Assessing the Positional Values of Chess Pieces by Tuning Neural Networks' Weights with an Evolutionary Algorithm

Eduardo Vázquez-Fernández, Carlos A. Coello Coello and Feliú D. Sagols Troncoso

***Abstract*—Finding a method that can automatically set the weights of the evaluation function of a chess engine is an important research topic, since the use of manual settings requires a significant amount of time and expertise, which are not always available. The specialized literature reports several works in which the weights of the positional values of the chess pieces are evolved based on values stored in tables. Here, however, we propose to use a neural network architecture to obtain the positional values of the chess pieces based on specific features of each position. The neural networks that we adopt for this sake are relatively small and we argue that they constitute a robust way of obtaining the positional values of the chess pieces. The adjustment of weights of such neural networks was done through the use of an evolutionary algorithm, producing an increase of 433 points of the ranking of our chess engine (from 1745 to 2178 points, reaching a value close to that of a chessmaster).**

I. Introduction

In 1947, Alan Turing [22] designed the first program to play chess, and two years later, Claude Shannon [19] proposed two different strategies to implement a chess program. The first was the "Type A" strategy, which considered all possible moves to a fixed depth of the search tree, and the second was the "Type B" strategy, which used chess knowledge to explore the most promising lines to a greater depth. In the $1950s$, chess programs could only play at a very basic level, and by the end of the $1960s$, chess programs could occasionally defeat amateur chess players. The development of chess programs during the $1970s$ was characterized by the use of heuristics to choose the best moves in the search tree on the fly, and more specialized hardware. In the mid-1980s, chess programs based on microprocessors started to win tournaments involving both human players and other chess programs based on supercomputers. In the $1990s$, chess programs based on personal computers began challenging grandmasters. In 1997, the IBM computer *Deep Blue* defeated world chess champion Garry Kasparov with a final score of 3.5 to 2.5. Deep blue was capable of evaluating 200 million positions per second.

In general, the strength of a chess engine is determined by the efficiency of the move generator, the depth reached along the search tree and the function to evaluate positions. Probably the last component is the most important one. In this work, we compute the positional values of the pieces through unsupervised neural networks whose weights are adjusted using an evolutionary algorithm.

The remainder of this paper is organized as follows. In Section II, we review the relevant previous related work about adjusting the evaluation function of a chess engine. Our chess engine and the evaluation function adopted for the purposes of this paper are described in Section III. In Section IV, we show the methodology that we adopted to obtain the positional values of the pieces. Our experimental results are reported in Section VI. Finally, in Section VII, we provide our conclusions and some possible paths for future work.

Eduardo Vázquez-Fernández is with CINVESTAV-IPN (Evolutionary Computation Group), Departamento de Computación, Av. IPN No. 2508, Col. San Pedro Zacatenco, México, D.F., 07360, MEXICO (email: eduardovf@hotmail.com).

Carlos A. Coello Coello is with CINVESTAV-IPN (Evolutionary Computation Group), Departamento de Computación, Av. IPN 2508, Col. San Pedro Zacatenco, México, D.F., 07360, MEXICO (email: ccoello@cs.cinvestav.mx).

Feliú D. Sagols Troncoso is with CINVESTAV-IPN, Departamento de Matemáticas, Av. IPN No. 2508, Col. San Pedro Zacatenco, México, D.F., 07360, MEXICO (email: fsagols@math.cinvestav.edu.mx).

II. Previous related work

The manual adjustment of the weights used by the evaluation function of a chess engine is a task that usually requires a significant amount of time. This has motivated the development of automated methods for this task, from which the main ones are briefly described next.

