Assignment Report: Othello AI Agent

Group 32

Wei Chen, Yijie Lu, Manlu He, Xiaonan Li

Modern Game AI Algorithms
Leiden Institute of Advanced Computer Science
The Netherlands
10-06-2022

Abstract

Super human playing strength in the game of Go has been achieved by AlphaGo Zero which integrates MCTS with neural network. Using data generated from self-play, the agent is trained by reinforcement learning without any human knowledge. This paradigm inspires us to explore its possibility in other games. In this project, we implement a game simulator with GUI that provides two playing modes in a board strategy game called 'Othello', where a human player can play against another human player or an AI agent that uses two types of neural network based on the main idea of AlphaGo Zero. This report covers the details of our implementation, including the main methods of the AI agent and GUI designing. In order to investigate the performance of our agent, we test it by letting it play against random human players and comparing it to other computer programs. In the matches against random human players, our agent is able to beat top 20% players on a game platform but still loses several matches. Also, after 20 rounds of matches, our agent is not comparable to top Othello programs, NTest and Zebra. The results might indicate that our agent learns some basic game-playing strategies such as maximizing the number of pieces in each move, but it is not able to achieve expert level performance which requires the ability of recognizing the pattern to select optimal moves, which is encoded by the opening book and evaluation function that those top programs leverage.

Contents

1 Introduction

1.1 Project objective

In this project, the game we study is the two-player board game Othello, and our project objective consists of two parts: the first part is to implement an Othello simulator with GUI interface, and the second part is to create an AI agent that can play Othello. At the end of the project, for the first part, we use *tkinter* [14] to successfully implement a simulator with GUI that has the rules and basic logic of the Othello built in and provides two game mode: human player versus human player, human player versus computer (our AI agent). For the second part, we are inspired by AlphaGo Zero [21], applying applying deep neural network (CNN, ResNet) and Monte Carlo Tree Search techniques to the agent and training the model using data generated by self-play. Afterwards, we evaluate the performance of our trained agent in a variety of ways, such as multiple play against strangers of different levels on a game platform and against Othello classic computer agent such as Ntest.

Our project is related to three lectures content: learning and optimization, Monte Carlo Tree Search, ANN and deep learning. Through this project we hope to stimulate interest in Othello and apply what we have learnt in a real situation.

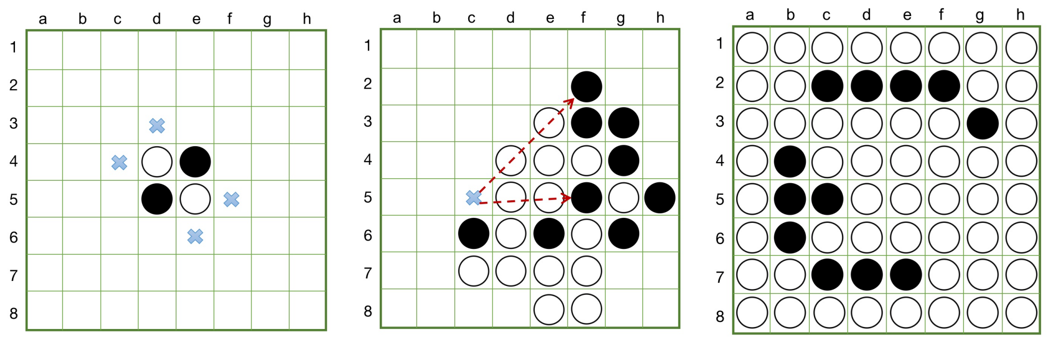
In the paper, we first define our research objective, introduce the rules of Othello and the progress of the board game computer agent. Then we briefly introduce AlphaGo and AlphaGo Zero. Next we detail the methodology of our agent and simulator implementation. In order to show the performance of our agent, in the section4 we report the results of our evaluation. And finally, we conclude our project and give the directions for further study and improvement.

1.2 Othello rules

Othello (also known as Reversi) is invented by the Engishmen Lewsi Waterman in the 19th century and developed and popularized by the Japanese Goro Hasegawa in the 1970s [6].The rules of Othello are simple and easy to start, but the changes in the state of the board are complex, therefore mastering the game is difficult and requires a lot of time to study and consider.

The Othello board size is 8×8 and has a set of reversible pieces with one side is black and the other is white, which are divided between two players. It is a two-player, zero-sum board game with perfect information. Next we briefly describe the basic rules of Othello:

- Black pieces move first, and both players move alternately.
- The initial board state of this game is that each player takes two pieces and places them crosswise in the center of the board, as shown in Figure 1a, with the white pieces on position $(4, d)$,$(5, e)$ and the black pieces on position $(4, e)$, $(5, d)$.
- A legal move is: placing a piece in an empty position, each move must outflank at least one different color piece, i.e. there must be at least one different color piece between two pieces of the same color. In Figure 1a, the positions marked with $'x'$ allow for the placement of black pieces, as they can outflank at least one white piece.
- The outflanked different color pieces should be flipped, and one move can cause the flips in several directions. As shown in Figure 1b, if a black piece place on position $(5, c)$, all white pieces passed by the red dotted line will be flipped, no more or no less are allowed.
- Unless at least one of the opponent's pieces can be flipped, no move can be made. Therefore if one side has no position to place, pass a move and allow the opponent to place in succession.
- The game ends when both players have no position to place pieces. The game ends not only when all 64 positions have been placed, but also when there are still empty positions on the board, but neither player can place a piece. The winner is judged by the number of pieces on the board at the end of the game, the player has more pieces on the board wins. As shown in Figure 1c, white wins.



(a) Initial board state. (b) Place a black piece and flip. (c) End board state.

Figure 1: Examples of the board state during an Othello game.

1.3 Computer playing board games

Board games is an interesting and realistic environment for AI agent development, training and evaluation. In the days before computers are common, i.e. before the 1880s, the strongest players in chess games are humans.

In the early 1990s, the strongest computer player *Thor* in Othello at the time is emerged. Its approaches are relatively traditional, with engineers encoding positional strategy, even importance, etc. into the program.

Around 1994, a platform emerged that allowed various computer agents to play against each other, called IOS. And in the same period, the computer agent Logistello [3] created by Michael Buro published. The Logistello begin to adjust strategies based on the results of the game, evaluate the opening books and save them. From 1993 to 1997, Logistello played in 25 races, winning 18 and finishing second in 6. At this point, there has a milestone in the research of computer player of Othello.

