Neural Learning of Heuristic Functions for General Game Playing

Neural Learning of Heuristic Functions for General Game Playing

Leo Ghignone, RossellaCancelliere

University of Turin, Department of Computer Science

Abstract. The proposed model represents an original approach to general game playing, and aims at creating a player able to develop a strategy using as few requirements as possible, in order to achieve the maximum generality. The main idea is to modify the known minimax search algorithm removing its task-specific component, namely the heuristic function: this is replaced by a neural network trained to evaluate the game states using results from previous simulated matches. A method for simulating matches and extracting training examples from them is also proposed, completing the automatic procedure for the setup and improvement of the model. Part of the algorithm for extracting training examples is the Backward Iterative Deepening Search, a new original search algorithm which aims at finding, in a limited time, a high number of leaves along with their common ancestors.

Keywords: Game Playing, Neural Networks, Reinforcement Learning, Online Learning

1 Introduction

A general game player is an agent able to develop a strategy for different kinds of games, based on only the rules of the game to be learned; differently from a specialized player (a notable example of which can be found in [1]), properties and particularities of the played games cannot be hardcoded into the player. Most of the literature on general game playing focuses on the usage of the Game Description Language (GDL) which is part of the General Game Playing (GGP) project of Stanford University, and has become the standard for general game playing during the last years. It is a logical language which describes the rules of a game with a set of logical rules and facts, using specific constructs in order to define properties such as initial states, player rewards, turn order, etc. This approach however causes a double loss in generality: a general game player is unable to learn games not describable by GDL (like games with hidden information or randomic elements), and a direct access to all the rules of the game is required. As a consequence, most of the usual general game players focus on logically inferring features and strategies from the rules of the game (see, e.g., [2–5]) even when they do admit some sort of learning.

The goal of this work is to propose a system, named Neural Heuristic Based player (NHB-player) that, in the framework of the classical two-players board games, gradually learns a strategy; for this purpose a neural network is used whose training set is collected through multiple simulated matches. Its main advantage is the possibility of learning from experience, which can give better results than rule analysis when the rules are complex or tricky, and allows the development of a strategy even if the rules are not given according to a logic-based description. The proposed general game player is immediately able to play any game showing the following properties:

- zero-sum, i.e. gain for a player always means loss for the other(s)
- turn-based
- deterministic, that is no random elements are permitted
- all players have perfect information, (no information is hidden to any player)
- termination is guaranteed.

State-space searches are a powerful tool in artificial intelligence, used in order to define a sequence of actions that can bring the system to a desired state. Searching through all possible evolutions and actions is almost always impractical, so *heuristic functions* are used to direct the search. A heuristic function is a function that, given a state, returns a value that is proportional to the distance from that state to a desired goal. These functions usually have to be implemented by human experts for each specific task, thus turning a general-purpose algorithm such as a state space search into a specific one. For this reason, state-of-the-art general game players [6] usually avoid heuristic functions, relying instead on Monte-Carlo searches to find the values of different states.

The first contribution of our work is to explore the possibility that a feedforward neural network can be automatically trained to compute heuristic functions for different game states rather than using Monte-Carlo methods for state evaluation.

The moves of our NHB-player in the space of all possible game states are chosen using a minimax algorithm with alpha-beta pruning and iterative deepening: this method is described in Section 2. The neural model is a single hidden layer feedforward neural network (SLFN), described in Section 3. Over a few decades, methods based on gradient descent have mainly been used for its training; among them there is the large family of techniques based on backpropagation, widely studied in its variations. The start-up of these techniques assigns random initial values to the weights connecting input, hidden and output nodes, that are then iteratively adjusted.

