AlphaGo Zero, Beyond Go

Haozhe SUN
M2 MVA
Télécom ParisTech
Paris, FR 75013
sunhaozhe275940200@gmail.com

Tong ZHAO
M2 MVA
Ecole des Ponts ParisTech
Champs-sur-Marne, FR 77420
tong.zhao@eleves.enpc.fr

Abstract

AlphaGo Zero is the improved version of AlphaGo, which is developped by DeepMind in 2017. Being the most powerful model at that moment, it achieved a great success. Unlike all previous models, AlphaGo Zero is impressive and elegant since it is learnt from scratch without any human knowledges. In this project, we investigated the algorithm, and then applied it in English Draughts, a popular two-player board game.

1 Introduction

The Game AI is always a hot research domain for decades. Researchers believe that it is a good measurement for the effectiveness of artificial intelligence by applying it in games. Alan Turing, the most famous computer scientist, (re)invented the Minimax algorithm and developed Turochamp [1] for chess in 1948, which is known to be the earliest computer game player. Deep Blue [2] attracted wide attention in computer science community after its first victory against Garry Kasparov, a world champion on 10 February 1996.

Go and chess are both very popular deterministic two-player board games, but Go is much more difficult because of the huge search space in each game state. In October 2015, DeepMind developped the first computer Go program - AlphaGo [3] - to beat a human professional Go player without handicaps on a full-sized 19 $\times$19 board, which shocked the world. It is a supervised learning system combining deep neutral network and tree search algorithm to learn how to make decisions for game state from human experts. However, on one hand, expert data sets are often expensive, unreliable or simply unavailable; on the other hand, labeled data constrain the learning ability of the model itself.

AlphaGo Zero [4] is a historical breakthrough in Game AI, not only because that it beated all the previous version of AlphaGo and all human experts, but also because that it is learnt by self-play without any human knowledge. The whole framework is composed of a deep neural network f_θ, which predicts at the same time the value and the policy for each game state, and a Monte Carlo Search Tree (MCTS) guided by f_θ, which improves the value and policy in turn. It can be easily generalized to many games.

In this project, we investigate at first the algorithm flow of AlphaGo Zero. A readily comprehensible interpretation is provided. We then apply the algorithm in English Draughts [5]. The implementation in Python and PyTorch can be found at `https://github.com/Tong-ZHAO/AlphaDraughts-Zero`.

The report is organized as following:

- In section 2 and 3, we explained the deep neural network and MCTS used in AlphaGo Zero, respectively.
- In section 4, we discussed how to train jointly the deep net and the game tree, the loss function design and the evaluate method.

- In section 5, we adapted the algorithm to English Draughts and developed our AlphaDraughts Zero.
- In the last section, we focus on some interesting open questions and our understandings.

2 Deep Neural Network

AlphaGo Zero makes use of a deep neural network, which aims to, at each time step, predict the optimal action to take given the current game state. For the implementation of our AlphaDraughts Zero, we follow the same architecture as the one of AlphaGo Zero except the input and output format.

2.1 Input and output format

2.1.1 AlphaGo Zero

The input of AlphaGo Zero's neural network is a $19 \times 19 \times 17$ image stack comprising 17 binary feature maps, from which 8 feature maps X_t consist of binary values indicating the presence of the current player's stones in last 8 rounds, 8 feature maps Y_t represent the corresponding features for the opponent's stones in last 8 rounds, 1 feature map has a constant value of either 1 if black is to play or 0 if white is to play. These maps are concatenated together to give input features. History features are necessary for the game of Go to forbid repetitions, the color feature plane is necessary because *komi* is not observable.

The output of the neural network is passed into two separate heads for computing the policy and the value respectively. The value head outputs a scalar value which represents the estimated state value, and the policy head outputs a vector of dimension $19 \times 19 + 1$ which represents the logit probabilities for all intersections and the pass move.

2.1.2 AlphaDraughts Zero

In AlphaDraughts Zero, the input of neural network is a $8 \times 8 \times 3$ image stack comprising 3 feature maps. The first map encodes the current game state and takes value in $\{-1, 0, 1\}$. The second feature map represents the mask, where 1 represents legal move positions and 0 represents illegal move positions. The third feature map indicates the current player, it has a constant value of either 1 if white is to play or -1 if black is to play.

The value head of AlphaDraughts Zero is exactly the same as AlphaGo Zero whereas its policy head takes a different form. The output of policy head is a vector of dimension $8 \times 8 \times 4$, where 8×8 encodes the piece to choose, 4 encodes the 4 possible move directions of the chosen piece.