In 1996, the "Deep Blue" [7] computer created by IBM team, based on brute force search, defeated world champion Garry Kasparov in Chess.

In 2016, AlphaGo [20], an AI agent developed by Google DeepMind team, won a Go game with a huge search space, defeating the world's second-ranked Lee Sedol. By applying deep learning and Monte Carlo Tree search techniques, and trained using millions of games of grandmaster history, it managed to solve Go problems that were said not be solved for decades.

In 2017, a year after the release of AlphaGo, the AlphaGo Zero [21] also created by Google DeepMind team, defeats Ke Jie, who is the world's highest-ranked Go player. It uses no human knowledge during the entire training process and AlphaGo Zero outperforms AlphaGo. This means that the computer can surpass humans level through self-learning, which is an exciting breakthourgh. We take inspirations from AlphaGo Zero, and we try to apply the ideas of them to our Othello AI agent. In next section, we briefly describe the thinking behind AlphaGo and AlphaGo Zero.

2 Related Work

2.1 AlphaGo

AlphaGo [20] is an artificial intelligence agent based on deep neural networks and MCTS developed by the Google DeepMind team to play Go. Go is a difficult board game for computers to master. Go has the following properties due to its rules:

- The state space size of Go is about 2×10^{170}. It is huge, therefore it is impossible to solve it by brute force search.
- As the game progresses, the number of pieces on the Go board increases, so using a game sheet with sequence of moves allows us to save most of the information about the moves.

As we analyzed above, the state space of Go is huge that it is impossible to enumerate, but the human brain can solve this problem. Therefore, the algorithm tries to generalize the laws of human player such as some heuristics, evaluation of the future state the game and so on. So the AlphaGo implements the algorithm in two stages: the first stage is to master the human strategy, the second stage is to evaluate the evolution of the game after this move, optimizing and determining the final position of the move based on the evaluation results.

For the first stage, what AlphaGo needs to do is to work out which position(19×19) a piece should be placed at each move of the game. This problem can be viewed as a multi-classification problem. According to the transformation from a multi-classification problem to a binary classification problem, at each specific position on the board, we treat it as a one-to-rest problem. Therefore 361 classifier are needed. Each position on the Go board has and only has three cases: black piece(1), white piece(-1) and no piece(0). The input to each classifier is the current state of the board, which is a 361-dimensional vector. Due to the non-linearity of the input features, AlphaGo uses the deep neural network with the ability to handle high-dimensional features. And some new features (based on Go domain knowledge) are added to simplify the model and training. The final dimensionality of the input features is $361 \times n$. With good feature engineering, AlphaGo gets the good classifiers using only 13-layer convolutional network. The output of each classifier is the score the piece placing on the position the classifier represents. From the outputs of these 361 classifiers, AlphaGo finds the position with the highest scoring as the selected place position.

Google DeepMind team created two versions of the classifier: supervised learning policy network p_θ and fast policy p_π. The supervised learning policy network is able to match 57% of the human behavior, the fast policy is able to match 24% of the human behavior, but is 1000 times faster than supervised learning policy network. These two models serve to simulate behavior but do not evaluate wins and losses.

The data for training the models is from the Go game platform KGS. DeepMind obtains 160,000 games of historical data of high-level human players from this platform. They use 30 million data pairs $< s, a >$ by considering the state of the board before each move as feature and the move as label. Using this data set, the models are trained.

After experimentation, both models perform poorly. This is caused by the fact that the evolution of the game after a move is made and the win or loss is not taken into account when approximating the value function. Therefore, DeepMind team optimize the evaluation function with win-lose criterion. This is done by:

1. adding new labels 1, -1 and 0 to indicate loss, win and draw.
2. A new value network is created, the input to the network is a 361 $\times$n dimensional vector and the output is the win rate. The value network is trained using the samples generated by self-play.

DeepMind team combine MCTS based on value network and MCTS based on fast policy to balance the advantages and disadvantages of the two approaches. Monte Carlo Tree Search is a traditional technique, if without the use of prior knowledge, the depth and width of the tree can be very large and computationally expensive. DeepMind team prune the MCTS through policy network, and the value network enables the MCTS to achieve better performance with a less deep search.

Finally, we can summary that the AlphaGo consists of four components:

1. Supervised policy network: given the current state of the board, predicting the next move.
2. Fast rollout: fast simulation behaviour at the expense of some quality.
3. Value Network: given the current state of the board, predicting whether white wins or black wins.
4. MCTS: combining the above three components.

2.2 AlphaGo Zero

In the year following the release of the AlphaGo, Google DeepMind team publish a new paper in the science journal *Nature* detailing their new creation, AlphaGo Zero [21]. It relies on no human knowledge, and can beat the master version of AlphaGo after only three days of training.

The main differences between AlphaGo and AlphaGo Zero are:

- AlphaGo Zero does not use any human empirical knowledge, the agent learns entirely from self-play.
- No hand-encoded features, the input to the network is only the board state.
- The policy network and value network introduced in AlphaGo are combined into one network, i.e. policy network and value network share parameters. And the structure of the network is improved to two-head ResNet, with outputs of policy and value respectively.
- Remove fast rollout and evaluation with neural network only.

Now we briefly describe the network training process. From the start of training, the system self-play and collects data using the current optimal network. The data includes: the board state and the probability of each move in the current board state, the score of each game and the cumulative score. Then random sampling is carried out from the generated data set and the sampled data is used to iteratively update the network, and then the updated network is evaluated. The evaluation is done by making the updated network play 400 games with the pre-update network, and only when the win rate exceeds 55%, setting the optimal network to the updated network.

The emergence of AlphaGo Zero is of great significance. It learns from scratch by playing against itself and does not require any human knowledge. This means that this system has the potential to solve any board games. Also, it shows to extent the future of deep learning. It is proving that even without the support of huge amounts of labelled data, computers can still learn by generating data through simulators (e.g.MCTS). And in some cases, it might be possible to surpass the highest human level.

3 Methods

3.1 Technology

For implementation, our program is entirely based on *Python*, where we use *Pytorch* to build the classic convolutional neural network(ConvNet) and residual neural network(ResNet), while the graphical interface is designed with the help of *Tkinter*.