Some non iterative procedures have been proposed in literature as learning algorithms for SLFNs based on random assignation of input weights and output weights evaluation through pseudoinversion; some pioneering works on the subject are [7] and [8]. Recently Huang et al. [9] proposed the Extreme Learning Machine (ELM) which has the property of reducing training to an analytic one-step procedure while maintaining the universal approximation capabilities of the neural network; we use for training our model an online formulation called OS-ELM proposed by Liang et al. [10] and described in Section 3.1, which allows to divide the training examples into multiple sets incrementally presented to the network. This choice comes from the fact that the search algorithm and the neural heuristic assignment in our case are complementary to each other during all the phases of player's training, so that the training instances, each composed by a state and the heuristic function value for that state, are subsequently collected during different training matches and are not entirely available from the beginning.

A crucial step for neural training is the selection of a sufficient number of training examples. Since we aim at creating a system which autonomously learns a heuristic function, we don't want to turn to records of previously played games or to external opponents to train against. We therefore propose an original method for extracting the training examples directly from matches simulated by the system itself through a procedure described in Section 4. Part of this procedure is the Backward Iterative Deepening Search (Backward IDS), a new search algorithm presented in Section 4.1, which aims at finding, in a limited time, a high number of leaves along with their common ancestors.

Performance is evaluated on two classical board games of different complexity: *C*onnect Four and *R*eversi. Implementation details, experimentation description and results are reported in Section 5.

2 Search Algorithm

The state-space search algorithm classically used for this kind of games is the minimax algorithm: it was originally developed for two-players zero-sum games, but it can also be adapted to other kinds of games. The interested reader can find a description of the minimax search and of other algorithms cited in this Section (alpha-beta search, iterative deepening) in [11].

The idea behind this algorithm is that a player wants to minimize its opponent's maximum possible reward at the end of the game. In order to know what is the best outcome reachable for the opponent at each state of the game, the algorithm builds a tree of all the possible evolutions of the game starting from the current state and then backtracks from the terminal states to the root. Since it is usually impractical to expand the whole tree to the end of the game, the search is stopped at a certain level of the tree and the heuristic function is applied to the nodes of that level in order to estimate how favorable they are. For games without a final score, reward values are set to -1 for a loss, 0 for a tie and 1 for a win.

In our model we utilise an improvement over the minimax algorithm, i.e. the so-called *alpha-beta pruning*: if at some points during the exploration of the tree a move is proven to be worse than another, there is no need to continue expanding the states which would be generated from it.

When using this pruning, the complexity of the search depends on the order in which nodes are explored: in our implementation nodes are ordered by their value of heuristic function, given by the neural network, thus resulting in searches that are faster the better the network is trained.

One of the main factors influencing performance is the depth of the search. The playing strength can be greatly improved increasing the depth at which the search is stopped and the heuristic function is applied, but this objective is reached at the cost of an exponentially increasing reasoning time.
With the aim of finding a good tradeoff between these two requirements we applied an iterative deepening approach: this choice has also the advantage of automatically adapting the depth during a game, increasing it when states with few successors are encountered (for example when there are forced moves for some player), and decreasing it when there are many moves available. Setting a time limit instead of a depth limit is easier since the duration of the training can be directly determined, and a finer tuning is allowed.

3 Neural Model and Pseudo-inversion Based Training

In this section we introduce notation and we recall basic ideas concerning the use of pseudo-inversion for neural training. The core of the NHB-player learning is a standard SLFN with P input neurons, M hidden neurons and Q output neurons, non-linear activation functions $\phi(x)$ in the hidden layer and linear activation functions in the output layer. In our model the input layer is composed by as many nodes as are the variables in a state of the considered game, while the hidden layer size is determined through a procedure of k-fold cross validation described in the following section. All hidden neurons use the standard logistic function $\phi(x) = \frac{1}{1+e^{-x}}$ as activation function. The output layer is composed by a single node because only a single real value must be computed for each state.

From a general point of view, given a dataset of N distinct training samples of (input, output) pairs $(\mathbf{x}_j, \mathbf{t}_j)$, where $\mathbf{x}_j \in \mathbb{R}^P$ and $\mathbf{t}_j \in \mathbb{R}^Q$, the learning process for a SLFN aims at producing the matrix of desired outputs $\mathbf{T} \in \mathbb{R}^{N\times Q}$ when the matrix of all input instances $\mathbf{X} \in \mathbb{R}^{N\times P}$ is presented as input.