Thrun developed in 1995 the program *NeuroChess* [21] which learns to play chess from the final outcomes of games and uses artificial neural networks to adjust the weights of its evaluation function. Fogel et al. [7] used a coevolutionary strategy in which a set of virtual players confronted one another several times allowing the survival of only the most successful virtual players. This program adjusted the weights of the piece-square tables (values stored in tables) and the weights of three neural networks to improve its performance above the master level (around 400 rating points). This program was evolved along 7462 generations by Fogel et al. [8], reaching a rating of 2650. Kendall and Whitwell [14] proposed a method for tuning the weights of the evaluation function of a chess engine using an evolutionary algorithm. They showed how the outcome of the game (win, loss or draw) can be used to develop such an evaluation function. Nasreddine et al. [17] proposed an evolutionary algorithm to adjust the weights of the evaluation function of a chess engine. The characteristic of this method is that the weight interval boundaries are dynamic. Bošković et al. [4], proposed a method based on differential evolution to adjust the material values of the chess pieces, being able to reproduce their "theoretical" values [19]. Hauptman and Sipper [11] solved mate-in-N problems without using the alpha-beta algorithm. David-Tabibi et al. [6] built a grandmaster-level chess program based on supervised and unsupervised learning. In this case, chess was learnt only from a database of games played by humans.

The main difference between our approach and other methods lies on the way in which the positional values of the chess pieces are obtained. In other works (see for example [7], [8], [3]) the positional values of the pieces are represented by values stored in tables. Positional values of the pieces defined in tables have the disadvantage of being static and do not depend directly on the characteristics of the position. For example, let's imagine that a bishop on $d5$ always has the same value; this is generally incorrect because its positional value depends, among other things, from the pawn's structure [18], [10]. In our method, however, the positional values are dynamic and depend on the characteristics of the position such as location, mobility, center control and so on.

III. Our chess engine

To carry out our experiments, we developed a chess engine with the following features: alpha-beta search algorithm [15], [16] with iterative deepening, stabilization of positions through the quiescence algorithm [2] (which considers the exchange of material and checks to the king), hash tables [23], [1] and move generator through the 0×88 hexadecimal method.[1]

The **evaluation function** used to determine (in a heuristic way) the relative value of a position with respect to one side (white or black pieces) is given by the following expression:

$$f = \sum_{i=1}^{r} m_i + \sum_{i=1}^{q} c_i \times p_i \quad (1)$$

where:
r is the number of pieces in the side under evaluation without considering the king.
q is the number of pieces in the side under evaluation.
m_i is the material value of the piece i.
c_i is the adjustment of the weight p_i ($c_i = 0.5 \times m_i$).
p_i is the positional value of the piece i. $p_i \in [0, 1]$ (0 represents the worst adjustment of weights and 1 represents a best adjustment of weights).

The **material value** of a piece is static, and it is 100, 300, 300, 500 and 900 for the pawn, knight, bishop, rook and queen, respectively; these values agree with the "theoretical" values considered for the chess pieces [19]. The **positional value** of a piece is dynamic, and it depends on many factors such as location, mobility, center control and so on.

IV. Our proposed methodology

A. Neural network architecture

Our architecture is composed of six neural networks that we use to calculate the positional values of the chess pieces of our chess engine, as defined in equation (1). Each neural network was fully connected and consisted of four nodes in the input layer,[2] nine nodes in the hidden layer and one node in the output layer. The decision to use three layers was based on the demonstration of Hecht-Nielsen [12], which established that any function can be approximated by a three-layer neural network. The decision to use nine nodes in the hidden layer was based on Kolgomorov's theorem [20] which established that the number of nodes in the hidden layer should be at least $(2i + 1)$, where i denotes the number of nodes in the input layer. As part of our future work, we intend to evolve the number of hidden units. The hidden nodes used a sigmoid defined by the logistic function $f(y_j) = 1/(1 + exp(-y_j))$, where y_j was the product of the incoming features from the chessboard and the associated weights between the input and hidden nodes, offset by each hidden node's bias term, i.e. $y_j = \sum_{i=1}^{4} f_i \times W_{ij} + \theta_j$, where f_i is the incoming feature i, W_{ij} is the weight between the node i and node j, and θ_j is the bias term of the node j. The output node also used the logistic function and its value is in $[0, 1]$, where 0 and 1 denotes the worst and the best adjustment of weights, respectively.