3.2 Monte carlo tree search

In our program, the MCTS algorithm we implemented is basically the same as the classic one. The main different point is on the simulation step where a neural network is introduced. Moreover, to build the search tree, we define a class featuring a node that represents the state in a game, pointing to its parent node and children nodes (could be None). Specifically, a node object here stores the number of times the node has been visited N, the prior probability of selecting the node itself P, the mean action-value Q, the total action-value W, and its *parent* and *children*, where the Q value is calculated by dividing W by N, while the W value is obtained by accumulating the state value (computed by neural network) of its children nodes. Most importantly, we create the unique identification of a state with the help of the move ID (0-63) that corresponds to the position of the board, where the format of the *children* property for a node is as $\{MoveId : Node\}$.

3.2.1 Selection

The first step of our MCTS is selection which starts from the root node, traversing down until a leaf node is reached (as shown in Figure 2) where the root node represents the current state of the game but is not necessarily the starting state in which there are four pieces initially. Furthermore, there is a subtle nuance in selecting a node. A node is selected by maximizing its UCT value (see Equation 1), i.e., upper confidence bounds applied to tree [13], plus its Q value (see Equation 2), instead of depending on UCT merely.

$$U(S,a) = C \cdot P(S,a) \cdot \sqrt{\frac{\sum_b N(S,b)}{1+N(S,a)}} \tag{1}$$

$$a = \underset{a}{\operatorname{argmax}}(U(S,a) + Q(S,a)) \tag{2}$$

where the S represents the current state of a game, a is a node (or could be seen as a branch) in the search tree, b are the sibling nodes of the node a (they have the same parent), $P(S,a)$ indicates the prior probability of selecting node a itself, and coefficient C determines the extent of the exploration when selecting.

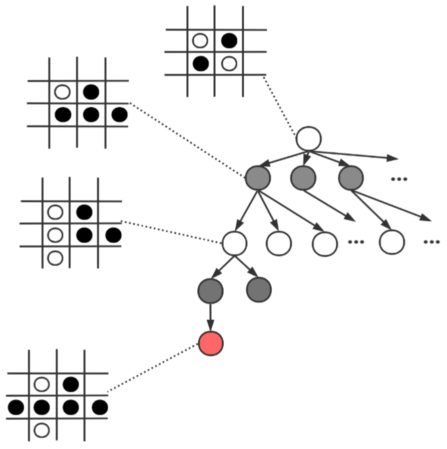


Figure 2: The selection step of the MCTS, where the topmost node of the tree is the root node which does not necessarily stand for the initial game state, and the red node is a leaf node not having any children. Each node in the tree represents a state of a game, while the ellipses are those nodes that are not shown in this figure.

3.2.2 Expansion and simulation

Once a leaf node is encountered, its children nodes are created according to the number of the next available actions(moves). Next in the simulation step, the game state that such leaf node represents, would be brought to a neural network, which always predicts the scalar *state value* and the vector *prior probabilities* (as shown in Figure 3). The state value and the prior probabilities would be then accordingly assigned to the leaf node itself and its expanded nodes (in one-to-one correspondence), respectively.

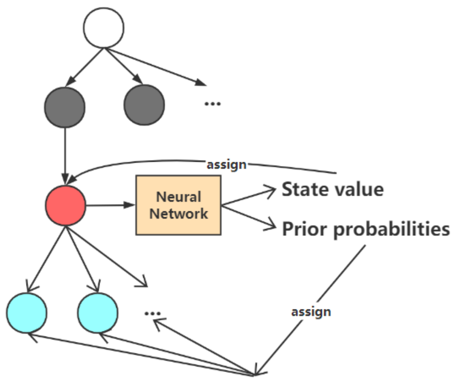


Figure 3: The expansion and the simulation steps of the MCTS, where the topmost node of the tree is the root node, the red node is a leaf node, and the blue nodes are the expanded nodes(children nodes) of the leaf node. Once a leaf node is reached, all available children nodes of it are expanded and a neural network would take as the input the game state that such leaf node represents, and outputs two values.

3.2.3 Back-propagation

The back-propagation step always starts after the simulation and expansion steps. The W value of the leaf node would be assigned the state value first as stated above. Next, all ancestor nodes of this node, are each updated by adding the state value (as shown in Figure 4), where this value is negative for its direct parent and is positive for its grandparent, and so on.

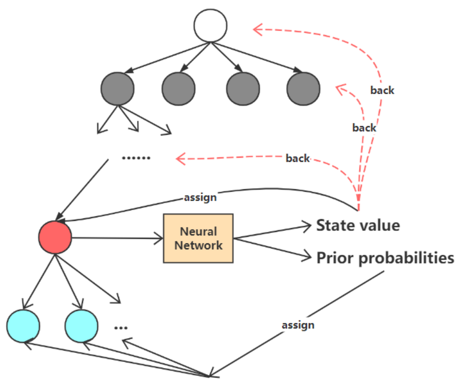


Figure 4: The back-propagation(back-up) step of the MCTS, where the topmost node of the tree is the root node, the red node is a leaf node, and the blue nodes are the expanded nodes(children nodes) of the leaf node. After the simulation step, the state value predicted by a neural network would be then inversely iterated from the leaf node to the root node.

3.3 Exploration and exploitation

In terms of reinforcement learning, most policies tend to be added an exploratory mechanism to allow for more possibilities thereby enabling the agent to better understand the environment in which it is placed. In addition to UCT, we also introduce two other methods to be a trade-off between exploration and exploitation, to make the data generated with self-play more diverse in our program.

The one is to apply the temperature parameter τ after a certain playouts. At this point, the agent chooses a node among the nodes in the second layer, as a real action(move), according to the number of times each node has been visited (see Equation 3).

$$\pi(a|S) = \frac{N(S,a)^{\frac{1}{\tau}}}{\sum_b N(S,b)^{\frac{1}{\tau}}} \tag{3}$$

where N(S,a) means the number of times the node a has been visited and nodes **b** are the siblings of node a. Lastly, the temperature parameter τ is leveraged to control the level of the exploration.

In practice, the other exploratory method is achieved by introducing Dirichlet noise to the property *prior probabilities* of the expanded nodes after the simulation step. The prior probability predicted by the neural network could be re-computed as the Equation 4 below:

$$P(S,a) = (1-\epsilon) \cdot \pi(a|S) + \epsilon \cdot \eta_a \tag{4}$$

where $P(S,a)$ is the prior probability of selecting the node a, while *pi* $\pi(a|S)$ is the probability of selecting node a under the condition state S. Additionally, ϵ and η_a are free parameters where ϵ is generally set to 0.25.