As stated in the introduction, in the pseudoinverse approach input weights c_{ij} (and hidden neurons biases) are randomly sampled from a uniform distribution in a fixed interval and no longer modified. After having determined input weights $\mathbf{C}$, the use of linear output units allows to determine output weights w_{ij} as the solution of the linear system $\mathbf{H}\,\mathbf{W} = \mathbf{T}$, where $\mathbf{H} \in \mathbb{R}^{N\times M}$ is the hidden layer output matrix of the neural network, i.e. $\mathbf{H} = \Phi(\mathbf{X}\,\mathbf{C})$.

Since $\mathbf{H}$ is a rectangular matrix, the least square solution $\mathbf{W}^*$ that minimises the cost functional $E_D = ||\mathbf{HW} - \mathbf{T}||_2^2$ is $\mathbf{W}^* = \mathbf{H}^+\mathbf{T}$ (see e.g. [12, 13]).

$\mathbf{H}^+$ is the Moore-Penrose generalised inverse (or pseudoinverse) of matrix $\mathbf{H}$.

Advantage of this training method is that the only parameter of the network that needs to be tuned is the dimension of the hidden layer. Section 4.2 will show the procedure utilized to do so.

3.1 Online Sequential Version of the Training Algorithm

The training algorithm we implemented to evaluate $\mathbf{W}^*$ is the variation proposed in [10], called Online Sequential Extreme Learning Machine (OS-ELM):

its main advantage is the ability to incrementally learn from new examples without keeping previously learned examples in memory. In addition to the matrices $\mathbf{C}$ and $\mathbf{W}$, containing the hidden and output layer weights, a network trained with this algorithm also utilises an additional matrix $\mathbf{P}$.

The starting values for the net's matrices $\mathbf{P}_0$ and $\mathbf{W}_0$ are computed in the OS-ELM algorithm with the following formulas:

$$\mathbf{P}_0 = (\mathbf{H}_0^T \mathbf{H}_0)^{-1}, \quad \mathbf{W}_0 = \mathbf{P}_0 \mathbf{H}_0^T \mathbf{T}_0 \tag{1}$$

where $\mathbf{H}_0$ and $\mathbf{T}_0$ are respectively the hidden layer output matrix and the target matrix relative to the initial set of training examples.

The update is done in two steps, according to the following equations:

$$\mathbf{P}_{k+1} = \mathbf{P}_k - \mathbf{P}_k \mathbf{H}_{k+1}^T (\mathbf{I} + \mathbf{H}_{k+1} \mathbf{P}_k \mathbf{H}_{k+1}^T)^{-1} \mathbf{H}_{k+1} \mathbf{P}_k \tag{2}$$

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \mathbf{P}_{k+1} \mathbf{H}_{k+1}^T (\mathbf{T}_{k+1} - \mathbf{H}_{k+1} \mathbf{W}_k) \tag{3}$$

The network weights $\mathbf{W}$ are thus updated basing on the hidden layer output matrix for the new set of training examples $\mathbf{H}_k + 1$, the target matrix of the new set of training examples $\mathbf{T}_k + 1$, and the previous evaluated matrices $\mathbf{P}_k$ and $\mathbf{W}_k$. $\mathbf{I}$ is the identity matrix.

The network can be trained with sets of examples of varying and arbitrary magnitude: it is however better to accumulate examples obtained from some different matches, instead of allowing learning after every game, in order to be able to discard duplicates, so reducing the risk of overfitting on the most frequent states. This technique can anyway cause a considerable decrease of the learning speed, since the matrix to be inverted in eq. (2) has a dimension equal to the square of the number of training examples presented. In this case or if the dimension of this matrix exceeds the available memory space it is sufficient to split the training set in smaller chunks and iteratively train the network on each chunk.

4 Training Examples Selection

As already mentioned, multiple training examples are required, each composed by a state and the desired heuristic function value for that state. Since we aim at creating a NHB-player which autonomously learns the heuristic function, we do not want to rely on records of previously played games or on external opponents to train against. Therefore, examples are extracted directly from simulated matches.

