k-th input vector is given by

$$\hat{y}(\mathbf{x}(k)) = \sum_{j=1}^{L} w_j \varphi_j\big(\mathbf{x}(k), \mathbf{c}_j\big), \tag{4}$$

where w_j is the synaptic weight of the k-th RBF unit.

After determining the hidden layer size and the location of the RBF centers, the vector of synaptic weights is determined. Due to the linear relationship between the hidden and the output layers, the synaptic weights can easily be calculated using linear regression. A typical approach is using the linear least squares method, where in the matrix form one gets the following:

$$\mathbf{w} = \left(Z^{\mathrm{T}} Z\right)^{-1} Z^{\mathrm{T}} \mathbf{y}, \tag{5}$$

where Z denotes a matrix that contains the outputs of the hidden layer for every input vector and $\mathbf{y} \in \mathbb{R}^K$ denotes a vector that contains the desired output, i.e., the target values.

Computing the number and locations of RBF nodes is deemed as the most challenging aspect of training an RBF network, which has great impact on its modeling capabilities. It is usually handled by using a clustering technique, such as the k-means algorithm which was actually the most popular approach in training RBF networks in their early days [35]. Despite its popularity, this algorithm comes with certain disadvantages that originate from its stochastic and iterative nature. For instance, the algorithm is too sensitive to the initial center location of clustering that may arise converging problems. Apart from that, a number of different runs are needed in order to check the consistency of the results. As for the network size, k-means algorithm lacks the ability to automatically determine the number of RBF centers, which means that an exhaustive trial and error procedure must be incorporated yielding the procedure inappropriate for cases with high dimensionality.

As the datasets in this work fall into this category, the use of a more efficient and less computational expensive procedures is mandatory. To this end, to overcome the disadvantages of the aforementioned technique, this work adopts the standard fuzzy means (FM) algorithm [25] described briefly in the next section which has been applied successfully to a wide range of applications [32, 36].

3.1. Fuzzy Means Algorithm for RBF Networks Training. The fuzzy means (FM) algorithm, in contrast to the k-means algorithm, is able to determine the number and the locations of the RBF centers in an automatic way avoiding any time-consuming iterative and stochastic procedures. The main idea of the algorithm is the partitioning of each dimension of the input domain into one-dimensional triangular fuzzy sets, creating a grid of multidimensional subspaces in the input space, where a selected subset of these subspaces will represent the RBF 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 normalized input variables, or in other words, the input space has N dimensions, and every dimension is partitioned into the same number M of triangular fuzzy sets, which can generally be described as follows:

$$A_m = \{a_m, \delta a\}, \tag{6}$$

where A_m denotes the m-th fuzzy set and a_m and $\delta\alpha$ denote the center element and the half of the width of m-th fuzzy set, respectively.

Partitioning the input space in this manner produces a total of S multidimensional fuzzy subspaces, equal to the product of the number of fuzzy sets in each dimension. Each fuzzy subspace is denoted as $\mathbf{A}(s)$, where $s = 1, 2, \cdots, S$, and its center element will be a vector containing the center elements of the respective fuzzy set in each dimension. For example, for a two-dimensional input using five fuzzy sets in each dimension, the total fuzzy subspaces are $S = 5^2$. These subspaces represent candidate RBF centers for the nodes on the hidden layer, which are selected in such a way so that they uniformly cover the distribution of the input space.

The criterion that the algorithm is based on for the selection of the appropriate subset of the fuzzy subspaces is derived from the membership function, which indicates to what degree a specific input vector belongs to a specific fuzzy subspace. The membership function for the k-th input vector to the s-th fuzzy subspace denoted as $\mu_{\mathbf{A}^s}(\mathbf{x}(k))$ is given by the following:

$$\mu_{\mathbf{A}^s}(\mathbf{x}(k)) = \begin{cases} 1 - d_r^s(\mathbf{x}(k)), & \text{if } d_r^s(x(k)) \le 1, \\ 0, & \text{otherwise}, \end{cases} \tag{7}$$

where $d_r^s(\mathbf{x}(k))$ is a distance function between the input data vector $\mathbf{x}(k)$ and the center of the fuzzy subspace $\mathbf{A}(s)$. This distance function is represented as a hypersurface 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 as from other vectors that do not. Since every dimension of the input space is partitioned using the same number of fuzzy sets, the distance function hypersurface is actually a hypersphere which can be defined by the following equation:

$$d_r^s(\mathbf{x}(k)) = \frac{\sqrt{\sum_{i=1}^{N} (\alpha_i^s - x_i(k))}}{\sqrt{N}\delta\alpha}, \tag{8}$$

where N denotes the dimensionality of the input space, α_i^s 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 training input vectors 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 noniterative procedure as the input dataset is processed only once, and hence, short computational times are achieved even in cases of large datasets [25].