3.4 Neural network

3.4.1 Input features

The input features need to be able to comprehensively represent the state of the board at any time, including its history of previous moves and derivative metrics from the state of the board [12]. In the process of designing the feature plane, we refer to some existing work, and finally we decided to use three feature planes to describe the existing pieces of the board, instead of using some more complex features such as move history, legal moves, etc. We use an 8×8 matrix and use one-hot encoding to represent the feature plane. In this way we can easily and directly transfer the information of the chess game to the convolutional neural network.

We design three concise feature planes to describe the game information. The first feature plane describes the positions of black pieces, and the 8×8 matrix corresponds to the 8×8 Othello board. The eigenvalue of the corresponding positions of black pieces are set to 1, and the rest of the positions will be set to 0. The second feature plane is used to describe the positions of white pieces, similar to the first feature plane, the eigenvalue of corresponding positions of white pieces in the matrix is set to 1, and the eigenvalues of all other positions are set to 0. The third feature plane mainly describes the position information of the pieces that can be placed on the board, the eigenvalues in the matrix corresponding to the positions on the board that have not yet been placed are set to 1, and the eigenvalues of other positions are set to 0. By combining these three feature planes, that is, these three 8*8 matrices, we can use them as the input of the neural network, which is a good way to pass the board information to the network while retaining its spatial characteristics .

3.4.2 Architecture

We design two network architectures, one is mainly composed of convolutional neural network, and the other is a residual network with residual modules added in the middle. The input of the network is the feature planes designed by the Section3.4.1, and the network outputs two values, one is the state value of the leaf node in MCTS, and the other is the prior probability of all nodes expanded under this leaf node. These two outputs can be combined into the MCTS algorithm to help the agent make decisions(as shown in Figure4).

Since we need to maintain the spatial properties of Othello game represented by the feature plane as much as possible [12], it is most reasonable to use a convolution module to process the input. The structure of the convolutional neural network model we designed is shown in Figure5.

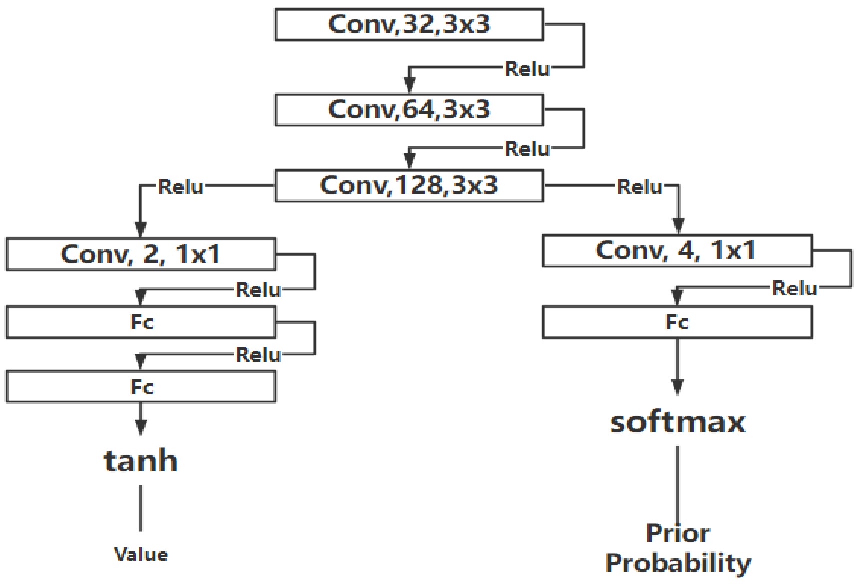


Figure 5: Architecture of CNN.

For the design of the residual network, we still use the convolution module when processing the input, However, we replace the convolution modules in the middle of CNN structure with residual modules, the structure of the residual network we designed is shown in Figure6.

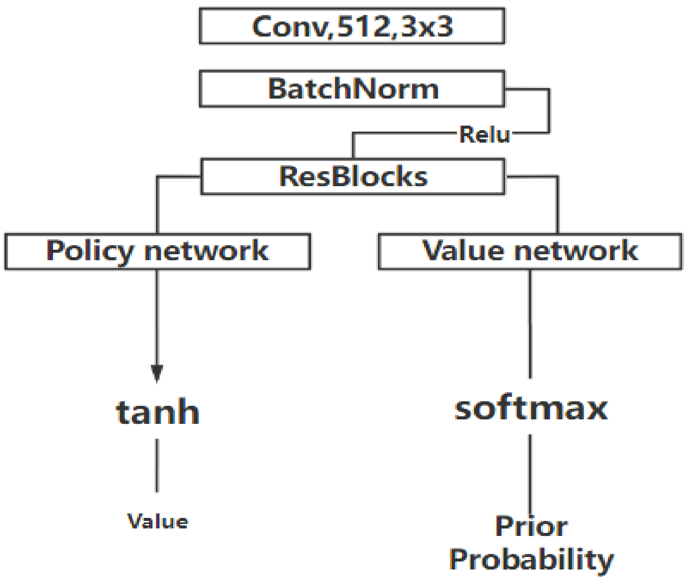


Figure 6: Architecture of ResNet.

The reason why we think of introducing the residual modules is mainly to solve the problem of network degradation caused by the increase of network depth. The convolution modules have been used to process the input features very well, but as the depth of network increases, the problem of gradient disappearance and gradient explosion may occur. The residual module adds shortcut connections and identity mapping, which can allow us to make the network deeper while maintaining accuracy and reducing the amount of computation. We added the Batch Normalization layers over residual modules, which can also help us solve the problem of gradient disappearance and gradient explosion.

3.5 Agent

Whether it's a self-play mode that generates game data for training or a human-versus-AI mode, agents need a strategy to choose a real action, or precisely, to make a decision. We borrow our idea from AlphaGo Zero; thus, our agent combines the MCTS algorithm with a neural network to do decision-making (as shown in Figure 7). As described in Section 3.2.2, a neural network is applied in our implementation of MCTS, to take a state as input to conduct a simulation instead of adopting the rollout policy, in which the latter tends to obtain an inaccurate result. Typically, each training is started after a certain amount of self-play. In other words, the neural network is leveraged for a period of time to first guide the MCTS. Then the generated data would be brought to improve the neural network.