The optimal candidates for building training examples are the terminal states of the game, whose value of heuristic function can be determined directly by the match outcome: we assign to these states a value 1 for a win, −1 for a loss and 0 for a tie. Using the search tree, the values of terminal states can be propagated unto previous states that lead to them; in this way we are able to build examples using non-terminal states, which are the ones for which a heuristic function is actually useful.

In addition, for each state of the played match a new example is generated: the heuristic value of these examples is computed as increasing quadratically during the match from 0 to the value of the final state. This way of assigning scores is arbitrary and it is based on the assumption that the initial condition is neutral for the two players and that the winning player's situation improves regularly during the course of the game. This assumption is useful because it allows to collect a certain number of states with heuristic function values close to 0, which can prevent the net from becoming biased towards answers with high absolute value (for games in which drawing is a rare outcome). Even if some games are not neutral in their initial state, they are nevertheless complex enough to be practically neutral for any human (and as such imperfect) player; moreover, the value of the initial state is never utilized in the search algorithm, so its potentially incorrect estimation doesn't affect the performance of the system. Finally, duplicate states within a training set are removed, keeping only one example for each state; if in this situation a value assigned for a played state and a value coming from propagation of a terminal state are conflicting, the second one is kept since it's the most accurate one.

After enough training matches, the system can propagate heuristic values from intermediate states, provided that these values are the result of previous learning and are not completely random. This can be estimated using the concept of *consistency* of a node, discussed in [14]: a node is considered consistent if its value of heuristic function is equal[1] to the value computed for that node by a minimax search with depth 1. All consistent nodes can be used for propagation in the same way as terminal nodes, maintaining the rule that, when conflicts arise, values propagated from terminal nodes have the priority.

4.1 Backward Iterative Deepening Search

When in the complete tree[2] of a game the majority of the leaves are positioned flatly in one or a small number of levels far from the root, they are not reachable with a low-depth search from most of the higher nodes. In games like *R*eversi, for example, no terminal states are encountered during the search until the very last moves, so the number of training examples extracted from every match is quite small.

In order to find more terminal states in a game of this kind, a deeper search is needed starting from a deep enough node. Good candidates for this search are the states occurred near the end of a match, but it is difficult to determine how near they must be in order to find as many examples as possible in a reasonable time. The solution is once again to set a time limit instead of a depth limit and utilize an iterative deepening approach: in this case we talk about a *Backward Iterative*

[1] In the case of the proposed system, since the output is real-valued, the equality constraint must be softened and all values within a margin ε are accepted.

[2] The *complete* tree of a game is the tree having as root the initial state and as children of a node all the states reachable with a single legal move from that node. The leaves of this tree are the terminal states of the game.

Deepening Search (or *Backward IDS*). The increase in depth at each step is accompanied by a raising of the root of the search, resulting in sub-searches all expanding to the same level but starting from nodes increasingly closer to the root (see Figure 1). Differently from the standard iterative deepening search, in the Backward IDS each iteration searches a sub-tree of the following iteration, so the value of the previous sub-search can be passed to the following in order to avoid recomputing; this iterative search has then the same complexity as the last fixed-depth sub-search done, while automatically adapting itself to different tree configurations.

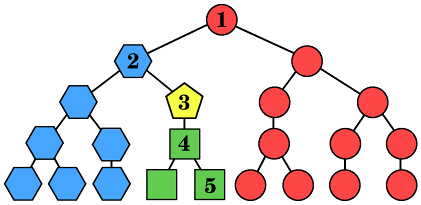

Fig. 1. An example of application of the Backward IDS. The played game is the sequence of states 1-2-3-4-5. The search starts from state 4 with depth one, then goes back to state 3 with depth two and so on until timeout. Nodes with the same shape are explored during the same iteration.

4.2 Training Procedure

Each training procedure starts with a certain number of games played between agents which both use random value heuristics. The states explored during these games by both players are collected, multiple copies of the same state are eliminated, and the resulting set is used as initial training set for the neural network.

