Machine learning methods in game of chess implementation

P. WÓJCIK[1)], J. WIŚNIEWSKA[2)]

p.wojcik95@outlook.com, joanna.wisniewska@wat.edu.pl

[1)] Narwik St. 17A/3, 01-471 Warsaw, Poland
[2)] Military University of Technology, Faculty of Cybernetics
Kaliskiego St. 2, 00-908 Warsaw, Poland

The following work presents methods of using machine learning to teach a computer to play chess. The first method is based on using records of games played by highly ranked players. The second method is based on the Monte Carlo Tree Search algorithm and reinforcement learning.

Keywords: chess, machine learning, neural networks.

DOI: 10.5604/01.3001.0015.9191

1. Introduction

Chess is one of the most popular board games in the world. The multitude of tactics and strategies reduces the repetitiveness of the games played, making them unique. The "royal game" has been an object of interest for programmers since the beginning of the existence of computers [1]. At first programs were implemented to play the game for two people, later programs were able to compete with a man. Due to the large number of possible moves the chess engines of those days were not able to match the best chess players in the world. In recent years, machine learning has played an important role in the development of chess engines. Classical engines based on minimax algorithm had to give way to engines based on machine learning.

This paper presents two approaches for teaching a computer to play chess. The first method is based on a neural network model [2, 6, 8] that will be able to return the best move based on data representing a chess position [3]. The second method involves a Monte Carlo Tree Search algorithm and a neural network model that assists in the process of selecting branches in the tree and evaluates the chess position [5].

2. Approach based on the use of played games

One method of machine learning for teaching a computer to play chess is to use a collection of chess positions and moves made by chess masters. In this method, the moves made from the dataset are assumed to be optimal.

To teach an artificial neural network, a set of input data and a set of output data are needed. In the method presented here, the input data set is the numbers which are the representation of the setting of the pieces on the chessboard. The online chess service *Lichess* provides monthly data from the games played in the last month. Each game contains metadata concerning, among others, players' ranking and the course of the game itself in the form of chess algebraic notation. In the learning process one should use games of players with high ranking in order to minimize the number of weak moves. The position is represented as a sequence of 384 numbers. This sequence consists of 6 strings of 64 numbers for each type of chess piece. Each position in the sequence relates to a specific field on the chessboard. The number in the first position in the sequence concerns the field A8 (upper left corner of the board), and the number in the last position concerns the field H1 (lower right corner of the board from the perspective of the player using white chess pieces, according to the Forsyth–Edwards notation [9]). The numbers in the sequence take 3 values:

- 1 – if there is a particular type of chess piece on the field belonging to the side making the move,
- 0 – if there is no piece on the field or a piece of a different type,
- –1 – if there is a piece of a particular type on the field belonging to the opposite side.

If black makes a move in a given position, the numbers in the sequence refer to the fields in reverse order, i.e. starting with H1 and ending with A8. The output is a sequence of zeros and

a single one at the position corresponding to the field. The position in the sequence corresponds to the fields in the order from A8 to H1. The solution uses two types of models; in one the output is for the field from which the player made the move, in the other it is for the field to which the move was made.

In each position without verifying that the move is legal, and ignoring the various possibilities for promoting the pawn, there are 4096 possible moves, 64 possibilities from which field the move is made, and – for each of these options – 64 possibilities of the field to which the move is made. Therefore, this problem requires the use of several models. The first model is responsible for choosing which chess piece is moved. The next six models (a different model for each piece type) will indicate on which field it is best to place the piece of a given type. The move considered the best will be the one with the highest value of the product of "from which field" and, depending on what kind of a piece is on that field, the value from a given model "to which field".

