Learning to Play Othello: A Self-Play Approach

Hao Fu
School of Data Science
14307130013

Chengming Xu
School of Data Science
14307130055

Abstract

Game playing, especially board games featuring perfect information, has long been a favorite area of artificial intelligence. Most early works attempt to train agents based on datasets comprising experts' move or play. However, a long-standing goal of artificial intelligence is an agent that learns superhuman proficiency without human knowledge. In this paper, we introduce an algorithm solely based on reinforcement learning of self-play. Starting tabula rasa, our agent learns to play Othello in Atari using neural networks for policy estimation and Monte Carlo Tree Search for policy improvement. For evaluation, we play the agent against fixed opponents of 3 levels and report win rates respectively.

1 Introduction

Learning from human is a mainstream methodology in game playing with artificial intelligence. Researchers implement algorithms which have computer programs learn from large scale data of expert playing, and become capable of superhuman performance on games like Chess and Go. For instance, Alpha Go [1] managed to defeat the top human player through extensive use of domain knowledge and training on the games played by top human players. Nevertheless, the authors pointed out in [2] the drawbacks including accessibility and reliability of expert data sets, and inherent technical bottlenecks of this training method. On the other hand, reinforcement learning based agents aimed to learn from their own experience and thus acquire knowledge beyond human expertise.

However, according to dual-process theory, human reasoning consists of two different kinds of thinking. *System 1* is a fast, unconscious and automatic model of thought, also known as *intuition* or *heuristic process*. *System 2*, an evolutionarily recent process unique to humans, is a slow, conscious, explicit and rule-based model of reasoning. When learning a board game, humans exploit both processes: strong intuitions allow for more effective analytic reasoning by rapidly selecting interesting lines of play for consideration. Thus, sole Reinforcement Learning agents are not capable of mastering board games.

Debuting with an overwhelming 100-0 victory against its predecessor, AlphaGo Zero [2] presented a new learning approach where the algorithm deployed no human knowledge and mastered superhuman performance entirely through self-play reinforcement learning. Furthermore, it used a single neural network, rather than separate policy and value networks. This inspiring result prompts us to apply the self-learning idea in a new scenario, the game of Othello. To this end, we introduce a self-play approach by which the agent achieves superhuman proficiency without guidance from human.

2 Related Work

Othello is a popular artificial intelligence subject due to its simple rules and well defined strategic concepts [3]. [4] described a evaluation function for Othello based on statistical analysis of a large set of example game positions and profits. In acquiring position evaluation functions for Othello, [5] proved temporal difference learning learned much faster than co-evolutionary learning, but the

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

latter could learn better playing strategies if well tuned. Multiple ways were proposed for agent training. [6] constructed CNN-based architectures of deep neural networks for move prediction and obtained state-of-the-art accuracy on the French Othello league game dataset WThor. [3] deployed co-evolutionary learning techniques which resulted in networks able to learn to play the game at an expert-master level and to discover advanced strategies.

Learning optimal playing strategies in games by self-play has been widely studied: Go [7], Chess [8], and Backgammon [9]. With respect to Othello in particular, [10] won against the top entries from the online Othello League without explicit use of human knowledge. [11] applied Monte Carlo methods to Othello. [12] compared learning from self-play to learning against a fixed opponent, and derived best strategy of learning differed per algorithm.

3 Methods

We deliver core techniques concerning reinforcement learning from a conceptual perspective. The algorithm uses no human knowledge except the rules of Othello, like validity of actions. We use a neural network to evaluate the value of a given board state and estimate the optimal policy. Monte Carlo Tree Search (MCTS) [13] is introduced as a policy improvement operator to guide the self-play. The outcomes thereof are regarded as rewards and used to train the neural network along with the improved policy. The neural network iteratively executes the self-play games, and uses the rewards to retrain the network. The following sections describe the different components of the system in detail.

3.1 Neural Network for Policy Estimation

The neural network f_θ parametrized by θ takes the board state s as input. The outputs are (1) the continuous value of the board state $v_\theta \in [-1, 1]$ from the perspective of the current player, and (2) a probability vector $\vec{p}$ over all possible actions. $\vec{p_\theta}$ is a stochastic policy used to guide the self-play.

Randomly initialized, the neural network takes training samples with the form $(s_t, \vec{\pi_t}, z_t)$ at the end of every iteration of self-play. $\vec{\pi_t}$ gives an improved estimate of the policy after performing an MCTS starting from s_t. $z_t \in [-1, 1]$ is the final outcome of the game from the perspective of the current player. The target loss function of the neural network is given as

$$L(\theta) = \sum_t (v_\theta(s_t) - z_t)^2 + \vec{\pi_t} \log(p_\theta(\vec{s_t})).$$

3.2 Monte Carlo Tree Search for Policy Improvement

The policy learned by the neural network is then improved by a Monte Carlo Tree Search. MCTS is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results [13]. MCTS explores the tree where nodes represent different board stages and a directed edge between two nodes $(i \to j)$ exists if state i can be transited to state j through a valid action. We maintain a Q value denoted by $Q(s, a)$ which is the expected reward for taking the action and $N(s, a)$ which is the number of times action a is taken from state s through different simulations. $P(s, \cdot) = \vec{p_\theta}(s)$ denotes the prior probability of taking a particular action from state s according to the policy given by the neural network. We derive the upper confidence bound $U(s, a)$ on the Q value of each edge as

$$U(s, a) = Q(s, a) + cP(S, a)\frac{\sqrt{\sum_b N(s, b)}}{1 + n(s, a)}$$

where c is a hyperparameter which controls the degree of exploration.