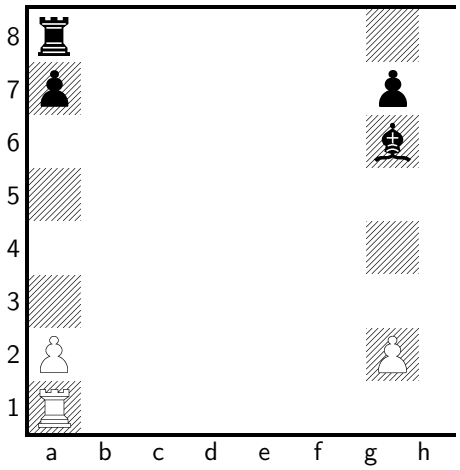

Fig. 1. Example diagram to illustrate feature extraction.

B. King's positional value

Our architecture used a neural network to obtain the positional value of the king. Its four input signals correspond to the features that we considered important to get the positional value of the king.

- *Attacking material*. It refers to the material value of the pieces that are attacking the opposite king. By this, we mean those pieces whose movements act on its opposite king's square or on its opposite king's adjacent squares (in this case, the movements of the pieces can jump to other pieces,[3] regardless of the pawns). For example, in Figure 1, the queen on $f6$ attacks the king on $g1$; therefore, the attacking material corresponding to the white king is 900.

[1] `http://www.cis.uab.edu/hyatt/boardrep.html`

[2] The fact that all neural networks have four nodes in the input layer is mere coincidence, and this number can vary depending on the characteristics chosen to obtain the positional value of a chess piece.

[3] Jumping other pieces allows to detect indirect attacks. For example, in Figure 1 the bishop on $b6$ will attack the white king when the pawn on $d4$ moves to another square.

- *Defending material*. It refers to the material value of the pieces that are defending its king. By this, we mean those pieces whose movements act on its king's square or on its king's adjacent squares (also, the movements of the pieces can jump to other pieces, regardless of the pawns). For example, in Figure 1, the queen on $g3$, the rook on $a1$, the rook on $f1$, the bishop on $f4$ and the knight on $d2$ all defend the white king; therefore, the defending material corresponding to the white king is 2500 ($900 + 500 + 500 + 300 + 300$).
- *Castling*. It is a binary value. It is one if and only if the king is castled. In Figure 1 this value is one for the white king.
- *Pawns*. It is the number of pawns located on its king's adjacent squares. In Figure 1 this value is two for the white king.

C. Queen's positional value

Our architecture used a neural network to obtain the positional value of the queen. Its four input signals correspond to the features that we considered important to obtain the positional value of the queen.

- *Queen mobility*. It is the number of movements of the queen. In Figure 1 this value is ten for the white queen.
- *Column type*. It is 0 if on the queen's column there are no pawns, it is 1 if on the queen's column there are adversary pawns and the queen is on front of its pawns (if any), and it is 2 if on the queen's column there are pawns at both sides and the queen is behind any of its pawns. In Figure 1 this value is 1 for the white queen.
- *Row*. It refers to the row occupied by the queen. In Figure 1 this value is three for the white queen.
- *Column*. It refers to the column occupied by the queen. In Figure 1 this value is seven for the white queen.

D. Rook's positional value

Our architecture used a neural network to obtain the positional value of the rook. Its four input signals correspond to the features that we considered important to obtain the positional value of the rook.