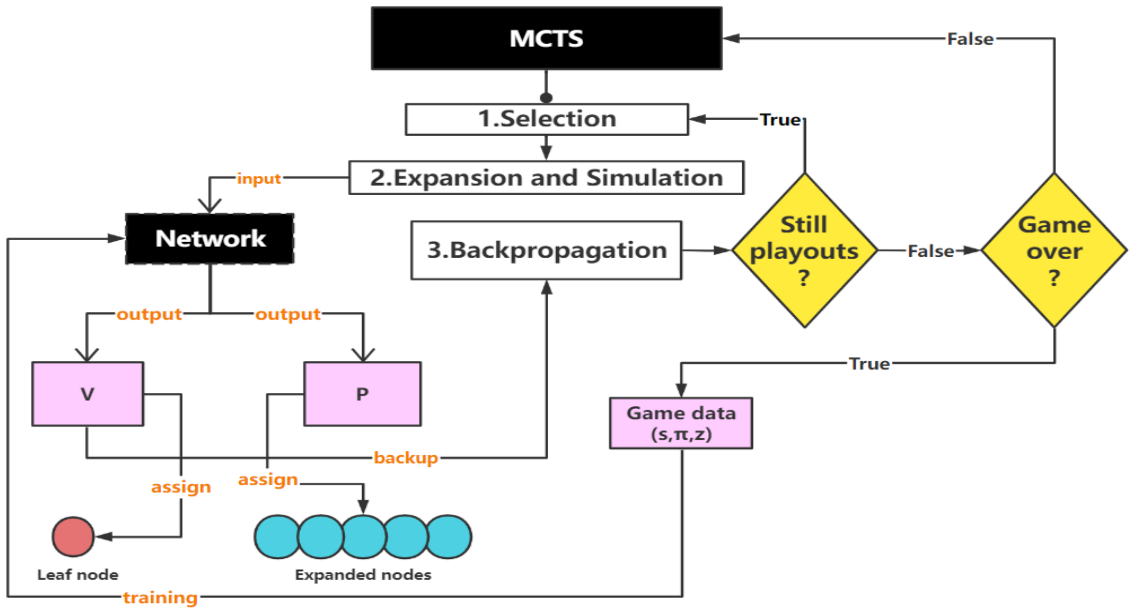


Figure 7: Decision made by our agent using MCTS which is combined with a neural network, where the v and p is the scalar *state value* and the vector *prior probabilities*. Game data is generated until a game is over. The generated data consists of the set of the states S, set of the *pi* $\pi(A|S)$, and set of the results Z for each time-step (or state), where actions A represents all nodes (or branches) in the search tree.

3.6 GUI

3.6.1 Tkinter

The tool we use to build the game GUI interface is called *Tkinter*, which is a standard Python interface for the Tk GUI toolkit. Tk is a lightweight cross-platform graphical user interface development tool that can run on various systems, including Unix platforms, Windows platforms and Macintosh systems [2]. Its code is concise and easy to read, and it can be well matched with our subsequent game logic development and agent development, so we choose *Tkinter* as our GUI interface development tool.

Tkinter is the officially recommended GUI toolkit, which belongs to the standard library that comes with Python, so it has high coding efficiency and can achieve the purpose of rapid development. Although *Tkinter* is not usually used to develop complex applications due to the nature of interpreted language, it contains common features of GUI packages, including different types of window controls, window layout managers, event handling mechanisms, etc. In the development of the Othello game interface, we summary the functions of the GUI software package we need to use, which mainly involve three parts, one is the drawing of the board, one is the window control for selecting the human versus machine and the color of the pieces and the last one is player's processing of mouse click events when playing chess. These needs can be achieved by *Tkinter* easily.

3.6.2 Interface design

The interface design of the game is shown in the Figure8. The layout of the entire game interface is mainly divided into two parts, one is the board part on the left, which is mainly responsible for displaying the situation of the Othello game. The other part is the control part on the right, which is mainly responsible for selecting the color of the pieces and the game mode (human versus AI/ human versus human). At the same time, an area is designed to display the current game state, which can remind the player who should make the move at this time. "Waiting" is displayed in this area when it has not started or otherwise requires no prompting.

The drawing of the board is done through Tkinter's Canvas class. When creating a Tkinter object, we fix the width and height of the window, so that the size of the window cannot be changed by dragging and dropping the mouse, which facilitates the conversion between the actual coordinate position in the board and our custom abstract coordinate. Through the methods in Canvas, we can easily create shapes such as squares, lines, and circles in the window. The Othello board is an 8×8 grid, with the pieces placed in the center of each grid. We first determine the side length of each small grid in the board and draw a large square as the border of the board. Then we establish the conversion relationship between the actual distance in Canvas and the abstract coordinates we set. For example, the abstract coordinate corresponding to the grid in the upper left corner is (0, 0), and the abstract coordinate corresponding to the grid in the lower right corner is (7, 7). The coordinates of a grid are based on the vertex of the upper left corner of each grid. With the conversion relationship between actual coordinates and abstract coordinates, we build a basic board grid through various line drawing in Canvas, and then realize the drawing of pieces by drawing circles and filling colors. In the initial board, according to the rules of Othello, we draw four pieces of alternating colors in the four grids at the center of the board in advance.

The GUI interface in the process of playing Othello requires us to process the mouse click event of the player, which can also be easily implemented in *Tkinter*. When we click the board with the mouse, we can obtain the coordinates of the mouse click position through the processing of the click event by Tkinter. And then we can change this coordinates to the abstract coordinate in the form like (0,0), and draw a piece according to this abstract coordinate. The color of the chess piece is easy to implement, set the fill color property when drawing the circle. However, the selection of the color of the pieces and the discoloration of the existing pieces on the board need to be judged according to the game logic and the information of the Othello board, which is explained in Section1.2.