The online chess service *Lichess* provides the run of games recorded in chess algebraic notation. Data preparation requires the implementation of a converter that converts the move notation into two values: from which field the move was made and to which field the move was made. In the implementation, fields are specified by numeric values from 0 to 63, starting at field A8 and ending at H1. Movements will be stored in the form ("from which field", "to which field"). The determination of the numerical value of the field designation is obtained from the formula:

$$f(w, k) = 8 * (w - 1) + a - 97 \qquad (1)$$

The check "+" and checkmate "#" notations are omitted because they do not affect the determination of fields. Moves in notation can be divided into three categories, given below:

- Figure moves – notation begins with one of the letters: K (king), Q (queen), R (rook), B (bishop), N (knight). In order to convert a notation into a move, all legal moves for all the pieces of a given type must be generated. The field occupied by the figure making the move to a field will be the value "from which field" in the notation. If a column and/or row description is also given in the notation, then figures not in that row and/or column are excluded. The "to which field" value is obtained by substituting the field notation contained in the move notation;
- Castling – marked "O-O" or "O-O-O", Information about the side making the move is sufficient to determine the fields associated with the move. For whites, short castling is move (60, 62), long castling (60, 58), while for black, short castling is (4, 6) and long castling is (4, 2);
- Pawn moves – the notation doesn't start with a letter characterizing the figures. If the pawn move is not capturing, then the first two symbols indicate the value of "to which field". The value "from which field" is obtained by checking which pawn can move there. If the movement of a pawn is a capture, then the movement notation also contains the column which the pawn moves from. The value "from which field" is obtained from the field from which the pawn in that column can move to the field in the notation. Additionally, when a pawn moves to the last row, the move notation contains information about the piece the pawn will change into. When making a move, the promotion option is taken into account.

After converting the notation into two numerical values of the move and the potential option of the pawn promotion, it is possible to make a move on the chessboard. Before the move is made, the current state on the chessboard is converted into input data. In the model responsible for selecting a chess piece, the output consists of 64 zeros and a single one in the place corresponding to the value "from which field". Then, depending on the type of chess piece that the move is made with, the input data for this model is identical to that of the piece selection model. The output data, on the other hand, has the same format as in the piece selection model, except that a single one is in the place corresponding to the value "to which field".

Each of the seven models mentioned earlier has the same structure. The models consist of three hidden layers, one with 32 neurons and two with 128 neurons. In the middle layers, ReLU (Rectified Linear Units) activation function was used, with function form: $f(x) = \max(0, x)$. In the output layer, the *softmax* activation function was used. This function is used in classification tasks where classes are mutually exclusive.

The model responsible for selecting the chess piece was learned with one million items, and the other models were learned with data

from about 150 to 200 thousand items. The learning took 10 epochs. The piece selection model achieved an accuracy of 35.87%. The rest of the "to which field" models, depending on the type of the piece they dealt with, obtained the results shown in Table 1.

Tab. 1. Accuracy of the trained models

Model	Accuracy
King	60.19%
Queen	40.6%
Rook	42.48%
Bishop	50.51%
Knight	53.72%
Pawn	49.46%

The models had no prior knowledge of the game of chess. It may happen that the move chosen by the model is illegal. Especially when the situation on the chessboard is related to capturing the king. In order to avoid a situation where the model chooses an illegal move, a list of legal moves is generated beforehand. Each move from the list is assigned a value, which is the product of the result from the piece selection model for the field where the chess piece making the move is, and the result from the model of the piece standing there for the target field. The move with the highest value will be made.

The models produced give higher values to legal moves in most cases. The exception is when the side making the move must escape mate, the models have not learned this aspect of the game. Figure 1 shows an example position along with a number that denotes the values assigned to the fields by the piece selection model. The ranges used: 0.01 – 0.1 (1), 0.21 (2), 0.47 (3). The remaining fields obtained values below 0.01.

The models are characterized by uneven play. On the one hand they make relatively good moves, on the other hand they often expose their own pieces to capture, fall into the opponent's traps and do not capture the opponent's undefended pieces. At a high level in chess a small mistake often results in a lost game, and the produced models make them often. It seems that with this approach it is impossible to teach a model to play chess at a high level. To do so, it is essential to analyze the positions in depth and to predict the opponent's response.