Since the more trained a network is the better training examples will it generate, further training has to be performed by alternating phases of game simulation (where training examples are extracted and accumulated through simulating games played by the trained system) and phases of neural network training (where the system improves basing on the training examples extracted during the previous phase).

So doing the extraction of post-initial training examples is performed in the framework of games played against a similar opponent which applies the same search strategy but whose heuristic function comes from a randomly initialized neural network. Most games will (hopefully) be won by the trained player, but the system will still improve itself through the exploration of new states.

It can also be useful to have the system play against itself, but for no more than one match in each training cycle. The deterministic structure of the search algorithm implies in fact that, unless the heuristic function is modified, multiple matches against the same opponent will be played exactly in the same way.

Aiming at finding the optimal number of hidden neurons that allows to avoid overfitting, we determine hidden layer size and hidden layer weights through a 10-fold cross validation: hidden layers of increasing sizes are explored until for 10 iterations in a row the best average validation error doesn't decrease. During each iteration three different random initialization for the hidden weights are tried, and only the best one is kept.

5 Implementation Details, Game Descriptions and Results

The system structure described in the previous sections is completely independent from the game to be played. A modular structure is adopted to maintain this independence: four game-dependent functions, separated from the rest of the system, completely describe each game and contain all its rules so that different implementations of these functions permit to play different games.
init_state: This function returns the initial state of the game, taking as argument the starting player (1 or -1 for the two-player games implemented).
expand: This function returns a list of all the moves available from a given state received as input, along with the list of states they lead to.
test_terminal: This function takes as input a state and checks if it is a terminal one, returning in this case also the final outcome.
apply: This function gets as input a state and a move, then applies the chosen move to the state and returns the new state reached.

We implemented two different games to test the learning capabilities of our artificial player: *Connect Four* and *Reversi*. These games have different degrees of complexity and, while they both possess the properties described in the Introduction, they have some peculiarities that make them apt to test different aspects of the system. We will now briefly present their characteristics and discuss the obtained results.

5.1 Connect Four

This game is played on a grid with 6 rows and 7 columns, where players alternatively place their tokens. A token can only be placed on the lowest unoccupied slot of each column[3], and the first player which puts four of its tokens next to each other horizontally, vertically, or diagonally is the winner.

Allis ([15]) demonstrated that the first player always wins, under the condition of perfect playing on both sides; this level of playing, almost impossible for a human, is difficult even for a machine, since artificial perfect players usually require a database of positions built with exhaustive search in order to successfully choose the best move.

[3] The most common commercial versions of this game are played on a suspended grid, where tokens fall until the lowest free position.

The branching factor[4] for this game is constant and equal to 7, until a column is filled. This makes the search depth almost constant, thus reducing the advantage of an iterative deepening search.

We evaluate the performance of our model looking at the percentage of matches won against an untrained opponent. We also explored the influence of the time limit for each move, varying the move time available during training: three different networks were trained, with move times of 0.25 seconds (s), 0.5 s and 1 s respectively. Aiming at comparing networks trained for the same **total** amount of time, the number of training matches was proportionally halved.

Table 1. Varying move time in Connect Four training. T=Training matches. L,D,W= Losses, Draws, Wins over 200 testing matches

move time=0.25s				move time=0.5s				move time=1s			
96 hidden neurons				112 hidden neurons				113 hidden neurons			
T	L	D	W	T	L	D	W	T	L	D	W
20	79	11	110	10	54	14	132	5	59	18	123
40	53	9	138	20	68	12	120	10	64	13	123
60	46	13	141	30	88	13	99	15	57	13	130
80	68	18	114	40	44	12	144	20	69	16	115
100	46	12	142	**50**	**43**	**9**	**148**	25	61	19	120
120	**42**	**10**	**148**	60	69	13	118	**30**	**47**	**18**	**135**
140	66	11	123	70	50	8	142	35	84	18	98
160	59	9	132	80	54	13	133	40	77	12	111
180	60	14	126	90	58	7	135	45	62	14	124
200	47	13	140	100	55	11	134	50	61	10	129

As shown in Table 1, the best performance is obtained with a move time equal to 0.25 s. and 120 training matches. The number of won matches clearly demonstrates the effectiveness of our proposed approach in developing a successfull playing stategy.