2.2 Common neural network structure

The general structure [4] of the deep neural network is a residual block [6] towers. It consists of a single convolutional block followed by 19 residual blocks [6].

The convolutional block consists of the following modules:

- A convolution of 256 filters of kernel size 3×3 with stride 1
- Batch normalization [7]
- A rectifier non-linearity [8]

Each residual block consists of the following modules:

- A convolution of 256 filters of kernel size 3×3 with stride 1
- Batch normalisation
- A rectifier non-linearity
- A convolution of 256 filters of kernel size 3×3 with stride 1

- Batch normalization
- A skip connection which adds the input to the block
- A rectifier non-linearity

The value head consists of the following modules:

- A convolution of 1 filter of kernel size 1×1 with stride 1
- Batch normalization
- A rectifier non-linearity
- A fully connected linear layer to a hidden layer of size 256
- A rectifier non-linearity
- A fully connected linear layer to a scalar
- A *tanh* non-linearity outputting a scalar in the range $[-1, 1]$

The policy head consists of the following modules:

- A convolution of 2 filters of kernel size 1×1 with stride 1
- Batch normalisation
- A rectifier non-linearity
- A fully connected linear layer that outputs a vector of the corresponding dimension

The hyperparameters of the structure such as the number of residual blocks or the number of filters in each residual blocks can be changed to increase or reduce the size of the neural network. In our implementation of AlphaDraughts Zero, the parameter setting is flexible, which allows experiments with different neural network architectures.

3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a well-known heuristic search algorithm to build an asymmetric tree model by exploring legal moves stochastically. It draws a lot of attention during these years because of its success application in computer Go program and its potential application in many different problems. Beyond the game itself, MCTS theory can be used in any field in which we predict the consequence by the (state, action) pairs. In this section, we introduce MCTS from the following aspects: data structure, basic algorithm and exploration-exploitation strategy.

3.1 Data Structure

There are two kinds of nodes representing graph node and graph edge respectively, namely the State node and the Action node.

3.1.1 State Node

As the name suggests, State node contains all information about the current game state. Its parent is an Action node, which represents the corresponding move from the last game state to itself. Its children are Action nodes representing all legal moves. A boolean flag indicates if the game terminates at the node.

3.1.2 Action Node

Both the parent and the children of an Action node are State nodes. We can consider the pair of (parent node, node, child) as (previous game state, move, current game state). It keeps track of four statistical variables of the edge:

- N: the number of times the action has been taken

- W: the total value of the next state
- Q: the mean value of the next state
- P: the prior probability of selecting this action

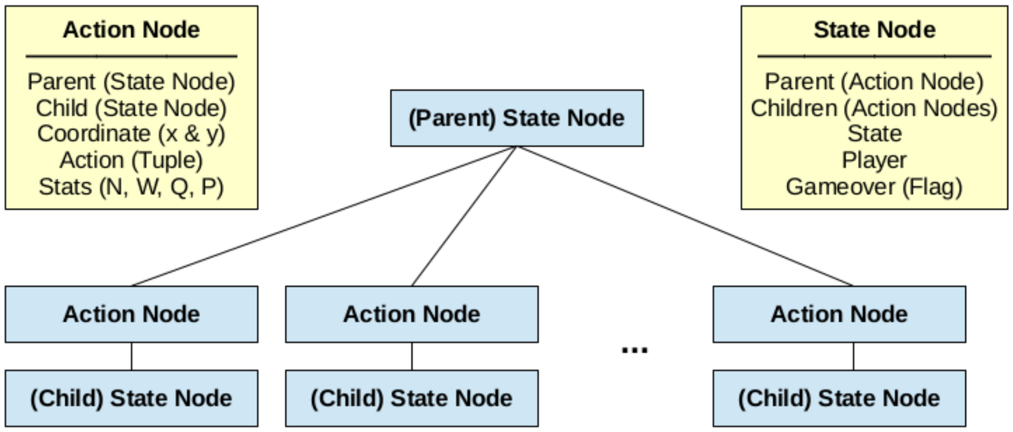


Figure 1: MCTS nodes

3.2 Basic Algorithm

There are four basic operations in MCTS which allow the tree grow up, namely Selection, Expansion, Evaluation and Backpropagation.

3.2.1 Selection

Everytime when we reach a non-leaf node, one of its children need to be selected according to a specific strategy. The child is selected deterministically for a competitive play, meaning that the edge with the greatest N will be chosen. The child is selected stochastically for an exploratory play, meaning that we choose the action from the distribution $\pi \sim N^{\frac{1}{\tau}}$.