In 1997, the Deep Blue computer was able to beat the then chess master. The developers of Deep Blue do not share all the information about the implemented algorithm. However, it is known that the algorithm was able to estimate about 200 million positions per second and, using a mini-max algorithm, select the best move. In-depth position analysis and prediction of up to a dozen moves ahead resulted in defeating a chess grandmaster. The approach discussed above lacks prediction of the opponent's response, and from the information about the move made for a given position alone, the neural network is not able to infer all the intentions that went behind the move.

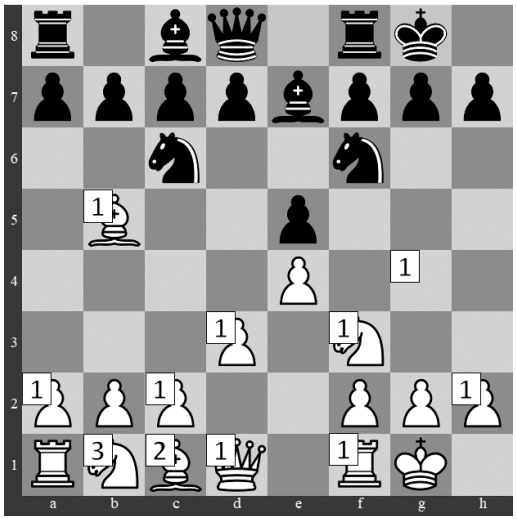

Fig. 1. Example item with field value designations

3. Alpha Zero engine

Currently, one of the best chess engines in the world is the Alpha Zero program, which is based on reinforcement learning and the use of the Monte Carlo Tree Search algorithm. The program is used in games such as shogi and Go, where it has been able to beat the best players in the world.

The basis of Alpha Zero is a neural network, which takes as input a representation of a position on a chessboard **s**. The output of the network is the value of a given position $\mathbf{v(s)} \in \mathbf{[-1, 1]}$. The value is determined from the perspective of the player making the move. Values close to 1 indicate an advantage for that player, while values close to –1 indicate an advantage for the opponent. The second output of the network is the policy $\vec{\boldsymbol{p}}\mathbf{(s)}$ which is a vector of probabilities of all possible moves.
Monte-Carlo Tree Search (MCTS) is a decision-making heuristic mainly used for choosing moves in games. In a search tree, each node represents a setting on a chessboard. A board edge between nodes is a move, after which one can move from one setting to the next.
The nodes hold given values:

- **Q(s, a)** – the expected reward for making move **a** at setting **s**,
- **N(s, a)** – the number of executions of move **a** at setting **s** during the simulation,
- **P(s, ·) = $\vec{p}$(s)** – initial values of move execution at setting **s** based on the policy returned by the neural network,
- $\mathbf{c_{puct}}$ – parameter controlling the degree of tree exploration (high value increases the frequency of selecting new nodes).

Based on these values, the upper confidence limit **U(s, a)** can be calculated from the formula:

$$U(s,a) = c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \qquad (2)$$

MCTS is used to improve the policy returned by the neural network. It is assumed that for a given position, the neural network is unable to find the best move without reviewing many positions in depth. A single simulation consists of three steps (see also Fig. 2):

- Selection – finding the move **a** with the highest value of the sum of the upper confidence limit **U(s, a)** and the reward value **Q(s, a)**. If position **s'** (obtained by playing a move **a** at position **s**) is in the tree, then the step for position **s'** is repeated until a new position is found or a position from which a move cannot be made;
- Branch-off – if position **s'** is not in the tree, then the position is added to the tree and the values **P(s', ·) = $\vec{p}$(s)** and **v(s')** from the neural network and the values **Q(s', a)** and **N(s', a)** as 0 for all moves **a** are initialized. In case the next state cannot be reached from the selected node, then a move directly to the third step is made;
- Backward propagation – in the direction of the tree root, the values of **Q(s, a)** are updated based on the formula:

$$Q'(s,a) = \frac{Q(s,a)\cdot N(s,a) + v(s')}{N(s,a)+1} \qquad (3)$$

If a position from which a move cannot be made occurs during the search, instead of the value **v(s')** calculated by the neural network, there is a value of +1 if the game is won by one of the parties, and a value of 0 if it ends in a draw. Position values are always treated from the perspective of the ancestor. Figure 3 shows an example Monte Carlo tree, where the color of a node indicates the side making the move. Assuming that node E is a position that ended in a mate (the move belongs to white, i.e., white lost), from the perspective of the ancestor (node B), it is a node that is worth visiting frequently, but for node A, node B leads to a loss, so during backward propagation, the value of **v(s')** after updating **Q(s, a)** changes to the opposite value each time.

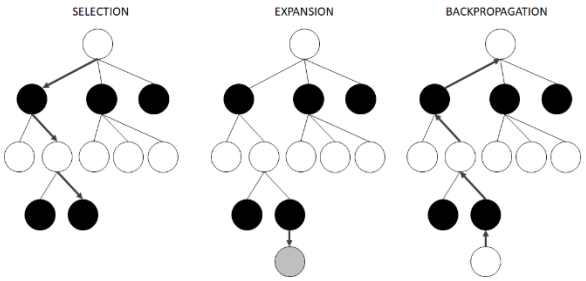

Fig. 2. Steps in the Monte Carlo Tree Search algorithm

The condition to complete the simulation can be a certain time or a certain number of iterations. The move considered best will be the move with the highest value indicated by the improved stochastic policy $\vec{\pi}(s)$. It is calculated from the formula:

$$\vec{\pi}(s) = \frac{N(s,\cdot)}{\sqrt{\sum_b N(s,b)}} \qquad (4)$$

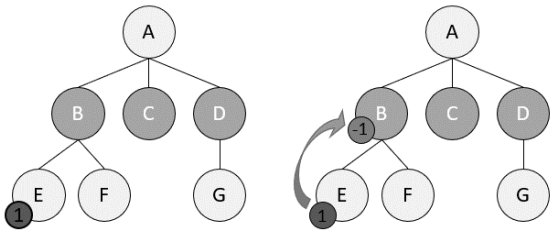

Fig. 3. Backward value propagation in the MCTS algorithm

In order to test the correctness of the algorithm, three positions were used in which:

- whites give a mate in one move, where there are two such moves,
- blacks mate in one move,
- blacks mate in two moves.

The values of the positions that the neural network should calculate are randomly drawn from the range –0.8 to 0.8. 25000 iterations were performed in the test. Figure 4 illustrates the result of the test for a position in which whites can give a mate in two ways: Bf7 and Qf7. Below the figures the values of **N(s, a)** for the given position are shown.