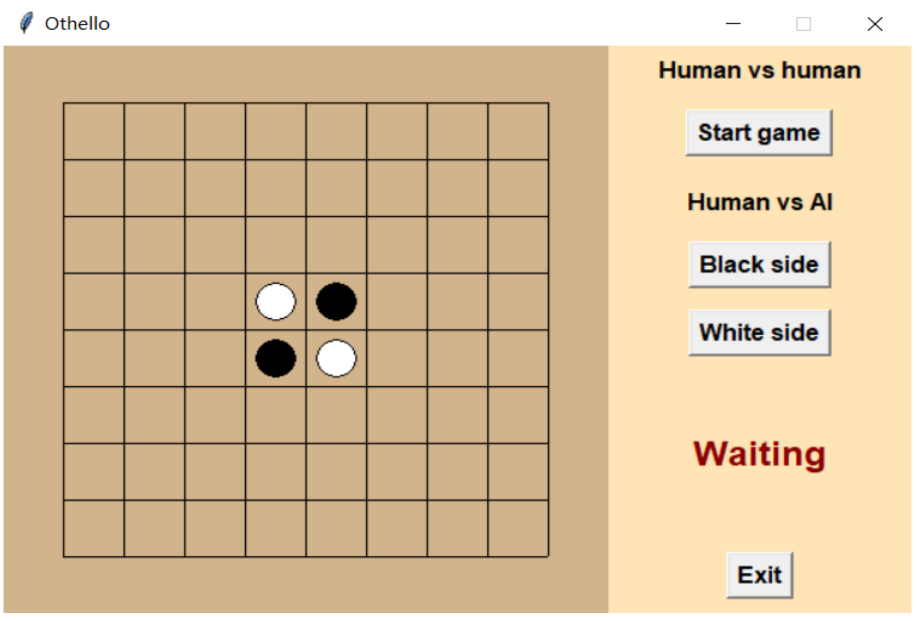


Figure 8: Interface design.

4 Evaluation

In this section, we let our agents play against each other to compare their performance. Then we select the better one for further evaluations, which include matches against random human players, matches against classic top Othello programs and some other agents.

4.1 Model battle

As we mentioned in Section 3, we implement two types of agents using traditional CNN and ResNet, each of which is trained with 2000 self-play games. In total 50 games are played between CNN based agent and ResNet based agent with the numbers of the MCTS play-out set to 400, the CNN agent wins 25 of the games and another 25 games end in draws. Given this outcome, we consider this CNN agent is more likely to achieve better performance than ResNet agent and thus be chosen as the optimal model for the following evaluations.

4.2 Human vs. our agent

To investigate our agent's performance against human players, inspired by the work of [17], we randomly play matches using our agent with the numbers of the MCTS play-out set to 400 against human players on a mobile application called 'OthelloQuest'.

In Figure 9, we present the results of 20 matches between our agent and 20 anonymous players on the mobile app. For each game, the time limit is set to 5 minutes, and the color of our side alternates between white and black. Both human players' and our agent's Elo rating [24] are automatically calculated by this application. The Elo rating assigned to our agent changes as games are played and remains between 800 to 860, which is approximately ranked in the top 60% among 47083 players on this platform according to its statistics. Our agent wins 7 matches, and it is worth noting that it wins 3 matches against players with Elo rating around 1400 and 2 matches against players with ELo rating around 1300, who are in the top 20%. This might indicate that our agent learned some game-playing strategies and can perform properly.

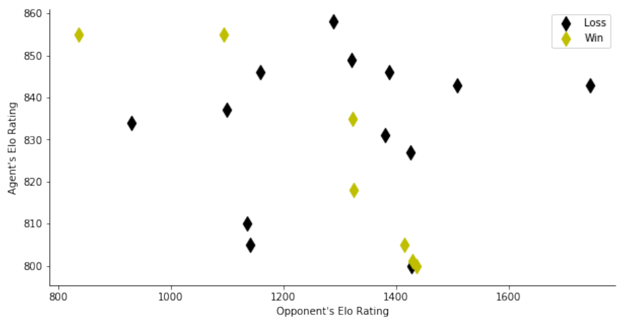


Figure 9: Agent's performance against random players on OthelloQuest.

4.3 Classic agents vs. our agent

In order to further explore our agent's playing strength, we test it against two top Othello programs: NTest and Zebra.

According to [4], many classic agents achieve strong performance using these main techniques: evaluation functions, selective search and opening book. A proper evaluation function guides the 'looking ahead' of the searching procedure by assigning scores to the board position [4]. Designing such an evaluation function for a specific game is a non-trivial task because it requires game knowledge where the features of the board state need to be carefully selected. When searching for next moves, an ideal selective search algorithm can identify promising area and abandon probably irrelevant branches [4]. The utilization of the opening book is game-dependent and aims to improve the program's performance in the opening phase [4]. Both NTest and Zebra leverage these three techniques.

4.3.1 NTest

The design of NTest's evaluation function consists of pattern coefficients, mobility, potential mobility and a constant. The pattern coefficients are learnt by a large sparse linear regression that treats pattern features as binary variables and fits the position labels [5]. Mobility is the number of valid moves that the current player can make, and potential mobility is the number of empty positions that are adjacent to at least one opponent's piece [11]. A constant is used for avoiding even/odd ply bounce, which is an effect predicted by theory for imperfect evaluation functions [22]. The tree search is done by the alpha-beta algorithm with the help of an opening book which stores seeking draws or wins value, avoiding draws value and best guess of the game outcome [22].

NTest offers agents of different levels based on the search depths. We let our agent play against $NTest_4$ and $NTest_8$, which denotes search depth of 4 and 8 respectively. For each variant of NTest, 5

	Round 1	Round 2	Round3	Round 4	Round 5
NTest4	58/4	58/6	58/4	62/2	57/7
NTest8	63/1	62/2	64/0	61/0	62/2
Zebra4	53/11	56/8	53/11	52/12	48/16
Zebra8	58/3	58/5	58/6	50/14	54/1

Table 1: Results against classic agents

rounds are played. The final outcome is presented in Table 1, in which '58/4' denotes the opponent has 58 pieces and our agent has 4 pieces when the game ends. It is obvious that NTest performs significantly stronger than our agent with search depth set to 4 and 8.

To find out possible reasons leading to this result, we analyse a middle game from one of the matches, which is shown in Figure 10. Here the stage is the number of the occupied positions on the board, and our agent is the white side. At the initial stage, our agent tends to fills more squares (stage 29). However, NTest then takes the corners and the edges adjacent to them (stage 34, 45), which relates to the patterns and heuristics (positional strategies) [16], and finally flips more pieces to win the game. We discuss our agent's weakness might be caused by the fact that the amount of training deployed to our model is not enough to let it learn these 'tricks'.