3.2.2 Expansion

Everytime when we reach a leaf node where the game is not yet finished, an expansion is applied to initialize the children of the current leaf node. The deep neural network takes the game state as input and gives the matrix of the estimated prior probability p, including some illegal moves. The tree needs to select all legal moves and initialize the action nodes with:

$$N = W = Q = 0$$
$$P = p_{move}$$

3.2.3 Evaluation

Evaluation is a complete process going from a root to a leaf node. In the simulation step, we choose the child with the maximum $Q + U$, where U is a function of P and N that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high, which we will give more details in next sub-section. An expansion is then performed when a leaf node is reached.

3.2.4 Backpropagation

Backpropagation is performed after evaluation steps. Each edge that was traversed to get to the leaf node is updated by:

$$N \leftarrow N + 1$$
$$W \leftarrow W + v$$
$$Q \leftarrow W/N$$

where v is the value estimated by value head.

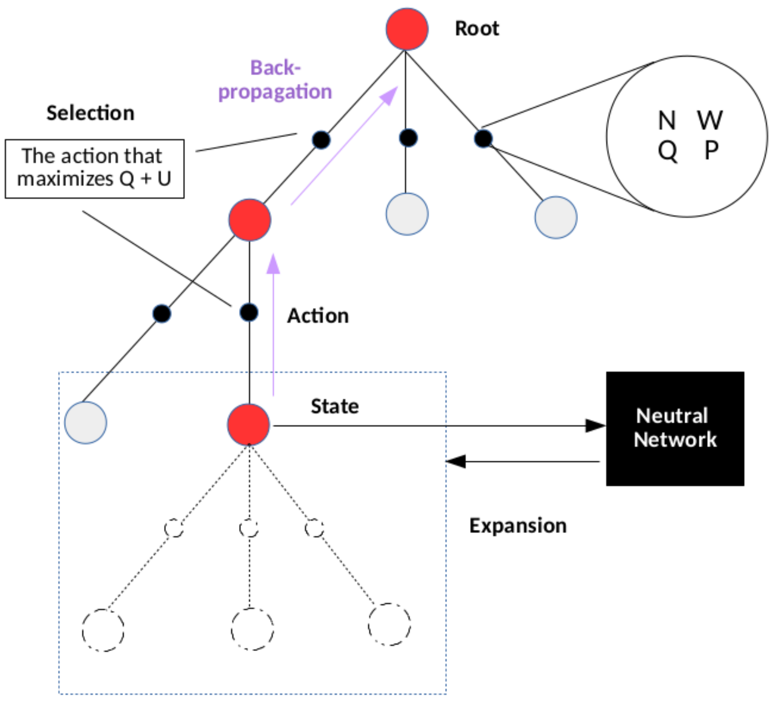


Figure 2: MCTS

3.3 Exploration-Exploitation Strategy

The exploration-exploitation trade-off is a fundamental dilemma whenever we learn about an environment by trying things out. Exploration means to make the best decision given current game state, while exploitation means to try more choices to gather more information. Similarly to multiarmed bandit problem, we must make a choice to maximize the income in each iteration.

The Upper Confidence Bounds for Trees (UCT) algorithm is an optimal choice for standard MCTS. It is derived from the Upper Confidence Bounds - 1 (UCB1) algorithm. In AlphaGo Zero, a variation called Polynomial Upper Confidence Trees (PUCT) is used. The score for a child (s, a) is calculated as:

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

Therefore, the less we have tried this action, the greater U will be. This encourages exploration. By increasing c_{puct}, we put more weight toward this exploration term. By decreasing it, we more strongly value exploiting the expected result Q.