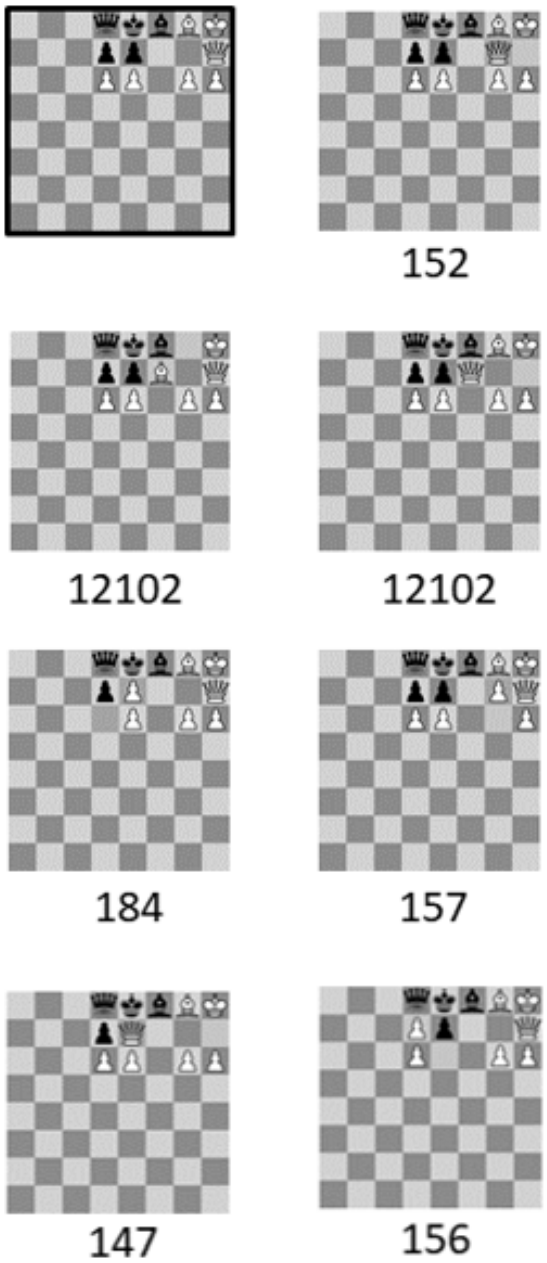

Fig. 4. Test result for the first position

Positions that lead to a mate are visited much more often than the others. By increasing the number of iterations the values of **N(s, a)** remain in similar proportions. In position two, the move leading to the mate accounts for 80% of all choices for the base node. In the third position, on the other hand, with a small number of iterations, the algorithm will not select with significant frequency the movement leading to the mate. By increasing the number of iterations, once the sequence leading to the mate is found, this move starts to be the most frequently chosen move in subsequent iterations.

In this approach, the neural network has the task of estimating whether a given position leads the side making the move to victory, defeat or a draw. The second task of the neural network is to return the policy $\vec{p}(\mathbf{s})$. The data for learning the network will be extracted from the games played. The algorithm will play against itself, recording the position and the refined stochastic policy $\vec{\pi}(\boldsymbol{s})$ at each move. After a game is played, the data set will be expanded to include the reward value **z**. All reward values for a set containing a position where the move is made by the side winning the game will be 1. In case of a loss the value will be –1. If the game ends in a draw, then regardless of the side making the move, the reward value of all positions will be 0. It is worth noting that in a duel both networks play the same number of games as white and black. The next step is to train the model on the basis of the data obtained from the played games and check whether the training has had the desired effect. The network that wins at least 55% of the games played becomes the best network. This method produces an infinite amount of data that can be used to teach a neural network.

In the learning process, the cost function is the sum of two components:

- policy cost function (categorical cross entropy),
- cost function of the item value (mean squared error).

Each move in the learning process is obtained based on 800 iterations. In order to speed up the games played, the subtree containing the selected move is used when searching for the next move. This occurs only when playing one model against itself. When comparing models, such solutions would lead to unreliable results.

As mentioned before, one set of data necessary in the learning process has the following form $(\mathbf{s}, \vec{\pi}(\boldsymbol{s}), \mathbf{z})$. The representation of the setting on the chessboard and the policy require prior normalization:

- **s –** the representation of the setting on the chessboard will consist of a 16 x 8 x 8 array.
  It can be divided into two components:
  - **chess pieces (12 x 8 x 8) –** each 8 x 8 array refers to the type of piece and the field on which it is placed. The 1 is located at positions that correspond to the piece type and the field on which it stands. The position is always shown from the perspective of the side making the move, i.e. when black makes a move, the board is reversed on the basis of the reflection with respect to the line between the fourth and the fifth row of the board;

- **castling rights (4 x 8 x 8)** – an 8 x 8 array, where the fields of the array are filled with ones when a given side has castling rights, otherwise the array is filled with zeros. The arrays refer, respectively, to: short castling of the side making the move, long castling of the side making the move, short castling of the opposing side, long castling of the opposing side;