More specifically, for each one of the input vectors in the training subset, the algorithm first checks if it is located outside from all the hyperspheres defined by the selected fuzzy subspaces. If that is the case, a new fuzzy subspace is selected based on the value of the membership function for each fuzzy set in every dimension for the current vector, and thus, a new RBF node is added to the network. After processing the whole input dataset, a selection of RBF centers that compose the hidden layer of the network has been constructed.

The algorithm handles only one operational parameter and that parameter is the number of fuzzy sets that all input dimensions will be partitioned by, which affects the size of the network and thus its modeling capabilities. There are various ways to define the value of the aforementioned parameter, the simplest of which in the case of the standard fuzzy means algorithm is a trial and error procedure, as each dimension is partitioned by the same number of fuzzy sets. In the case of the nonsymmetric version of the algorithm, which is based on the same basic principles with standard FM, there are N different operational parameters to be determined, as it is allowed different number of fuzzy sets in each dimension. In this case, an optimization procedure is necessary in order to produce an RBF network with optimal configuration within a logical computation time, most common of which include metaheuristic search methods belonging to evolutionary computation [36] and swarm intelligence [25, 37].

4. Case Study

4.1. Experimental Setup. Before applying any modeling techniques, the created datasets should be divided randomly in three independent subsets, namely the training, validation, and testing subsets. The training subset is used during the training procedure and more specifically for the determination of the RBF network parameters. The validation subset is used for model selection that is the size of the RBF network, in order to avoid overfitting to the training subset. As for the testing subset, it is used merely for an unbiased model evaluation. There is not a strict rule regarding the splitting of the available dataset. In this work, the available data points are randomly divided in a manner 50%, 25%, 25% for the training, validation, and testing subsets, respectively.

In order to evaluate the performance of the tested modeling techniques, two different indicators were chosen. The first concern the metric of the mean absolute error (MAE) given by the following equation:

$$\text{MAE} = \frac{\sum_{i=1}^{K}\left|y_i - \hat{y}_i\right|}{K}, \tag{9}$$

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

$$R^2 = 1 - \frac{\sum_{i=1}^{K}\left(y_i - \hat{y}_i\right)^2}{\sum_{i=1}^{K}\left(y_i - \overline{y}\right)^2}, \tag{10}$$

where y_i and $\hat{y}_i$, $(i = 1, 2, \cdots, K)$ are the real measurements and model estimations, respectively, and $\overline{y}$ is the mean value of the actual targets.

The optimal value of the operational parameter s for the RBF network, that is, the number of the fuzzy sets in each dimension, was chosen after a trial and error procedure in a specific range $[s_{\min}, s_{\max}]$, where $s_{\min}$ and $s_{\max}$ are the lower and upper bounds of the search space, i.e., the minimum and maximum number of fuzzy sets, respectively. In this work, the optimal value of s was found using the validation subset and an exhaustive search procedure in the range [5—20]. For comparison purposes, two different neural network architectures were tested. More specifically, a two hidden layer MLP network architecture trained with the Levenberg–Marquardt backpropagation algorithm was employed [38]. The optimal combination of hidden layer sizes was found using the validation subset, by performing an exhaustive trial and error procedure in the range [5—20] and [5—10] for the first and the second layer, respectively. The other rival method concerns a CNN model of two 2D convolutional layers with 20 5×5 and 50 3×3 filters in the first and the second layer, respectively, trained with stochastic gradient descent (SGD) algorithm, as proposed in [4]. Due to their stochastic nature, the MLP and CNN models produce a different result for each run, and thus, 30 different runs were performed in order to test the consistency of the results.