4 Training Pipeline

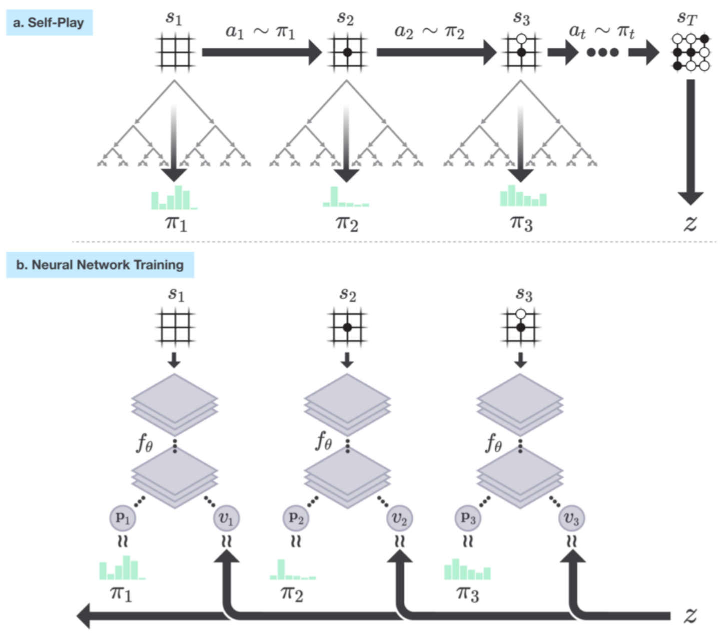


Figure 3: Self-play reinforcement learning in *AlphaGo Zero*. Reprinted from [4].

To summarize the training process in Figure 3, in each iteration, AlphaGo Zero plays a game $s_1, s_2, ..., s_T$ against itself. In each position s_t, a Monte-Carlo tree search (MCTS) is executed using the latest neural network. Moves are selected according to the search probabilities π_t computed by the MCTS, i.e. $a_t \sim \pi_t$, where a_t denotes the action to take. The terminal position s_T is used to compute the game winner z_t of each time step t of the latest self play game. z_t takes value in $\{-1, 1\}$, it is computed based on the terminal position s_T and the player at each time step t, it represents the outcome of the game from the perspective of the player at time step t. When one game is finished, the data of each game positions s_t is stored together with the corresponding search probability π_t, the outcome of game z_t, we then get a tuple (s_t, π_t, z_t) for each time step t of the latest self play game. All these data will be thus part of the dataset with which we train the neural network. The data will be sampled uniformly from the dataset, i.e. from all time steps of recent self play games. The optimisation of neural network consists of minimizing the loss Equation 1, where c is a parameter controlling the level of L2 weight regularisation to prevent overfitting, θ denotes the parameter of the neural network, v denotes the output of the value head of the neural network, $\mathbf{p}$ denotes the output of the policy head of the neural network. The neural network is optimised to minimise the mean squared error between the predicted value v and the self play winner z, and to maximise the similarity between the neural network's predicted move probability $\mathbf{p}$ and the MCTS's search probability π by minimising the cross-entropy loss between them.

$$\text{loss} = (z - v)^2 - \pi^\intercal \log \mathbf{p} + c||\theta||^2 \tag{1}$$

AlphaGo Zero's self-play training pipeline [4] consists of three main components, all executed asynchronously in parallel. These three components are **optimisation**, **evaluator**, **self-play**.

4.1 Optimisation

Each neural network is optimised with 64 GPU workers and 19 CPU parameter servers. The batch-size is 32 per worker, for a total mini-batch size of 2048. Each mini-batch of data is sampled uniformly at random from all positions from the most recent 500000 games of self-play. Neural network parameters are optimised by stochastic gradient descent with momentum and learning rate annealing, using the loss Equation 1 with L2 regularisation term c is set to 10^{-4}. The optimisation process produces a new checkpoint every 1000 training steps.

4.2 evaluator

As the data is generated by self-play, we need to ensure the quality of the data by making sure to use the best player obtained so far to generate it. Every time a new checkpoint of neural network is generated, it will be evaluated against the current best neural network. Each evaluation consists of 400 games using an MCTS with 1600 simulations. If the new checkpoint wins by a margin greater than 55%, then it becomes the best neural network and will be used to generate data.

4.3 self-play

The current best neural network is used to generate data. In each iteration, this network plays 25000 games of self-play, using 1600 simulations of MCTS. The clearly lost games could be resigned to save computation time, for example, AlphaGo Zero could resign if its root value and best child value are lower than a certain threshold.

4.4 Implementation

In the implementation of our AlphaDraughts Zero, the three main components **optimisation**, **evaluator**, **self-play** are implemented in a sequential manner. The pseudo code is Algorithm 1.