- $\vec{\pi}(s)$ – the policy is in the form of a list with length corresponding to the number of moves possible at position **s**. The neural network requires a fixed size for the output data, so its size will be 4096 (the number of possible moves "from the field" times the number of possible moves "to the field"). The various options for promoting the pawn have been omitted. In the MCTS algorithm, the policy value for the pawn move on the last row will be equal for all pawn promotion options. An array of 4096 zeros will be generated during normalization. The cells with the index "from which field" times 64 + "to which field" will take the value from $\vec{\pi}(s)$ corresponding to that move. If blacks makes a move, then the field numbers refer to other moves. Figure 5 on the left shows the position with the possible move of the knight from B8 to C6. This move refers to index numbers 1 and 18. On the right the reflected position is shown. To make sure that black's moves correspond to the actual situation on the chessboard, an array with the following values has been used: [56, 40, 24, 8, –8, –24, –40, –56]. Each value in the array refers to a row on the chessboard, starting from the eighth row. The corresponding value from the array is added to the field index number.

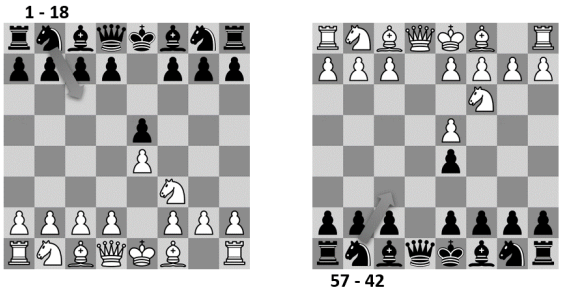

Fig. 5. Perspective of representation of positions relative to blacks

In the initial phase of the learning process, the policy returned by the neural network will often give non-zero values to illegal movements. The *softmax* function returns items whose values sum to one. The policy values assigned to all legal moves in an item must sum to one. For this purpose, before assigning policy values to individual nodes, these values are first summed. Then, each node's policy value is calculated as the quotient of the value returned by the neural network and the sum of the policy values of all legal moves.

The neural network model was built using the *keras* library. The following layers were used in the implementation:

- Conv2D – convolutional layer, which is used in processing two-dimensional images. The position on the chessboard is treated as an image with dimensions of 8 x 8 and 16 channels (12 – types of chess pieces, 4 – castling rights);
- BatchNormalization – applies a transformation that keeps the mean activation close to 0 and the standard deviation of the activation close to 1;
- Activation (ReLU) – layer that processes received data in accordance with the established activation function;
- Add – layer that receives on input 2 data sets of the same size and on output returns a data set as a sum of individual elements;
- Flatten – layer which processes data in the form of tensor of n dimensions into tensor of one dimension;
- Dense – standard layer used in neural networks, where each neuron in the layer receives on input data from each output of the previous layer.

Fig. 6 shows a layout of the neural network model used in this approach. The data that is the position representation becomes the input data of the neural network. The layer block indicated by the blue rectangle is repeated 7 times in the model. The model at the end splits into two outputs: the value output (on the left) and the policy output (on the right). The value output at the end has a *tanh* activation function that assumes values $f(x) \in [-1, 1]$. The output of the policy ends with a *softmax* function.

During the training of the neural network, the matches played must be different to some extent. Without additional modifications, the MCTS algorithm with the neural network would always play the same game – or two games, in case of swapping sides. During a duel between two models, it is important to provide training data from different games without significantly interfering with the level of play of a given model.

