Applying General Reinforcement Algorithm on Games with Small Search Spaces

James Parker[1]

Research School of Computer Science, Australian National Unievrsity
u5744091@anu.edu.au

Abstract. AlphaZero[10] algorithm is the current state-of-the-art method to train an agent in deterministic 2 player zero-sum games. We have tested whether the slightly simplified version of the algorithm performs well in two of the simpler game domains: TicTacToe and Kalah. Our agent learned to perform well in TicTacToe under much shorter training time and computation. After 1000 episodes of training, the agent was playing almost perfect games. In Kalah, however, the agent using simpler model struggled to improve its performance without some tradeoffs. This suggests that deep neural network with a convolutional neural network is a necessarily required for training an agent with game domains with some complexity.

Keywords: TicTacToe · Kalah · AlphaZero · MCTS · PUCT · Reinforcement Learning

1 Introduction

The recent artificial intelligence techniques with neural networks have been producing surprising achievements in various fields of research.[3, 6, 10] The one of the most outstanding work was produced by Google DeepMind team creating an AI that a game of Go in superhuman level[8]. After beating the world-class player Lee Sedoul in 2016, an improved version of the program has beaten the world champions on an online Go tournament, winning all 60 games played against professinals[10]. In the end 2017, DeepMind has produced its last version of AlphaGo called AlphaGoZero[10] which performs better than all of its previous versions with less computation power and less training time. The most fascinating fact about the algorithm is that it does not use a dataset of previously played professional games; it learns all of its techniques only through repeated self-plays. In a matter of 3 hours of training, the program was performing professional level. After 3 days of training, the agent overperformed all of its previous versions of AlphaGo.

It was also shown that the algorithm works on other zero-sum two-player games such as chess and shogi[9], which is a Japanese variation of chess. Unlike Go, chess and shogi lack in structural symmetry with smaller search spaces. Hence, they were thought to be less suited for the algorithm. DeepMind has tested the general version of the previous algorithm and called it AlphaZero.

Despite the concerns, the agents trained with AlphaZero surpassed the best-performing programs in each of the game domains.

Although chess and shogi have smaller search space than Go, they are still considered to be highly complex, and neither of the games is solved. Game-tree complexity is defined as the total number of possible games that can be played. Shannon estimated that the game-tree complexity of chess is about 10^{120}[7]. On the other hand, the game-tree complexity of shogi is estimated to be 10^{226}[4]. AlphaZero is shown to be an effective algorithm on these larger games, but we could not find any research on whether the algorithm works well on games with smaller game-tree complexities where a standard minimax algorithm with alpha-beta pruning can perform well. In this paper, we will we consider two such games: TicTacToe and Kalah.

1.1 TicTacToe

TicTacToe is a famous two player game where each player draws their symbol on 3 by 3 grid in alternating turns. The symbols are usually played with Os and Xs. The player who achieved to place their symbols three in a row wins the game. Without considering the geometric symmetry of the games, the game-tree complexity is roughly 10^5. It is shown that this game ends with a draw if both players play optimally.

1.2 Kalah

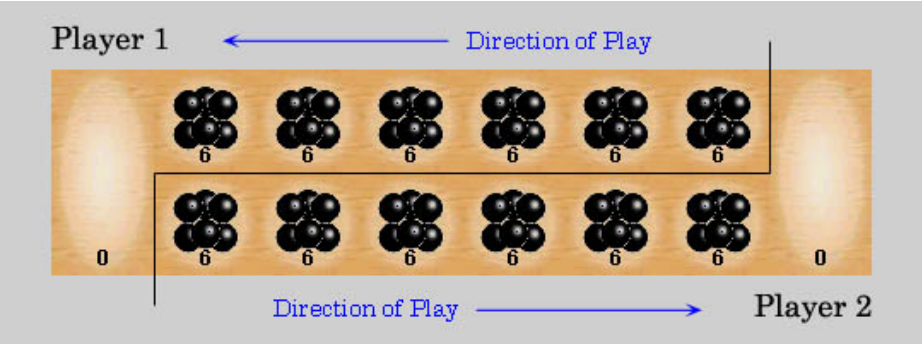

Fig. 1. A figure of a normal Kalaha(6, 6) board. The diagram shows the starting state of the game. Two players sit at the opposite ends. There are six houses on each side of the board with six seeds in each them.

Kalah is another two-player zero-sum game with several variants. The board of kalah consists of houses and end zones for placing seeds. Each player sits on the opposite sides of the boards. There are several fixed numbers of houses on

each side of players. End zones are located on the opposite ends of the board. The goal of the game is to move as many seeds to your own end zone. We will be considering a standard Kalah(6,6), which means there are 6 houses on each side with 6 seeds in each of the houses in the initial state. The board is shown in the Fig 1. The rules for moving the seeds are the following.