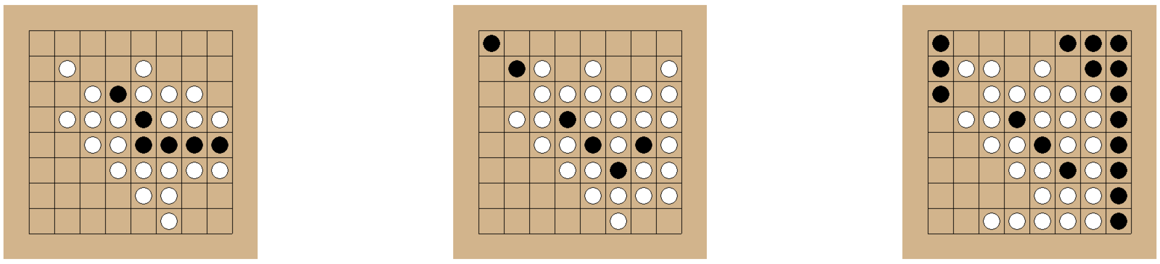

(a) Stage 29 (b) Stage 34 (c) Stage 45

Figure 10: Board states during a match between NTest and our agent.

4.3.2 Zebra

Like NTest, Zebra also leverages a pattern-based evaluation function in which pattern configurations related to line patterns and corner regions on each pre-defined game stages are tuned by a linear regression [1]. Two search algorithms, NegaScout and Multi-ProbCut (MPC), are used for selective search. NegaScout is a variant of the alpha-beta algorithm to increase the cutoffs in the game tree [10]. Based on the fact that values obtained from minimax searches from different depths are strongly correlated [4], Multi-ProbCut uses shallow search to guide the value of alpha and beta of a deeper search [10]. As for Zebra's opening book, it is learnt from games between top players and self-play games [1].

The search depths that Zebra considers when looking for the best move can also be specified and we set them to 4 and 8. As can be seen from Table 1, Zebra is superior to our agent over all the rounds. Just like the matches against NTest that we mentioned above, our agent works quite well at first but Zebra manages to flip its opponent's pieces at the end by capturing valuable positions.

4.4 Other agents vs. our agent

The results against the top Othello programs are not surprising as both Ntest and Zebra are significantly stronger than a previous top Othello program that beats a former world champion in 6-0 [23]. We also are curious about our agent's strength compared to other agents, so we find another two Othello agent from Github to let them play against our agent.

The first opponent agent [8] is build upon the three main techniques that those top Othello programs leverage. The evaluation function is constructed using heuristics by linearly combining the following features [8]:

	Round 1	Round 2	Round 3	Round 4	Round 5
Version 1	48/16	49/15	50/14	53/11	50/14
Version 2	56/8	58/6	58/6	59/5	60/4

Table 2: Results against other agent 1.

- Mobility and potential mobility: these are the same as NTest;
- 'Makeup': the number of current player's pieces divide by the number of opponent's pieces;
- Stability: it calculates the ratio of the number of the stable pieces gained by the current player to its opponent's;
- Parity: used for predicting which player will make the final move.

The search algorithm is alpha-beta with iterative deepening and killer heuristic (history heuristic), which saves optimal path (moves) as the starting positions for the next search and thus leads to alpha-beta cutoffs and allows deeper searches [18]. An opening book database is used for guiding the first 9 turns and the search algorithm is called if no valid moves can be obtained from the database [8].

Unlike the class agents that the playing strength are parameterized by the search depth, levels of this agent is set by the time limit for performing the search. Our agent takes around 0.3 seconds to make a move so we set the time limit of this opponent agent to 0.3s as version 1 and set it to 1s as version 2. For each version, we play 5 matches against it and the results are shown in Table 2. This agent outperforms our agent as version 1 and becomes stronger as version 2 when more searches are performed. This indicates that an Othello program with good playing strength might be implemented by leveraging the three main techniques.

The second simple Othello agent [19] we used for performance comparisons are created by a student team from the University of Hong Kong. In this agent, the authors use the Iterative-Deepening Minimax Search algorithm and $\alpha - \beta$ Pruning technique. In practice, the authors first set the search depth of the algorithm to 3, and then increase the search depth as much as possible within the time limit. In order to ensure search efficiency, the authors use pruning techniques to make the algorithm search only for valuable moves.

Furthermore, the authors of this agent, like the authors of many other agents, add the heuristic based on domain knowledge of Othello into the program. The evaluation function defined by the authors for assigning scores to the nodes of the tree is the sum of scores of the following five components(assuming that the computer agent holds black pieces and the human player holds white pieces):

- Difference in the number of pieces on the board: Count the number of pieces of each side of the game. If the number of black pieces $A >$ the number of white pieces H, the score is $\frac{A}{A+H} \times 100$; if the number of black pieces $A <$ the number of white pieces H, the score is $-\frac{H}{A+H} \times 100$; If $A = H$, the score is 0.
- Number of pieces in the corners of the board: Since pieces in the four corners of the board can not flipped by the opponent. As a rule of thumb, the more corners you occupy, the better the chances of winning. Therefore the authors evaluate the difference in the number of pieces in the corners of the board between the two players. The score is defined as $25 \times$ *the difference between the number of black and white pieces in the corner positions*.
- Number of pieces close to corners: the score is defined as $12.5 \times$ *the difference between the number of black and white pieces close to the corner*.
- Number of legal moves: Measure how many legal moves each player of the game has in the current state of the board. If the number of computer agent's legal moves $A >$ the number of human player's legal moves H, the score is $\frac{A}{A+H} \times 100$; if the number of computer agent's legal moves $A <$ the number of human player's legal moves H, the score is $-\frac{H}{A+H} \times 100$; if A=H, the score is 0.

- the Stability of a piece: this is a very important indicator. It shows how easily a piece can be flipped. The value of stable, semi-stable and unstable piece are 1, 0, -1. This means that the more stable pieces a board state contains, the higher the score.