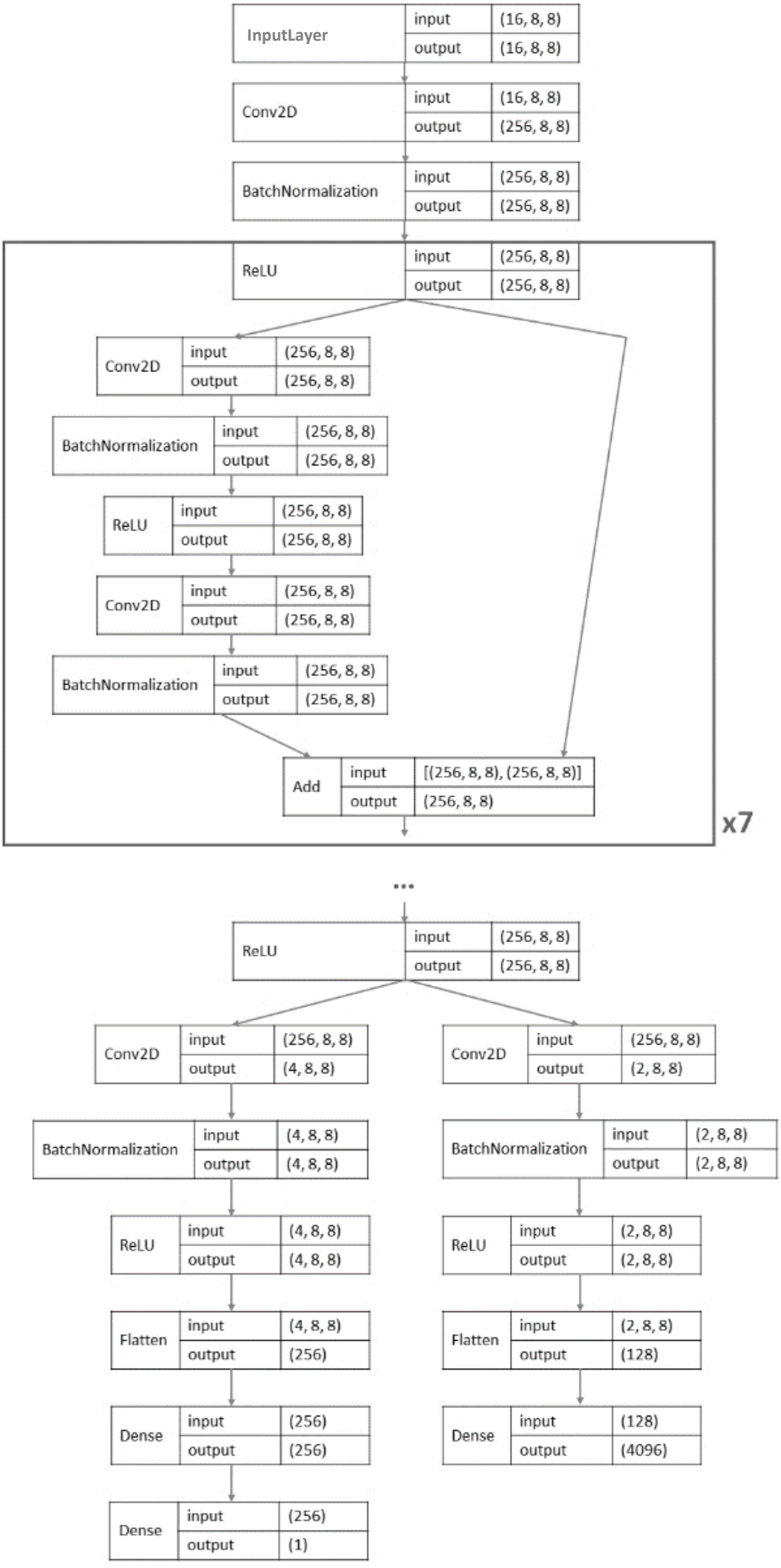

Fig. 6. Schematic of the neural network model

For this purpose, the direct descendants of the base node have the value **P(s, a)** modified as follows:

$$P'(s, a) = (1 - \varepsilon) \cdot P(s, a) + \varepsilon \cdot \eta_a \qquad (5)$$

The vector η has a length equal to the number of possible moves at a given position. The elements of this array sum to a value of 1, and their distribution depends on the parameter α. A parameter α close to infinity leads to a uniform distribution, while the parameter α close to zero leads to an array of only zeros and a single one. The value of the parameter α is 0.3, and the value of ε used above is 0.25. Additionally, *Dirichlet noise* with enough simulations ensures that all direct descendants of the base node are visited at least once.