Algorithm 1 Sequential Training Pipeline - AlphaDraughts Zero

```
Random initialization of neural network $f_\theta$
for iteration $i$ in a certain number of iterations do
    build a new MCTS $\alpha(f_\theta)$ with a certain number of simulations
    for a certain number do
        data buffer ← empty list
        current state node ← the root of current best player MCTS $\alpha(f_\theta)$
        while current game is not finished do
            $\pi$ ← search policy based on statistics stored on the current state node
            get action node according to $\pi$
            add new data item $(s, \pi, z)$ to data buffer            ▷ $s$ denotes the game position
            current state node ← the only child of action node        ▷ move to next state node
        update $z$ value of each data item in data buffer based on the outcome of self play
        save data buffer to dataset on disk              ▷ dataset is a fixed-sized FIFO container
    train $f_\theta$ for a certain number of epochs using the current dataset
    if $i$ modulo save frequency $= 0$ then
        save $f_\theta$ as a new checkpoint
        evaluator updates best player if the new checkpoint is better than the current best player
```

However, this sequential training pipeline could easily be adapted to a parallel training pipeline. It is sufficient to distribute different tasks to different processes. The communication between processes could be managed by passing new *checkpoint*, latest neural network parameters θ, current *dataset* size, etc. to each other when needed. Moreover, asynchronous IO techniques could be considered when saving self play data to save time.

5 AlphaDraughts Zero

In this section we present our AlphaDraughts Zero, where we apply the algorithm to English Draughts, a popular two-player board game. It is implemented in Python and PyTorch. The source code can be found at `https://github.com/Tong-ZHAO/AlphaDraughts-Zero`. In the following, we first introduce the game rule, and then the adapted algorithm.

5.1 Game Rule

Board The game is on a 8×8 chequered board. The playable surface consists of the 32 dark squares only.

Piece Each player has 12 pieces in total. The start positions are shown in figure 4. The player with the darker-colored pieces moves at first. There are two kinds of pieces: Men and Kings. At the beginning, all the pieces are Men.

Kings If a Man moves into the last row on the opponent's side, i.e. the first row of the board for white, and the 8th row of the board for black, it becomes a King.

Move There are two ways to move in English Draughts, i.e. single move and eat jump. A single move means to slide a piece diagonally to an adjacent and unoccupied dark square. A jump move means that we jump over an opponent's piece which is on an adjacent dark square. In this case, the opponent lose its piece. Multiple jumps are possible and mandatory. If a player has the option to jump after one jump, he must take it, even if doing so results in disadvantage for the jumping player. However, if there is more than 1 way to jump, the player can choose among them. Men can only move forward and Kings can move both forward and backward.

End A player wins by capturing all of the opponent's pieces or by leaving the opponent with no legal move.

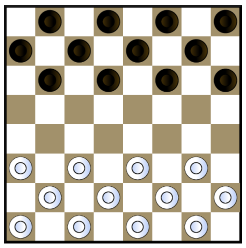
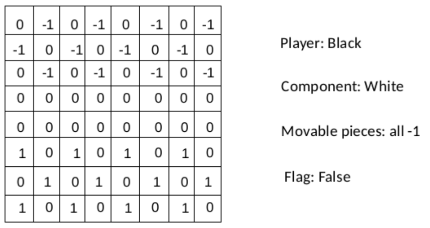


Figure 4: A typical game board



0	-1	0	-1	0	-1	0	-1
-1	0	-1	0	-1	0	-1	0
0	-1	0	-1	0	-1	0	-1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

Player: Black

Component: White

Movable pieces: all -1

Flag: False

Figure 5: Numerical representation

5.2 Game State

Game state records all necessary information to define the current hand. Besides standard information like the board map, the player and the opponent, we add a list of movable pieces and a flag indicating if it is mandatory to do a jump. These two additional variables are used to handle multiple jumps. An example is shown in figure 5.

5.3 Move

In order to deal with the case of multiply jumps, the move is defined by five elements: the coordinate of the piece, the direction, a flag indicating if the move is a jump and a flag indicating if we change the player in next turn, i.e. (x, y, direction, flag-jump, flag-change). This setting helps us decompose a multiply jump into several steps, which facilitates the calculation of the neural network. So now we can formulate all kinds of moves:

Simple Move is represented by (x, y, direction, False, True).

Jump is represented by (x, y, direction, True, False). The only movable piece in next state is (x, y).

5.4 Neural Network

While the network structure, ResNet [6], is kept, we adjusted the input and the output of the network.

Input Data is composed of three 8×8 matrices, representing the current board map, the movable pieces and the current player, respectively.

Policy Head gives a vector of dimension $8 \times 8 \times 4$, where $P_{i,j}$ indicates the predicted probability for moving the piece at the position (i, j) to 4 directions: Northwest, Northeast, Southwest and Southeast.

Value Head gives a single value in the range $(-1, 1)$, which indicates the estimated success rate in the current game state.