Except from testing a different network architecture, an investigation on the input features was also performed, by applying four different methods of neural network training. To be more specific, our proposed training dataset consists of the 194 input variables mentioned earlier, plus the 3 low-depth evaluation inputs. Variations of this dataset were created in order to observe the impact of the low-depth evaluation input on the final outcome, namely including all of the three evaluation inputs for depths 8, 4, and 2, including only the variables for depths 4 and 2, and finally, including none of these variables at all, altering the total amount of inputs to 197, 196, and 194, respectively. As far as architecture is concerned, the first method included RBF networks while the second and third method included MLPs and CNNs models; in both cases, training was performed using the aforementioned input dataset. The fourth method, also concerning MLP architecture, made use of the methodology found in [12], which was executed on our own dataset, so that the network training input dataset was constructed, in order to compare the resulting performances. In [12], two alternative board representations, referred to as “bitmap” and “algebraic,” are used, and MLP models were used, among others. The model giving the best performing case is the bitmap representation. 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 contain 768 values. As mentioned earlier, variations were created using the low-depth evaluation inputs resulting in datasets containing 768, 770, and 771 inputs, respectively.

As for the created datasets, when a chess engine evaluates a position, there are two types of possible outcome: an evaluation score or the number of moves until mate, if such a sequence is found. While processing, only middlegame positions lowers the possibility of the latter case, and positions with mating sequences may still occur. Such positions were then filtered out and were omitted from the final dataset, since they do not correspond to an actual evaluation score. The resulting dataset in this case consists of 80,425 middlegame positions.

In a second level of refinement, another filtering was performed for data having extremely high evaluation scores. In a typical chess game, evaluation scores tend not to be very high since the game is usually more or less balanced. For instance, evaluation scores over 6 indicate that an imbalance of two minor pieces and values over 9, which 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 datasets containing much more data in the region nearer to zero and less as values get higher. The threshold for the aforementioned filtering was set to 20, so as to examine the impact of this situation on the training of the neural network. The resulting dataset following this second type of filtering consisted of 79,874 middlegame positions.

4.2. Results and Discussion. The results for the created datasets in the two different cases of dataset filtering that have been described are depicted in Tables 3 and 4. The different input dataset variations regarding the low-depth evaluation inputs as described above are denoted as scenario 1, 2, and 3. Along with the values of the MAE and R^2 indicators for each scenario, for the validation and test sets, additional information for each method is presented, namely the computational time for network training, the size of the yielded network model in terms of nodes, and the number of selected fuzzy sets, applicable only in the RBF method.

A graphic representation of target versus predicted values for the best performance of the RBF network in the test subset for the two dataset filtering cases can be seen in Figures 5 and 6, respectively, where the line 1:1 is also included as reference.

Considering both tables, it can be observed that scenario 1, which contains all three supplementary evaluation input variables, outperforms the other scenarios in every method. On the other hand, scenario 2 that contains only the two lower depth variables performs slightly better than scenario 3 that merely contains static features, especially in the RBF method.

Observing the results of the over-20 evaluation filtering in comparison 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 earlier in this section. 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 diagram in Figure 5, 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 are denser, the necessary interpolation is much more likely to be efficient.

The RBF network model 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 network 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 network 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 networks and the CNN 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 network performs more or less the same (even slightly better in respect to R^2) as the bitmap MLP method does in scenario 2, where two low-depth evaluation variables assist its performance. Therefore, the RBF network is the one proposed as the best performing and most promising approach. Its success could be attributed to the effectiveness of the FM algorithm which has been shown to outperform other machine learning approaches in various problems [25].

Generally, we may notice that within both filtering approaches, all the tested networks 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, these methods seem to be affected to different extends by these additional inputs with the RBF method being the least dependent on them and the bitmap MLP being the most improved in their presence. As it can be seen, the improvement of the RBF network 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.

As far as a comparison between MLP and CNN models is concerned, one can see that the former outperforms the latter in all cases and scenarios in terms of mean and best metric values. The dominance of MLP models over CNNs was also verified by the work of Sabatelli et al., where they concluded that even though both architectures can be used as robust function approximators, MLP models lead the performance [4]. In fact, their results showed that MLP networks performed better or equally to CNNs in all the experiments, regardless the choice of bitmap or algebraic inputs in classification or regression tasks. Moreover, they discovered that simply increasing the depth of CNNs was not sufficient to improve performance, as it only resulted in increased complexity and longer training times. It should also be noted that in Sabatelli et al., CNNs were found to perform equal or worse when using bitmap inputs compared to algebraic ones, which is the reason that bitmap inputs were not used for CNNs in this work.