1. A player can pick one house on their side containing seeds in their own turn.

2. All seeds are taken from the house and distributed in a counter-clockwise direction around the board. The player places one seed in each housew and end zones.

3. If the last seed is placed in the empty house on the player's side, it can capture seeds on the opposite house. All of the seeds in the last house and the opposite house are moved to the player's end zone. The capture can only occur if the opposite house contains a seed.

4. A player can play another turn if the last seed is placed in the player's end zone.

The game ends when one player does not have any seeds in their own houses. A player wins if the number of seeds on his/her side is more than that of the other player. The game-tree complexity of Kalah(6,6) is about 10^{33}[5], and the game was solved in 2011[1]. The optimal moves always allow the first player to win the game.

2 Methods

2.1 AlphaZero

The core of AlphaZero algorithm is a deep neural network aided monte-carlo tree search (MCTS). The deep neural network f_θ with parameters θ is used to evaluate the current state. The network takes in a representation of the game state s and outputs move probabilities and a value, $(\boldsymbol{p}, v) = f_\theta(s)$. The vector of move probabilities represent the probabilities taking the action a at the state s, $p_a = P(a|s)$. The scalar value v represents the probability of the current player winning at the current position. The v values of $1, 0.5$, and 0 represents a winning position, drawing position, and losing position respectively. Combining the probability and the value in a single network maintains the consistency between the two outputs. The original AlphaZero utilized 20 layered deep neural networks with convolutional layers and pooling layers, but in our paper, we used a much simpler model (see Architecture).

2.2 MCTS

The deep neural network is trained from games of self-play. In each state s, MCTS is executed guided by f_θ. MCTS outputs the probability vector $\boldsymbol{\pi}_s$ of playing each move at state s. At the final state s_f, we compute the winner of the game z. Then, for each of the states, we compute the winner value in respect to

the players' turns $r_s = \pm z$. We scale the values r_s to match the range of position value to compute r'_s. Finally, we store the values $(s, \boldsymbol{\pi}_s, r_s)$ in our history. This history of position information is overwritten from the oldest values. We use the history size of $50,000$. At the end of each gameplay, uniformly sampled state information from the history is fed into the network, and trained in mini-batch fashion. The loss l of the network is computed with the following equation[9].

$$(\boldsymbol{p}, v) = f_\theta(s) \text{ and } l = (r'_s - v)^2 - \boldsymbol{\pi}_s^\top \log(\boldsymbol{p}) + c||\theta||^2 \tag{1}$$

This is a combined loss of mean squared error of r'_s and v and cross entropy error between $\boldsymbol{\pi}_s$ and $\boldsymbol{p}$ with L_2 regularization.

MCTS performs series of game simulation and stores the playout statistics in the structure of a tree. Each edge (s, a) in a tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, a total action value $W(s, a)$, and an average action value $Q(s, a)$. These values are initialized to $P(s, a) = p_a$, $N(s, a) = 0$, $W(s, a) = 0$, and $Q(s, a) = 0$. Each simulation starts at the root of the tree and iteratively selects an action $\hat{a}$ the action with Polynomial Upper Confidence Tree (PUCT) algorithm using the following criteria[2].

$$\hat{a} = \arg\max_a (Q(s, a) + U(s, a)) \tag{2}$$

where

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}. \tag{3}$$

The PUCT algorithm allows some diversities in the selected action. At the same time, well-performing actions are favored more as the search is repeated. When the game is completed with an output of v, we update the edge statistics with the following algorithm.

1. Update edge statistics. $N(s, a) = N(s, a) + 1$.
2. Update action value. $W(s, a) = W(s, a) + v$
3. Update mean action value. $Q(s, a) = \frac{W(s,a)}{N(s,a)}$

The number of simulations for MCTS for each domain is explained in the Experiment section.

Once the simulation is completed, we evaluate $\boldsymbol{\pi}$ with

$$\pi(a, s_0) = \frac{N(s_0, a)^{1/\tau}}{\sum_b N(s_0, b)^{1/\tau}} \tag{4}$$

where s_0 is the current state and τ is a temprature parameter[9]. Higher the temperature, the probabilities are more distributed. As $\tau \to 0$, the algorithm becomes greedy. In the training phase, we set $\tau = 1$ in half of the games. In all the other settings, we set $\tau \to 0$.

2.3 Architecture

We have conducted an experiment with four-layer feed-forward neural network. The original paper used convolutional layers before feed-forward layers, but we

have kept the architecture simple as it turns out to be still effective in small games with small state space such as TicTacToe or Kalah.