5.5 MCTS

MCTS plays an important role in the simulation. It ensures that all nodes in the tree are legal and controls the next player and its movable pieces. When it initializes a leaf node, all illegal moves are ignored. An exception happens when we finish a multiple jump, during which all moves are illegal while the game is not finished. In this case, we inserve the player and the opponent in the current state node. Its children will then be initialized when the node is visited again.

6 Experiments

In this section, we briefly show some plots of our experiments. Figure 6 shows the training curve of one of our experiments, the first row represents the average loss of each training epoch of the neural network, the second row represents the length of self-play games in each iteration. Figure 7 shows the graphical user interface of the English draughts game. We implemented it from scratch and it is used to demonstrate the human-machine competition, which is used to assess the capability of the model.

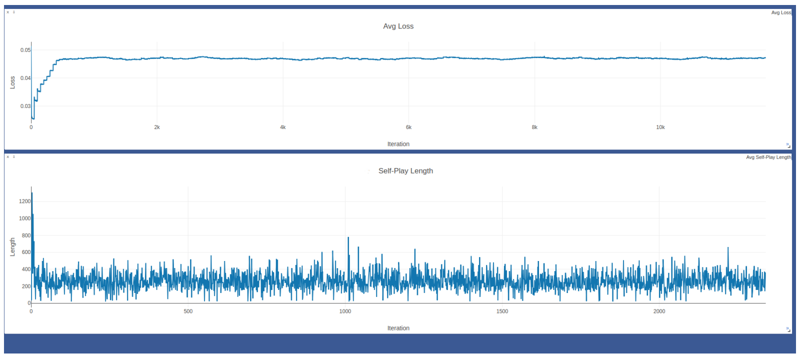


Figure 6: Training curves. Average loss, self-play game length.

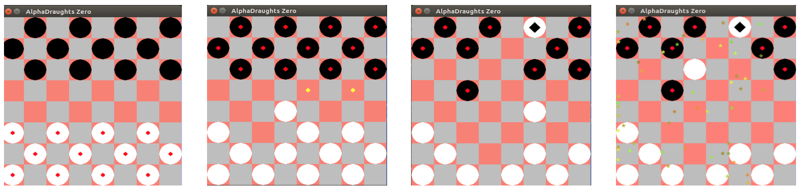


Figure 7: Graphical user interface. The 4 figures show selection of piece, selection of move given a piece, crowned piece / king and the celebration of crown

Human-machine competition is a qualitative assessment. The quantitative assessment can be achieved by Elo rating system [9]. We provide the basic functions of Elo rating system in our source code.

7 Discussion

In this section, we plan to talk about some open questions. One famous open question is the performance of AlphaGo Zero style techniques in face of games with incomplete information, which reflects their versatility. In such situations, the decision making process is no longer only dependent on the current observable states, it also demands the capacity of dynamic reasoning. An ideal artificial intelligence should be able to guess the behavior of the opponent, to assess the opponent's situation, or even to deduce what the opponent knows about itself to make the correct decision.

Essentially, AlphaGo Zero is a mechanism of finding optimal decision based on observed information. For example, the neural network of AlphaGo Zero takes the observed current state as input and returns a estimated decision, this is achieved by searching in a high dimensional space with some techniques to reduce the time complexity and to improve the search efficiency. In games with complete information, this decision will be consistent as long as the observed information is identical. However, in games with incomplete information, the optimal decision may not be the same given the current observed information, as it may depends on chances, etc. This could be considered as a source of noise. We are not sure if this mechanism is robust enough to deal with such kind of situations, we are not sure if the training could converge by learning to take into account such noises or it just results in oscillations. One approach to tackle this problem could be to make use of historical information. When dealing with games with incomplete information, humen often deduce hidden current situation by analyzing historical information. This could be done by using RNNs, LSTMs [10] or Transformers [11].

References

[1] Wikipedia contributors. "turochamp" wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Turochamp&oldid=877335597`. accessed January 14, 2019.

[2] Wikipedia contributors. "deep blue (chess computer)" wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Deep_Blue_(chess_computer)&oldid=877916985`. accessed January 14, 2019.

[3] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[5] Wikipedia contributors. "english draughts" wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=English_draughts&oldid=869357578`. accessed January 14, 2019.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, page arXiv:1512.03385, December 2015.

[7] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, page arXiv:1502.03167, February 2015.

[8] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[9] Wikipedia contributors. "elo rating system" wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Elo_rating_system`. accessed January 28, 2019.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv e-prints*, page arXiv:1706.03762, June 2017.