The lack of uniformity in the input dataset can be easily observed in Figure 5, where it can be seen that a big part of the data is gathered in the area close to zero. Another observation in Figure 6 is that as far as very large evaluation

Table 3: Aggregate presentation of indicators with mating evaluation filtering.

Scenario	Method	MAE		R^2		Training time	Nodes
		Testing	Validation	Testing	Validation	(s)	(fuzzy sets)
	RBF	**0.76**	**0.76**	0.50	0.52	7304	38219 (25)
1	CNN	0.94 (1.01 ± 0.06)	0.93 (1.01 ± 0.06)	0.31 (0.30 ± 0.01)	0.32 (0.31 ± 0.01)	3563	—
(3 low-depth evaluation inputs)	MLP	0.75 (0.82 ± 0.04)	0.75 (0.81 ± 0.04)	0.46 (0.42 ± 0.02)	0.52 (0.48 ± 0.03)	642	[20 10] (–)
	MLP-bitmap inputs	0.70 (0.78 ± 0.06)	0.71 (0.79 ± 0.06)	0.50 (0.47 ± 0.02)	0.50 (0.48 ± 0.02)	9480	[20 10] (–)
	RBF	**0.83**	**0.84**	0.43	0.45	12448	38229 (21)
2	CNN	0.98 (1.05 ± 0.07)	0.98 (1.04 ± 0.07)	0.25 (0.25 ± 0.007)	0.26 (0.25 ± 0.008)	3624	—
(2 low-depth evaluation inputs)	MLP	0.93 (1.00 ± 0.06)	0.93 (1.00 ± 0.06)	0.36 (0.30 ± 0.04)	0.38 (0.33 ± 0.03)	623	[20 10] (–)
	MLP-bitmap inputs	0.85 (1.04 ± 0.17)	0.85 (1.04 ± 0.17)	0.40 (0.35 ± 0.06)	0.42 (0.37 ± 0.04)	9342	[20 10] (–)
	RBF	**0.85**	**0.85**	0.41	0.43	5745	38238 (21)
3	CNN	1.01 (1.06 ± 0.06)	1.00 (1.06 ± 0.06)	0.23 (0.20 ± 0.007)	0.25 (0.24 ± 0.008)	3902	—
(no low-depth evaluation inputs)	MLP	0.96 (1.05 ± 0.04)	0.97 (1.04 ± 0.04)	0.32 (0.26 ± 0.03)	0.34 (0.30 ± 0.03)	598	[20 10] (–)
	MLP-bitmap inputs	0.94 (1.02 ± 0.06)	0.95 (1.02 ± 0.06)	0.35 (0.28 ± 0.05)	0.35 (0.29 ± 0.04)	9211	[20 10] (–)

The table depicts the best performance in the respective dataset in terms of MAE along with the mean value and standard deviation in parenthesis wherever applicable. The best result in terms of MAE in each scenario is marked with bold text.

Table 4: Aggregate presentation of indicators with over-20 evaluation filtering.

Scenario	Method	MAE		R^2		Training time	Nodes
		Testing	Validation	Testing	Validation	(s)	(fuzzy sets)
	RBF	**0.38**	**0.38**	0.78	0.80	5344	37983 (19)
1	CNN	0.6 (0.65 ± 0.05)	0.59 (0.65 ± 0.05)	0.54 (0.53 ± 0.023)	0.58 (0.56 ± 0.021)	3253	—
(3 low-depth evaluation inputs)	MLP	0.44 (0.47 ± 0.03)	0.44 (0.46 ± 0.03)	0.72 (0.70 ± 0.02)	0.73 (0.71 ± 0.02)	587	[20 10] (–)
	MLP-bitmap inputs	0.42 (0.45 ± 0.03)	0.43 (0.45 ± 0.03)	0.76 (0.74 ± 0.02)	0.76 (0.73 ± 0.02)	9185	[20 10] (–)
	RBF	**0.44**	**0.43**	0.72	0.73	5809	37995 (23)
2	CNN	0.69 (0.71 ± 0.02)	0.67 (0.69 ± 0.02)	0.5 (0.5 ± 0.015)	0.52 (0.52 ± 0.016)	3689	—
(2 low-depth evaluation inputs)	MLP	0.58 (0.60 ± 0.02)	0.57 (0.60 ± 0.02)	0.57 (0.54 ± 0.03)	0.59 (0.56 ± 0.03)	570	[20 10] (–)
	MLP-bitmap inputs	0.53 (0.56 ± 0.02)	0.54 (0.56 ± 0.02)	0.64 (0.60 ± 0.02)	0.62 (0.60 ± 0.01)	9049	[20 10] (–)
	RBF	**0.45**	**0.44**	0.69	0.70	8638	37970 (25)
3	CNN	0.71 (0.75 ± 0.02)	0.68 (0.71 ± 0.02)	0.49 (0.48 ± 0.024)	0.48 (0.48 ± 0.020)	3842	—
(no low-depth evaluation inputs)	MLP	0.62 (0.64 ± 0.02)	0.62 (0.64 ± 0.02)	0.51 (0.48 ± 0.02)	0.50 (0.48 ± 0.02)	561	[20 10] (–)
	MLP-bitmap inputs	0.59 (0.61 ± 0.02)	0.59 (0.61 ± 0.02)	0.56 (0.54 ± 0.01)	0.56 (0.54 ± 0.01)	8892	[20 10] (–)