- *Rook mobility*. It is the number of movements of the rook. In Figure 1 this value is four for the rook on $a1$.
- *Column type*. See the definition of the column type for the queen. In Figure 1 this value is 2 and 1 for the rooks on $a1$ and $f1$, respectively.
- *Seventh row*. It is a binary value. It is 1 if and only if the rook is on the seventh row. In Figure 1 this value is 0 for the rook on $a1$.
- *Seventh row folded*. It is a binary value. It is 1 if and only if there are at least two rooks on the seventh row. In Figure 1 this value is 0 for the rook on $a1$.

E. Bishop's positional value

Our architecture used a neural network to obtain the positional value of the bishop. Its four input signals correspond to the features that we considered important to obtain the positional value of the bishop.

TABLE I
WEIGHTS OF BLACK PAWNS THAN OBSTRUCT THE BLACK BISHOP'S MOVEMENT.

8	0	0	0	0	0	0	0	0
7	2	4	4	8	8	4	4	2
6	2	4	8	16	16	8	4	2
5	2	4	12	24	24	12	4	2
4	2	4	4	4	4	4	4	2
3	2	2	2	2	2	2	2	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

TABLE II
WEIGHTS OF THE WHITE PAWNS THAN OBSTRUCT THE BLACK BISHOP'S MOVEMENT.

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	1	1	1	1	1	1	0
5	0	1	2	2	2	2	1	0
4	0	1	2	2	2	2	1	0
3	0	1	1	1	1	1	1	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

- *Bishop mobility*. It is the number of movements of the bishop. In Figure 1 this value is five for the bishop on $f4$.
- *Pawn's mobility*. It is the number of movements of its pawns which obstruct the movement of the bishop. In Figure 1 this value is five for the bishop on $b6$.
- *Ahead*. It is the number of pawns which are in front of its bishop and obstructing its movement. In Figure 1 this value is one for bishop on $b6$, because the pawn on $d4$ obstructs its movement.
- *Weight*. Any chess expert will notice in Figure 1 that the pawn on $d4$ obstructs more the movement of the bishop on $b6$ than the pawn on $a7$. Each square on the board is assigned a numeric value that reflects the degree of obstruction of a pawn on the bishop's movement. Table I shows the weights of black pawns than obstruct the black bishop's movement (these values were taken from [10]), and Table II shows the weights of the white pawns that obstruct the black bishop's movement (these values have been assigned by an expert in chess). The weights of the white pawns and the black pawns that obstruct the white bishop's movement are the mirror of Tables I and II, respectively. In Figure 1, the weight for the bishop on $b6$ is 26 ($2 + 4 + 16 + 4$).

F. Knight's positional value

Our architecture used a neural network to obtain the positional value of the knight. Its four input signals correspond to the features that we considered important to obtain the

positional value of the knight.

- *Knight mobility*. It is the number of movements of the knight. In Figure 1 this value is four for the knight on $d2$.
- *Periphery*. It is a binary value. It is 1 if and only if the knight is on the periphery of the board (first row, eighth row, first column or eight column). In Figure 1 this value is 0 for the knight on $d2$.
- *Supported*. It is a binary value. It is 1 if and only if the knight is supported by one of its pawns. In Figure 1 this value is 0 for the knight on $d2$.
- *Operations base*. It is a binary value. It is 1 if and only if the knight is on an *operations base*. A knight is on an *operations base* if it cannot be evicted from its position by an opponent pawn. In Figure 1 this value is 0 for the knight on $d2$.

G. Pawn's positional value

Our architecture used a neural network to obtain the positional value of the pawn. Its four input signals correspond to the features that we considered important to obtain the positional value of the pawn.

- *Doubled*. It is a binary value. It is 1 if and only if there are at least two pawns located in the same column. In Figure 1 this value is 1 for the pawn on $d6$.
- *Isolated*. It is a binary value. It is 1 if and only if a pawn cannot be defended by another pawn. In Figure 1 this value is 0 for the pawn on $d6$.
- *Central*. It is a binary value. It is 1 if and only if the pawn is on any of the following squares: $c4$, $c5$, $d4$, $d5$, $e4$, $e5$, $f4$ or $f5$. In Figure 1 this value is 1 for the pawn on $e4$.
- *Past*. It is a binary value. It is 1 if and only if the pawn cannot be stopped by an opponent pawn. In Figure 1 this value is 0 for the pawn on $e4$.