Looking at the results obtained with move time = 0.5 s. we see that a comparable performance is reached with about half the number of training matches, i.e. 50; it is interesting to note that in this way the total training time remains almost unchanged. Besides, also the network trained with move time equal to 1 s. reaches the best performance in approximately half the number of training games with respect to the case of move time = 0.5 s., but now it appears to be slightly worse: this indicates the possible existence of an optimal move time that can produce the best trained network in a fixed total training time. Developing a way to find this optimal time will be an objective for future work.

[4] The branching factor is the average number of branches (successors) from a (typical) node in a tree. It indicates the bushiness and hence the complexity of a tree. If a tree branching factor is B, then at depth d there will be approximately B^d nodes.

5.2 Reversi

*R*eversi, also known as Othello because of its most popular commercial variant, is a more complex game played on a 8 by 8 board where two players alternate placing disks whose two faces have different colours, one for each player. When the board is full or no player has a legal move to play, disks are counted and whoever owns the highest number of them wins the match. A more detailed description of this game can be found in [16].

This game is characterised by high average branching factor and game length (for almost all matches 60 moves, except for the rare cases where no player has a legal move); finding available moves and checking for terminal states is very expensive from a computational point of view, resulting in a high time required to complete a search.

While since 1980 programs exist that can defeat even the best human players (see [17] for a review of the history of Othello computer players), the game has not yet been completely solved: performance of the top machine players indicates that perfect play by both sides probably leads to a draw, but this conjecture has not yet been proved. The branching factor varies during the game, usually reaching the highest value (about 15) during the middle phase.

99	-8	8	6	6	8	-8	99
-8	-24	-4	-3	-3	-4	-24	-8
8	-4	7	4	4	7	-4	8
6	-3	4	0	0	4	-3	6
6	-3	4	0	0	4	-3	6
8	-4	7	4	4	7	-4	8
-8	-24	-4	-3	-3	-4	-24	-8
99	-8	8	6	6	8	-8	99

Fig. 2. The positional evaluation table used for state evaluation in old Microsoft Windows Reversi programs.

For the game of *R*eversi many heuristic functions for artificial players have been defined in the past (e.g. in [18]). In order to compare our proposed model with some baseline performance, we chose two human-defined heuristics, the piece-advantage and the positional ones: the first one is extremely simple while the second one is actually used in artificial playing.

Piece-advantage: this heuristic simply assigns state values equal to the difference between the number of owned disks and the number of opponent's disks.

Positional: this heuristic assigns to each disk a value which depends on its position; the values of all disks owned by each player are then summed and the difference between the two players' scores is returned. For our implementation we chose the same evaluation table used by some Microsoft Windows *R*eversi programs (as noted in [19]), shown in Figure 2.

Table 2 compares performance obtained by the NHB-player and by two players which choose their moves basing on the heuristics described above.

Table 2. Performance comparison among NHB, Piece-advantage and Positional heuristics over 100 matches against opponents who play using random heuristics.

Heuristic	Training Matches	Losses	Draws	Wins
NHB (131 hidden neurons)	50	36	5	59
	100	**24**	**4**	**72**
	150	39	2	59
	200	36	0	64
	250	30	6	64
	300	41	4	55
	350	38	5	57
	400	42	2	56
	550	39	2	59
	600	36	3	61
Piece-advantage	-	53	3	44
Positional	-	26	2	72

We can see that the Piece-advantage heuristic performes even worse than the random one, losing more than half of the matches. This result emphasizes that the development of a heuristic function is not a simple task: the Piece-advantage heuristic, that may seem good in theory because it tries to maximise the score at each stage, turns out to be harmful in practice.