The table depicts the best performance in the respective dataset in terms of MAE along with the mean value and standard deviation in parenthesis wherever applicable. The best result in terms of MAE in each scenario is marked with bold text.

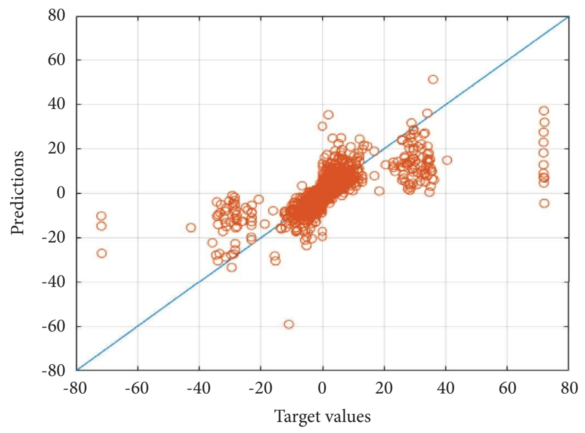


Figure 5: Target values vs. RBF predictions using the proposed features with mating evaluation filtering.

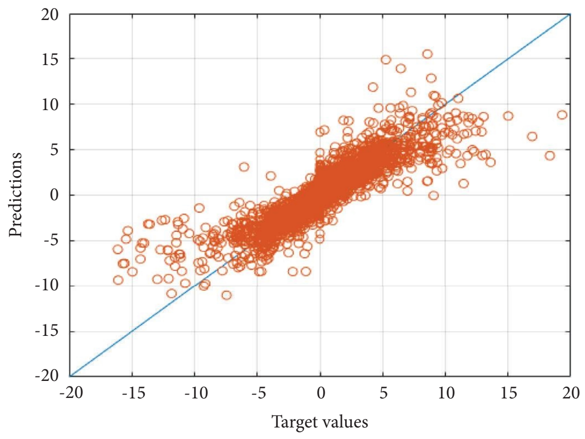


Figure 6: Target values vs. RBF predictions using the proposed features with over-20 evaluation filtering.

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. Apart from that, however, the overall arrangement of the data points tends to follow the expected shape of the ideal model line, as depicted in Figure 6.

An observation in Figure 5 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 result 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.

Finally, it should be pointed out that although the development 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.

5. Conclusions

A new approach to evaluating chess position scores based on RBF networks trained with the symmetric fuzzy means algorithm is proposed in this work. The proposed set of features for network training was obtained using the Stockfish 10 chess engine. In order to provide a dynamic aspect of the chess position in the training dataset, three more supplementary input variables are devised 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, resulting in three different scenarios. The available data points were collected and processed by over 1500 top-level games, resulting in a database of about 80,000 chess positions, and two filtering techniques were performed for each tested scenario, regarding mating evaluation and high evaluation scores. For comparison proposes, an MLP and a CNN network are included in the study. Moreover, we compare against a bitmap board representation described in [12]. The comparative results are in favor of the RBF network trained with our proposed set of features as inputs, regardless the choice of the scenario. The results showed that our proposed set of features significantly decreases the network training time, compared to existing techniques as confirmed by the experiments where it was concluded that an evaluation for several moves ahead (about 28 moves) can be made in about 7 milliseconds on average.