Prior to the main training, the model was taught using 120,000 positions derived from games played by chess engines. The policy was encoded with zeros and ones corresponding to the move selected. This was followed by a duel between the taught model (M1) and a new model (M0), where M1 won 17 games and tied 3. The next step was to collect self-play data for about 24 hours. During this time, 17,000 data sets were collected and used to learn the model. As a result of training the M1 model on the data obtained, the M2 model was obtained. The M2 model won 12 games, lost 2, and tied 6. In the algorithm, most of the time (over 90%) is occupied by the operations related to the use of the model. For better results, one can use the computing power from the graphics card or collect data using multiple computers simultaneously.

Alpha Zero developers used the 5000 TPU (Tensor Processing Unit) of the first generation for self-game and 64 TPU of the second generation to train the model. Such hardware allowed 44 million games to be played at 800 simulations per move in 9 hours. After this time, Alpha Zero faced one of the best chess engines – Stockfish. Alpha Zero won 25 games and tied 25 when playing white, while when playing black it won 3 and tied the remaining 47.

4. Conclusions

The purpose of this study was to analyze and implement machine learning methods for their usefulness in teaching a computer to play chess. The first chapter included a description of the rules of chess and how to record the flow of the game and record positions, which were useful in subsequent chapters. Two methods for teaching a computer to play chess were discussed in the work: the first based on played games, the results of which show that finding a good move based solely on the position on the chessboard is not possible without a deeper analysis of the position. The best chess engines, both the classical ones and those based on neural networks, come down to searching positions in depth, assuming that the opponent will always choose the best move for himself. The second method resembles the process of choosing a move by a human, who intuitively selects for analysis sequences of moves that he thinks are worth attention. A significant advantage of Alpha Zero is its good scalability with computational power compared to classical engines, which have exponential computational complexity [5]. Another advantage is the possibility of continuous improvement of the game level. Due to the lack of limitations, each successive model can be better than the previous one. Disadvantages of Alpha Zero include the need for good hardware, which is necessary both to generate training data and to measure game level. The long game play time made it difficult to test the algorithm. Finding an error often required the program to run for several hours. The implementation also included a graphical user interface that enabled the game to be played and facilitated the testing of the algorithm through the ability to enter and modify positions.

5. Bibliography

[1] Herud K., Mueller C., "End-to-End Deep Neural Network for Automatic Learning in Chess", *International Journal of Computer Theory and Engineering*, Vol. 10, No. 5, 146–151 (2018).

[2] Kosiński R., *Sztuczne sieci neuronowe: Dynamika nieliniowa i chaos*, Wydawnictwo Naukowe PWN, Warszawa 2020.

[3] Oshri B., Khandwala N., *Predicting Moves in Chess using Convolutional Neural Networks*, http://cs231n.stanford.edu/reports/2015/pdfs/ConvChess.pdf.

[4] Silver D., et al, "Mastering the game of Go without human knowledge", *Nature*, 550, 354–359 (2017).

[5] Silver D. et al., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, https://arxiv.org/pdf/1712.01815.pdf.

[6] Wantoch-Rekowski R., *Sieci neuronowe w zadaniach: perceptron wielowarstwowy*, Bel Studio, Warszawa 2003.
[7] Wójcik P., *Metody uczenia maszynowego w implementacji gry w szachy*, Warszawa 2020.
[8] Zocca V., et al., *Deep Learning. Uczenie głębokie z językiem Python. Sztuczna inteligencja i sieci neuronowe*, Helion, Gliwice 2018.
[9] Forsyth–Edwards Notation [@] https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation

Metody uczenia maszynowego w implementacji gry w szachy

P. WÓJCIK, J. WIŚNIEWSKA

W pracy zaprezentowano metody wykorzystania uczenia maszynowego do nauki komputera gry w szachy. Pierwsza metoda bazuje na użyciu zapisów przebiegów partii rozgrywanych przez wysoko klasyfikowanych graczy, zaś druga opiera się na algorytmie Monte Carlo Tree Search oraz uczeniu przez wzmacnianie.

Słowa kluczowe: szachy, uczenie maszynowe, sieci neuronowe.

This work was financed by Military University of Technology under research project UGB no 860 /2021.