Because of the higher number of matches lost, the Positional heuristic performes slightly worse with respect to the NHB-player one: we stress the fact that this happens although it is a game-dependent human-defined heuristic.

6 Conclusions

Results discussed in the previous section show that our main objective has been achieved: for each implemented game the NHB-player is able to develop a successful playing strategy, without the necessity of tailor-made human contributions but only thanks to the knowledge of game rules.

The strength of the system lies in its adaptability, which allows to deal with both other games with the discussed properties and games characterized by different features.

Future work will focus on one hand in enabling the exploitation of additional information that may be given to the player (for example a score given turn-by-turn, as is the case in [20]), on the other hand in expanding our model to make it able to learn other interesting set of games, some of which are:

chance-based games, that can be managed by simply modifying the minimax search algorithm

games with repeated states, that are hard to control since they can transform into never-ending ones. In this case a modification of the search algorithm in the training phase is necessary to avoid infinite loops.

References

1. Silver, D., Huang, A., et al.: Mastering the game of Go with deep neural networks and tree search. In: Nature 529, no. 7587, pp. 484–489. (2016)
2. Draper, S., Rose, A.: Sancho ggp player, `http://sanchoggp.blogspot.com`
3. Michulke, D.: Neural networks for high-resolution state evaluation in general game playing. In: IJCAI-11 Workshop on General Game Playing (GIGA11), pp. 31–37. (2011)
4. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: 22nd National Conference on Artificial intelligence, pp. 1191–1196. AAAI Press, Menlo Park(2007)
5. Swiechowski, M., Mandziuk, J.: Specialized vs. multi-game approaches to AI in games. In: Intelligent Systems 2014, pp. 243–254. (2015)
6. Swiechowski, M., Park, H., Mandziuk, J., Kim, K.: Recent Advances in General-Game Playing. In: The Scientific World Journal (2015)
7. Schmidt, W.F., Kraaijveld, M., Duin, R.P.W., et al.: Feedforward Neural Networks with Random Weights. In: International Conference on Pattern Recognition, Conference B: Pattern Recognition Methodology and Systems, pp. 1–4. (1992)
8. Pao, Y.H., Park, G.H., Sobajic D.J.: Learning and Generalization Characteristics of the Random Vector Functional-link Net. Neurocomputing, 6, 163–180 (1994)
9. Huang, G.B., Chen, L., Siew, C.K.: Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE Transactions on Neural Networks, 17, 879–892 (2006)
10. Liang, N.Y., Huang, G.B., Saratchandran, P., Sundararajan, N.: A fast and accurate online sequential learning algorithm for feedforward networks. IEEE Transactions on Neural Networks, 17, 1411–1423 (2006)
11. Russell, S., Norvig, P.: Artificial Intelligence: a Modern Approach. Prentice-Hall, Egnlewood Cliffs (1995)
12. Penrose, R.: On best approximate solutions of linear matrix equations. Mathematical Proceedings of the Cambridge Philosophical Society, 52, pp. 17–19. (1956)
13. Bishop, C.: Pattern Recognition and Machine Learning. Springer (2006)
14. Gherrity, M.: A game-learning machine. PhD Thesis, University of California, San Diego (1993)
15. Allis, L.W.: A knowledge-based approach of connect-four. Technical report, Vrije Universiteit, Subfaculteit Wiskunde en Informatica (1988)
16. British Othello Federation: Game Rules, `http://www.britishothello.org.uk/rules.html`
17. Cirasella, J., Kopec, D.: The History of Computer Games. CUNY Academic Works, New York (2006)
18. Mitchell, D.H.: Using features to evaluate positions in experts' and novices' Othello games. Masters Thesis, Northwestern University, Evanston (1984)
19. MacGuire, S.: Strategy Guide for Reversi & Reversed Reversi, `www.samsoft.org.uk/reversi/strategy.htm#position`
20. Mnih, V., Kavukcuoglu, K., Silver, D., et al.: Human-level control through deep reinforcement learning. Nature 518, no. 7540, pp. 529–533. (2015)