It is worth noticing that the values obtained with our neural network architecture are conceived to correspond to the chess pieces' positional values of a mid-game.

H. Use of an evolutionary algorithm

Figure 2 outlines the evolutionary algorithm adopted to adjust the neural networks' weights in order to compute the pieces' positional values. The first module, called "initialize population", assigns initial random weights to the neural networks and the weights of the pawns that obstruct the bishop's mobility. The features of the position (inputs of the neural network) are obtained in the module "Features extraction".

The module "Play tournament" coordinates a tournament between n virtual players (in our case $n = 20$). Each virtual player is allowed to play $n/2$ games with randomly chosen opponents. The side (either black or white) is also chosen at random. Games are executed until one of the virtual players receives checkmate or a draw condition arises. Depending on the outcome of the game, a virtual player obtains one point, half a point or zero points for a win, tie or loss, respectively. Draw conditions are given by the rule of 50 moves (after a pawn's move there are 50 moves to pose a checkmate to the opponent), by the third repetition of the same position and by the lack of victory conditions (e.g., in the fight of a king and a bishop against a king). This module uses the chess engine described in Section III.

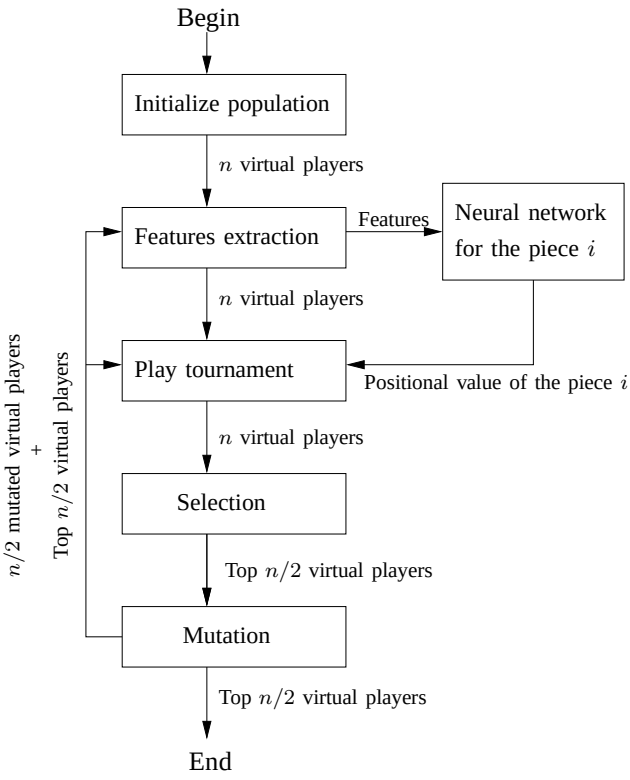

Fig. 2. Flowchart of the evolutionary algorithm adopted in this work.

After finishing the tournament, the "Selection" module chooses the $n/2$ virtual players having the highest number of points, and in the module "Mutation" these virtual players are mutated to generate the remaining $n/2$ virtual players. Finally, the evolutionary algorithm (based on evolutionary programming [9]), continues running for 50 generations.

V. Experimental design

The experiments were carried out on a PC with a 64-bits architecture, having two cores running at 2.8 GHz each and 3 GBytes in RAM. The programs were compiled using *g++* in the OpenSuse 11.1 operating system. For the experiments reported next, we used the opening book *Olympiad.abk* both for the virtual players and for the chess engine Rybka 2.3.2a.