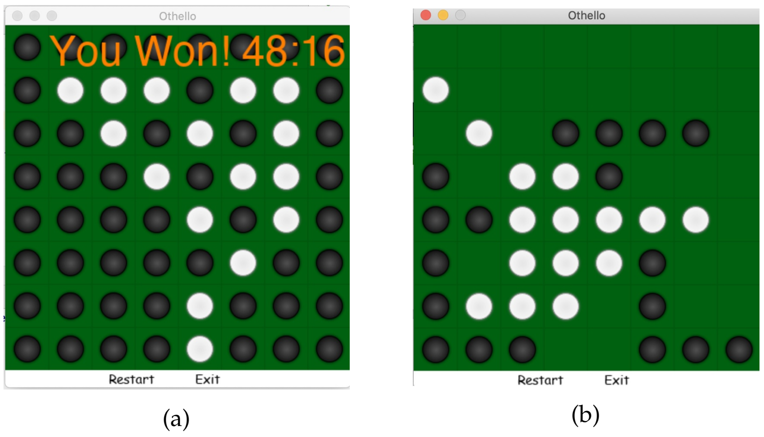


Figure 11: Our agent versus the agent created by the student team from HongKong University.

Also, in order to further improve the efficiency of the algorithm search, the authors built into the program a library of actions that define the actions that definitely be executed in a certain board state to avoid wasting computational resources.

We play our agent against this agent, with our agent holds the black pieces and this agent holds white pieces, our agent eventually win the score of 48:16, as shown in Figure 11a. By observing the pattern of moves on both sides, we find that our agent is able to win probably because our agent not only learned how to flip as many of his opponent's pieces as possible by training, but also learned some valuable strategies that may better than the evaluation function manually coded by the author of agent.

5 Conclusion & Further work

In this project, we implement an Othello program based on the the main idea of AlphaGo Zero that combines MCTS with neural network for self-play reinforcement learning. Our CNN model outperforms our ResNet model, and is evaluated by matches against various players e.g. humans and computer programs. On an Othello game platform, our agent is considered as an intermediate player and wins several games against top 20% players, meaning that through self-play, our agent can learn game-playing knowledge to some extent. However, it still losses many games against random human players, and is not able to beat top Othello programs. Also, we notice that our agent tend to select moves by maximizing the number of discs, which is a mistake made by most beginners [15]. We may say that these outcomes reveal that our agent is not trained enough to learn how to wisely infer moves from the patterns embodied in the board states while those human players and top programs might have already acquired the ability to make the best moves by recognizing patterns.

Inspired by some related previous work, we discuss that there are improvements that can be done as our future work. Firstly, since that AlphaGo Zero is trained using 4.9 million games [17], we can imagine a performance gain of our agent if we have better computational resources and more training time. However, according to the work of [17], it seems that master level performance in the game of Othello could be achieved with limited resources by modifying AlphaGo Zero's paradigm. One of the future work can be that we not only consider the board states from real self-play when creating the training set, but also include the board states that has been explored more often [17]. This is helpful in reducing the cost as the self-play phase is most expensive [17]. Another modification can be made to the MCTS by increasing the number of the play-out after certain number of self-play games are played. Secondly, hyper-parameter tuning and changing the architecture of the neural network may lead to better performance of our agent if we can experiment intensively. Thirdly, we can explore more input features such as the history of previous moves that are used in AlphaGo zero, or the player mobility and stability used in the work of [9], so the board states can be better encoded. Lastly, given that traditional Othello engines reach high-level performance using smart evaluation functions, search algorithms and opening book, and the first other agent can achieve better performance than our agent. We can simply design an agent using classic search algorithm, opening book and evaluation function with more heuristics involved.

References

[1] Andersson, G.: Zebra short facts, http://www.radagast.se/othello/zebra.html/

[2] Anita_harbour: Introductno to tkinter. [EB/OL], https://www.cnblogs.com/anita-harbour/p/9315340.html Accessed May 28, 2022

[3] Buro, M.: Logistello: A strong learning othello program. In: 19th Annual Conference Gesellschaft für Klassifikation eV. vol. 2. Citeseer (1995)

[4] Buro, M.: How machines have learned to play othello. In: IEEE Intelligent Systems Journal (1999)

[5] Buro, M.: The evolution of strong othello programs. In: IWEC (2002)

[6] Buro, M.: The evolution of strong othello programs. Entertainment Computing pp. 81–88 (2003)

[7] Campbell, M., Hoane Jr, A.J., Hsu, F.h.: Deep blue. Artificial intelligence **134**(1-2), 57–83 (2002)

[8] Curro, C.: Othello ai, https://github.com/ccurro/othelloAI

[9] Daví, H.H.: Predicting expert moves in the game of othello using fully convolutional neural networks (2017)

[10] Elnaggar, A., Gadallah, M., Mostafa, M., Eldeeb, H.: A comparative study of game tree searching methods. International Journal of Advanced Computer Science and Applications **5**, 68–77 (06 2014). https://doi.org/10.14569/IJACSA.2014.050510

[11] Festa, J., Davino, S.: ”iago vs othello”: An artificial intelligence agent playing reversi (2013)

[12] Hlynur Daví, H.: Predicting expert moves in the game of othello using fully convolutional neural networks (2017)

[13] Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: European conference on machine learning. pp. 282–293. Springer (2006)

[14] Lundh, F.: An introduction to tkinter. URL: www. pythonware. com/library/tkinter/introduction/index. htm (1999)

[15] Marcin: How to win at othello., https://bonaludo.com/2017/01/04/how-to-win-at-othello-part-1-strategy-basics-stable-discs-and-mobility/

[16] Moriarty, D., Miikkulainen, R.: Discovering complex othello strategies through evolutionary neural networks. Connection Science (2000). https://doi.org/10.1080/095400995500039228

[17] Norelli, A., Panconesi, A.: OLIVAW: mastering othello with neither humans nor a penny. CoRR (2021), https://arxiv.org/abs/2103.17228

[18] Plaat., A.: Learning to Play: Reinforcement Learning and Games. Springer (2020)

[19] QiuHaoran: Othello. https://github.com/James-QiuHaoran/Othello

[20] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484–489 (2016)

[21] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of go without human knowledge. nature **550**(7676), 354–359 (2017)

[22] Welty, C.: Chris's othello stuff, https://web.archive.org/web/20070109170329/http://othellogateway.com/ntest/

[23] Wikipedia: Computer othello, https://en.wikipedia.org/wiki/Computer_Othello

[24] Wikipedia: Elo rating system, https://en.wikipedia.org/wiki/Elo_rating_system