Future research plans include the creation of a larger database of chess positions implementing more handcrafted supplementary input variables in order to investigate to what extend could the modeling capabilities increase, the implementation of more sophisticated techniques, such as an RBF network trained with a nonsymmetrical fuzzy means algorithm [25, 34], or the development of a fully functional chess-playing application utilizing model predictive control (MPC) [32, 39] as a decision-making tool that the application would rely on in order to decide the next move.

Data Availability

The data used in this study contain moves from a collection of chess games. They can be downloaded from the following online chess databases: https://www.chessgames.com/ and https://www.365chess.com/.

Disclosure

Part of this article is based on the MSc Thesis of Mr. Kagkas, entitled "Chess Position Evaluation Using Neural Networks."

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The implementation of the doctoral thesis of D. Karamichailidou was cofinanced by Greece and the European Union (European Social Fund, ESF) through the Operational Programme "Human Resources Development, Education and Lifelong Learning" in the context of the act "Enhancing Human Resources Research Potential by Undertaking a Doctoral Research" and subaction 2: IKY Scholarship Programme for PhD candidates in the Greek Universities.

References

[1] M. Campbell, A. Hoane, and F.-H. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[2] B. Oshri and N. Khandwala, "Predicting Moves in Chess using Convolutional Neural Networks," in *Stanford CS231n Course Report*, Stanford University, Californica, CA, USA, 2015.

[3] E. David, N. S. Netanyahu, and L. Wolf, "Deepchess: end-to-end deep neural network for automatic learning in chess," 2017, http://arxiv.org/abs/1711.09667.

[4] M. Sabatelli, F. Bidoia, V. Codreanu, and M. A. Wiering, "Learning to evaluate chess positions with deep neural networks and limited lookahead," in *Proceedings of the ICPRAM*, Madeira, Portugal, January 2018.

[5] N. Hallam, H. S. Poh, and G. Kendall, "Using an evolutionary algorithm for the tuning of a chess evaluation function based on a dynamic boundary strategy," in *Proceedings of the 2006 IEEE Conference on Cybernetics and Intelligent Systems*, Bangkok, Thailand, June 2006.

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

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

[8] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice-Hall, New Jersey, NJ, USA, 2 edition, 1999.

[9] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal differences," *Machine Learning*, vol. 40, no. 3, pp. 243–263, 2000.

[10] S. Thrun, "Learning to play the game of chess," in *Advances in Neural Information Processing Systems*, G. Tesauro, D. Touretzky, and T. Leen, Eds., MIT Press, Cambridge, CA, USA, 1995, https://proceedings.neurips.cc/paper/1994/file/d7322ed717dedf1eb4e6e52a37ea7bcd-Paper.pdf.

[11] M. Lai, "Giraffe: using deep reinforcement learning to play chess," 2015, http://arxiv.org/abs/1509.01549.

[12] M. Sabatelli, W. Marco, and V. Codreanu, "Learning to play chess with minimal lookahead and deep value neural networks," *Philosophy*, 2017.

[13] D. Silver, T. Hubert, J. Schrittwieser et al., "A general reinforcement learning algorithm that masters chess, Shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[14] P. Lehana, S. Kulshrestha, N. Thakur, and P. Asthana, "Statistical analysis on result prediction in chess," *International Journal of Information Engineering and Electronic Business*, vol. 10, no. 4, pp. 25–32, 2018.

[15] A. Maesumi, "Playing chess with limited look ahead," *Artificial Intelligence*, 2020.

[16] V. Chole, N. R. Gote, S. Ujwane, P. Hedaoo, and A. Umare, "Design and development of game playingsystem in chess using machine learning," *International Research Journal of Engineering and Technology*, vol. 8, 2021.

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, A Bradford Book, Cambridge, MA, USA, 2018.

[18] J. Bo, S. Li, R. Geng, and J. Tong, "Design of No-Go game algorithm based on reinforcement learning," in *Proceedings of the 2019 IEEE International Conferences on Ubiquitous Computing and Communications (IUCC) and Data Science and Computational Intelligence (DSCI) and Smart Computing, Networking and Services (SmartCNS)*, pp. 712–716, Shenyang, China, October 2019.