A. Initialization

The initial population of our evolutionary algorithm consisted of $n = 20$ (10 parents and 10 offspring in subsequent generations) virtual players whose weights were randomly initialized within their allowable bounds using a uniform distribution. The weights and biases of the neural networks were initialized in the range $[-15, 15]$ and the weights of the pawns which obstruct the bishop's mobility were initialized in

the range $[0, 20]$ (we carried out different experiments, and we found that the ranges of these weights fall into these intervals).

B. Mutation

One offspring was created from each surviving parent by mutating all weights and biases by adding a Gaussian random variable with zero mean and a standard deviation of 0.05 as Chellapilla and Fogel did in [5]. If, after mutating a weight, its value falls outside the range, this value is re-set to the nearest extreme of its range.

VI. Experimental Results

A. Experiment A

This experiment consisted of performing ten runs, and in each of them we had 20 virtual players that were evolved during 50 generations. The weights of the virtual players were randomly initialized within the allowable bounds with a different seed for each run. At the end of each run, we carried out 200 games between the best virtual player in generation 50 and the best virtual player in generation 0, Table III shows these results. For example, in run 1 the best player in generation 50 won 180, drew 14 and lost 6 games against the best player in generation 0 (the percentage of games won by the best player in generation 50 was 93.50%). The best result corresponds to the third run, in which the best virtual player in generation 50 won 185, drew 12 and lost 3 games against the best player in generation 0 (the percentage of games won by the best player in generation 50 was 95.50%). In this experiment we used a search depth of four plies (1 ply corresponds to the movement of one side),

TABLE III
Number of games won, drawn and lost for the best virtual player in generation 50 against the best virtual player in generation 0.

Run	Wins	Draws	Losses	Wins%
1	180	14	6	93.50%
2	171	26	3	92.00%
3	185	12	3	95.50%
4	169	28	3	91.50%
5	174	25	1	93.25%
6	176	19	5	92.75%
7	182	16	2	95.00%
8	183	15	2	95.25%
9	178	18	4	93.50%
10	168	28	4	91.00%

B. Experiment B

In this experiment, the best virtual player in generation 0, was called $player_0$ and played 60 games against the chess engine Rybka 2.3.2a using each of the following ratings: 2300, 2100, 1900 and 1700. The histogram of results is shown in Figure 3. For example, $player_0$ won, drew and lost 0, 3 and 57 respectively against Rybka 2.3.2a at 2300 rating points; $player_0$ won, drew and lost 4, 6 and 50 respectively against Rybka 2.3.2a at 2100 rating points. The same experiment was carried out with the best virtual player for the ten runs in Table III. This virtual player was called $player_{50}$ and corresponds to the third run in this table. The results against the chess engine Rybka 2.3.2a are shown in Figure 4. In this Figure we can see that $player_{50}$ won, drew and lost 14, 10 and 36 respectively against Rybka 2.3.2a at 2300 rating points; $player_{50}$ won, drew and lost 26, 22 and 12 respectively against Rybka 2.3.2a at 2100 rating points.

Based on these played games, we used the Bayeselo tool[4] to estimate the ratings of players using a minorization-maximization algorithm [13]. The obtained ratings are shown in Table IV. In this table we can see that the rating for the virtual player $player_0$ was 1745, and the rating for virtual player $player_{50}$ was 2178, representing an increase of 433 rating points between the non-evolved and the evolved virtual players after 50 generations for the third run of Table III (2178 ratings points is a value close to a chessmaster level [7]).

In this experiment we used a search depth of six plies for the chess engine Rybka2.3.2a, as well as for $player_0$ and $player_{50}$.

It is worth noticing that Thrun [21] employed one neural network with 175 input nodes, 165 hidden nodes and 175 output nodes within his program *NeuroChess*. NeuroChess successfully won 11% of the games versus the program *GnuChess* (about 2300 rating points), and our chess program won 31.6% of the games versus Rybka 2.3.2a at 2300 rating points. In another previous related work, Fogel et al. [7] employed three neural networks, each one having 16 input nodes, 10 hidden nodes and 1 output node. The strength of their program was about 2550 rating points.