2.4 Preprocessing

The original paper did not mention about the preprocessing as all of its inputs were binary values. Similarly, we did not perform any preprocessing for TicTacToe domain as all the inputs are binary. With Kalah, however, input can be any non-negative integer with an average value of 5.14. The input was normalized so that most values fall in the range of 0 to 1.

3 Results and Discussions

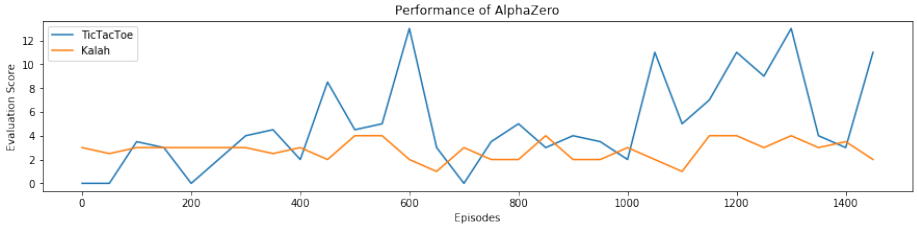

Fig. 2. The perfomance of the agent in each of the domain. The score is evaluated with $w + 0.5d$ where w is a number of wins and d is the number of draws against minimax agents

We trained our agent for 1500 episodes in each of the game domains. We have tested agent every 50-time steps against minimax agent with varying depth from one to seven. When the agent reaches the leaf node which is not a terminal state in the game, the state is evaluated with the heuristics. In TicTacToe, the agent used an uninformed heuristics which always returns 0. In Kalah, the agent used a heuristic value based on the number of seeds in its own end subtracted by the number of seeds in the opponent's end. The agent is tested twice against each agent with the switched sides. The score of an agent is evaluated with $w + 0.5d$ where w is a number of wins and d is the number of draws against the minimax agents. If the AlphaZero agent does not lose twice against the minimax agent of the same depth, it is tested with the agent of higher depth.

The result of the experiment is shown in the Fig 2. The agent has learned to perform well in TicTacToe domain. In 600 episodes, the agent has performed to beat most of the minimax agents winning almost perfectly. In fact, it was able to win against the minimax agent with depth 7 which requires playing optimally on all of its moves. We can also see that the performance of an agent is not consistent throughout the test. Between the episodes 650 and 1000, the agent

did not perform well as before, but it is able to regain its performance with more consistency later.

There was not much improvement in the performance of the agent in Kalah domain. It won an almost consistent number of games throughout the training phase. However, there were some improvements as well. After 500 episodes, the agent was able to gain some wins against minimax agent with depth 2 or 3. On the other hand, its performance against the depth 1 agent declined. This shows that there was some overfitting or overgeneralization of the model. The original paper was able to perform superhuman level after 700,000 episodes of training, and our agent was not able to perform anywhere close to that level.

4 Conclusion and Future Work

The agent has learned to perform well in TicTacToe domain with some consistency, but Kalah(6,6) domain was a challenging domain. The agent's performance in TicTacToe domain was almost perfect after 1000 episodes of training and proved that AlphaZero is an effective learner in a game with smaller search space even with a simpler model. Additionally, our agent learned in a smaller number of episodes than the agents in the paper. On the other hand, the agent struggled to perform well in Kalah(6,6) with some consistency. The reason can be the limitation of the current model. With a larger model, however, it can get undertrained with the current number of episodes.

The paper left several questions about the AlphaZero algorithm. How much of the performance of agent is affected by removing CNN before feed-forward neural network still remains unknown. The future work can test the performance of the agents under different types of model.

Additionally, the speed of the training procedure against the different model is still untested. Our result suggested some tradeoffs between the complexity of the model and the number of episodes required for training. The optimal balance between these two values is still unknown if the agent is required to perform over some fixed performance.

References

1. Solving (6,6)-kalaha. http://http://kalaha.krus.dk/, accessed: 2018-05-30
2. Auger, D., Couetoux, A., Teytaud, O.: Continuous upper confidence trees with polynomial exploration–consistency. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. pp. 194–209. Springer (2013)
3. Chen, J., Zeng, Z., Jiang, P.: Global mittag-leffler stability and synchronization of memristor-based fractional-order neural networks. Neural Networks **51**, 1–8 (2014)
4. Dan, Y.: Possibility of human grid computing for artificial intelligence systems. In: Applications and the Internet, 2008. SAINT 2008. International Symposium on. pp. 452–454. IEEE (2008)
5. Irving, G., Donkers, H., Uiterwijk, J.: Solving kalah. Icga Journal **23**(3), 139–148 (2000)

6. Kim, Y.: Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882 (2014)
7. Shannon, C.E.: Programming a computer for playing chess. In: Computer chess compendium, pp. 2–13. Springer (1988)
8. 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)
9. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
10. 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 (2017)