[19] M. Li and W. Huang, "Research and implementation of Chinese chess game algorithm based on reinforcement learning," in *Proceedings of the 2020 5th International Conference on Control, Robotics and Cybernetics (CRC)*, pp. 81–86, Wuhan, China, October 2020.

[20] R. Drężewski and G. Wątor, "Chess as sequential data in a chess match outcome prediction using deep learning with various chessboard representations," *Procedia Computer Science*, vol. 192, pp. 1760–1769, 2021.

[21] T. Bertram, J. Fürnkranz, and M. Müller, "Supervised and reinforcement learning from observations in reconnaissance blind chess," in *Proceedings of the 2022 IEEE Conference on Games (CoG)*, pp. 608–611, Beijing, China, August 2022.

[22] A. Alexandridis, E. Chondrodima, N. Giannopoulos, and H. Sarimveis, "A fast and efficient method for training categorical radial basis function networks," *IEEE Transactions on*

Neural Networks and Learning Systems, vol. 28, no. 11, pp. 2831–2836, 2017.

[23] W. He, W. Zhao, and Y. Jiang, "Application of Q-learning and RBF network in Chinese chess game system," *IOP Conference Series: Materials Science and Engineering*, vol. 677, no. 2, Article ID 022101, 2019.

[24] D. Kagkas, "Chess position evaluation using neural networks," M.Sc. thesis, University of West Attica, Aigaleo, Greece, 2021.

[25] A. Alexandridis, E. Chondrodima, and H. Sarimveis, "Radial basis function network training using a nonsymmetric partition of the input space and particle swarm optimization," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 24, no. 2, pp. 219–230, 2013.

[26] T. Romstad and M. Costalba, *Stockfish Evaluation Guide*, Github, California, CA, USA, 2008.

[27] Tcec, "Tcec current and past Tcec tournaments," 2020, https://tcec-chess.com/.

[28] S. Russell, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Upper Saddle River, NJ, USA, 2 edition, 2003.

[29] Uci Protocol, "Uci protocol," 2021, http://wbec-ridderkerk.nl/html/UCIProtocol.html.

[30] D. Karamichailidou, S. Koletsios, and A. Alexandridis, "An RBF online learning scheme for non-stationary environments based on fuzzy means and givens rotations," *Neurocomputing*, vol. 501, pp. 370–386, 2022.

[31] D. Karamichailidou, A. Alexandridis, G. Anagnostopoulos, G. Syriopoulos, and O. Sekkas, "Modeling biogas production from anaerobic wastewater treatment plants using radial basis function networks and differential evolution," *Computers and Chemical Engineering*, vol. 157, Article ID 107629, 2022.

[32] M. Stogiannos, A. Alexandridis, and H. Sarimveis, "Model predictive control for systems with fast dynamics using inverse neural models," *ISA Transactions*, vol. 72, pp. 161–177, 2018.

[33] M. Papadimitrakis and A. Alexandridis, "Active vehicle suspension control using road preview model predictive control and radial basis function networks," *Applied Soft Computing*, vol. 120, Article ID 108646, 2022.

[34] A. Alexandridis, E. Chondrodima, and H. Sarimveis, "Cooperative learning for radial basis function networks using particle swarm optimization," *Applied Soft Computing*, vol. 49, pp. 485–497, 2016.

[35] C. Darken and J. Moody, "Fast adaptive K-means clustering: some empirical results," in *Proceedings of the IEEE International Joint Conference on Neural Networks*, pp. 233–238, San Diego, CA, USA, January 1990.

[36] A. Alexandridis and E. Chondrodima, "A medical diagnostic tool based on radial basis function classifiers and evolutionary simulated annealing," *Journal of Biomedical Informatics*, vol. 49, pp. 61–72, 2014.

[37] E. Chondrodima, H. Georgiou, N. Pelekis, and Y. Theodoridis, "Particle Swarm Optimization and RBF Neural Networks for public transport arrival time prediction using GTFS data," *International Journal of Information Management Data Insights*, vol. 2, Article ID 100086, 2022.

[38] M. Hagan and M. Menhaj, "Training feedforward networks with the marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

[39] A. Alexandridis, H. Sarimveis, and K. Ninos, "A Radial Basis Function network training algorithm using a non-symmetric partition of the input space–application to a Model Predictive Control configuration," *Advances in Engineering Software*, vol. 42, no. 10, pp. 830–837, 2011.