TABLE IV
Ratings on the third run against Rybka2.3.2a.

Rank	Name	Elo	+	-	Games	Score (%)	Oppo.	Draws (%)
1	$Rybka_{2300}$	2309	64	59	120	83%	1961	11%
2	$Player_{50}$	2178	38	37	240	69%	1997	18%
3	$Rybka_{2100}$	2097	51	50	120	63%	1961	23%
4	$Rybka_{1900}$	1883	51	52	120	41%	1961	16%
5	$Player_0$	1745	40	41	240	25%	1997	12%
6	$Rybka_{1700}$	1699	56	60	120	25%	1961	9%

In the previous experiments each virtual player was allowed to play $n/2$ games with randomly chosen opponents. Also noteworthy that these experiments were repeated allowing each virtual player to play against the remaining virtual players (in total $n-1$ games), and in this case, the best virtual player with $n-1$ games was only three points higher than the best virtual player with $n/2$ games.

VII. Conclusions and future work

In this paper we introduced an approach in which we obtained the positional values of chess pieces through a neural network architecture based on unsupervised learning. The weights of this neural network architecture were evolved using

[4]http://remi.coulom.free.fr/Bayesian-Elo/

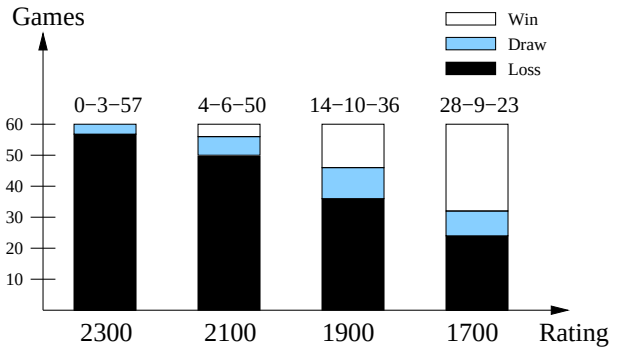

Fig. 3. Histogram of wins, draws and losses for the best virtual player at generation 0 ($player0$) against Rybka 2.3.2a.

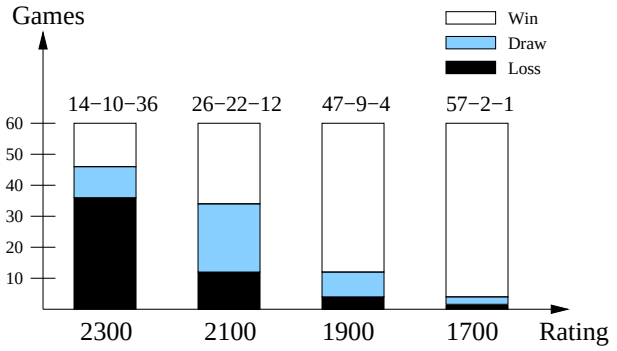

Fig. 4. Histogram of wins, draws and losses for the best virtual player at generation 50 ($player_{50}$) against Rybka 2.3.2a.

evolutionary programming. In some previous related work, these values had been evolved with "piece-square tables". The disadvantage of these methods is that the positional values of the pieces do not depend directly on the characteristics of the position. In contrast, our method by taking into account these characteristics to obtain these positional values.

We also believe that our architecture constitutes an alternative that is easy to implement and that is scalable (the programmer only needs to add the inputs to the corresponding neural network that is relevant to evaluate the chess piece of interest). After adjusting the neural network's weights using our evolutionary algorithm, we increased the rating of our chess engine in 433 points (from 1745 to 2178). It is expected that by adding the number of inputs to the neural networks, the positional value of the chess pieces will be assessed in a more precise manner and, consequently, the strength of the chess engine will be increased (and certainly this is part of future work). We are also interested in evolving the parameter c_i in our evaluation function and in evolving the number of nodes in the hidden layers of our neural network architecture in order to obtain more accurate assessments of the positional values of the chess pieces. It is worth noticing that the evolution of the parameter c_i will define the style of play of our chess engine.

Acknowledgements

The first author acknowledges support from CINVESTAV-IPN, CONACyT and the National Polytechnical Institute (IPN) to pursue graduate studies at the Computer Science Department of CINVESTAV-IPN. The second author acknowledges support from CONACyT project no. 103570.

References

[1] D. Beal and M. C. Smith. Multiple probes of transposition tables. *ICCA Journal*, 19(4):227–233, 1996.
[2] D. F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, April 1990.
[3] D. F. Beal and M. C. Smith. Learning piece-square values using temporal differences. *Journal of The International Computer Chess Association*, 22(4):223–235, December 1999.
[4] B. Bošković, S. Greiner, J. Brest, and V. Žumer. A differential evolution for the tuning of a chess evaluation function. In *2006 IEEE Congress on Evolutionary Computation*, pages 1851–1856, Vancouver, BC, Canada, July 16-21 2006. IEEE Press.
[5] K. Chellapilla and D. Fogel. Evolution, neural networks, games, and intelligence. *Proceedings of the IEEE*, 87(9):1471 –1496, Sept. 1999.
[6] O. David-Tabibi, H. J. van den Herik, M. Koppel, and N. S. Netanyahu. Simulating human grandmasters: evolution and coevolution of evaluation functions. In *GECCO'09*, pages 1483–1490, 2009.
[7] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 92(12):1947–1954, 2004.
[8] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. Further evolution of a self-learning chess program. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05)*, pages 73–77, Essex, UK, April 4-6 2005. IEEE Press.
[9] L. J. Fogel. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, 1966.
[10] M. Guid, M. Možina, J. Krivec, A. Sadikov, and I. Bratko. Learning positional features for annotating chess games: A case study. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 192–204. Springer. Lecture Notes in Computer Sciences, Vol. 5131, Heidelberg, Germany, 2008.
[11] A. Hauptman and M. Sipper. Evolution of an efficient search algorithm for the mate-in-n problem in chess. In *Proceedings of the 10th European conference on Genetic programming*, EuroGP'07, pages 78–89, Berlin, Heidelberg, 2007. Springer-Verlag.
[12] R. Hecht-Nielsen. *Neurocomputing / Robert Hecht-Nielsen*. Addison-Wesley Pub. Co., Reading, Mass., 1990.
[13] R. Hunter. Mm algorithms for generalized bradley-terry models. *The Annals of Statistics*, 32:2004, 2004.
[14] G. Kendall and G. Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, volume 2, pages 995–1002. IEEE Press, May 2001.
[15] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
[16] T. A. Marsland and M. Campbell. A survey of enhancements to the alpha-beta algorithm. In *Proceedings of the ACM '81 conference*, ACM '81, pages 109–114, New York, NY, USA, 1981. ACM.
[17] H. Nasreddine, H. 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 2006 IEEE international Conference on Cybernetics and Intelligent Systems (CIS'2006)*, pages 1–6. IEEE Press, 2006.
[18] L. Pachman. *Estrategia moderna en ajedrez*. Colección escaques, 1972.
[19] C. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 7(41):256–275, 1950.
[20] K. Swingler. *Applying neural networks: A practical guide*. Academic Press, London, 1996.
[21] S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems (NIPS) 7*, pages 1069–1076, Cambridge, MA, 1995. MIT Press.
[22] A. Turing. *Digital Computers Applied to Games, of Faster than Thought*, chapter 25, pages 286–310. Pitman, 1953.
[23] A. Zobrist. A new hashing method with application for game playing. Technical Report 88, The University of Wisconsin, Madison WI, USA, 1970. Reprinted (1990) in ICCA Journal, Vol. 13, No. 2, pp. 69-73.