To find a policy from a given state s, an MCTS tree is constructed from s as the root. We calculate the action a to take which maximizes $U(s, a)$ at each step of the iteration. The simulation continues if the next state already exists in the tree. Otherwise, a new node initialized with $P(s, \cdot) = \vec{p_\theta}(s)$ and $v = v_\theta(s)$ is added to the tree, meanwhile initializing $Q(s, a)$ and $N(s, a)$ with 0 for all a. The reward v is then propagated back up the tree, and all the $Q(s, a)$ values seen are updated during the simulation before a new iteration starts from the root again. When reaching a terminal state, we propagate the actual reward found from the board and restart the MCTS procedure.

Algorithm 1 Monte Carlo Tree Search

procedure MCTS(s, θ)
  if s is terminal **then**
    return $result$
  end if
  if $s \notin Tree$ **then**
    $Tree \leftarrow Tree \bigcup s$
    $Q(s, \cdot) \leftarrow 0$
    $N(s, \cdot) \leftarrow 0$
    $P(s, \cdot) \leftarrow \vec{p_\theta}(s)$
    return $v_\theta(s)$
  else
    $a \leftarrow \arg\max_{a' \in A} U(s, a')$
    $s' \leftarrow$ getNextState(s, a)
    $v \leftarrow$ MCTS(s')
    $Q(s, a) \leftarrow \frac{N(s,a) \cdot Q(s,a) + v}{N(s,a)+1}$
    $N(s, a) \leftarrow N(s, a) + 1$
    return v
  end if
end procedure

After a few simulations, $N(s, a)$ values are able to well approximate the optimal stochastic process from each state. Thus the action taken is randomly sampled from a distribution π_s with the probability proportional to $N(s, a)^{\frac{1}{\tau}}$, where τ is a temperature parameter. A large τ indicates almost uniform distribution, while a τ towards 0 promises the best action. Therefore, τ is another hyperparameter controlling the degree of exploration. The training samples starting at s is (s, π_s, r) where $r \in \{1, -1\}$ is assigned at the end of the game which determines whether the current player win or not.

3.3 Policy Iteration through Self-play

Here we describe the complete training algorithm. The neural network initialized with random weights starts with a random policy. We run a number of episodes of self-play using MCTS in each step of iteration, which produce a set of training samples with the form $(s_t, \vec{\pi_t}, z_t)$. We take advantage of the invariance to board symmetries of Othello, and generate 7 extra reflections and rotations per original sample. Then the new samples are used to update the neural network and result in a new one. We play it against the old network for a number of games. While the AlphaGo Zero compares the old and immediate model to get the more powerful one, we do not take this step because simulate a game in the given Gym Env is time consuming, so we just always accept the newest model, and run another iteration to further expand the training set. Algorithm 2 and 3 below illustrate the aforementioned procedure in detail.

Algorithm 2 Policy Iteration through Self-play

procedure POLICYITERATION
  $\theta \leftarrow$ initNN()
  $samples \leftarrow []$
  for i in $[1, \cdots, iter]$ **do**
    for e in $[1, \cdots, epi]$ **do**
      $s \leftarrow$ executeEpisode(nn)
      $samples$.append(s)
    end for
    $\theta_{new} \leftarrow$ trainNN$(sample)$
    $\theta \leftarrow \theta_{new}$
  end for
end procedure

Algorithm 3 Execute Episode

procedure EXECUTEEPISODE
  $examples \leftarrow []$
  $s \leftarrow$ startState()
  while True **do**
    for i in $[1, \cdots, sim]$ **do**
      MCTS(s, θ)
    end for
    $examples$.add$((s, \pi_s, z))$
    $a^* \sim \pi_s$
    $s \leftarrow$ nextState(s, a^*)
    if isTerminal(s) **then**
      $examples \leftarrow$ assignRewards$(examples)$
      return $examples$
    end if
  end while
end procedure

4 Experiments

Settings We implement a neural network that takes a compressed board states as inputs, with 1 standing for current player, -1 for the opponent and 0 for the rest of the board. After the input layer follow 4 convolutional layers and 2 fully connected layers. The 2 output layers behind provide the continuous value v_θ and the policy vector $\vec{p_\theta}$ separately. Conditions for training include: (1) Adam optimizer with learning rate of 0.001 with a batch size of 64, (2) a dropout rate of 0.3, and (3) batch normalization after convolution layers.

With respect to parameters concerned in Algorithm 2 and 3, we run 100 episodes of self-play during MCTS, and update the policy-value network every 5 episodes, with a replay buffer of 10000 trajectories. The temperature parameter τ is set to 1 for the first 5 turns in an episode to encourage exploration at early times, and then set to 0. We follow the same setting during evaluation. The parameter c is set to 1. In the following text, we denote this agent as MCTS-PVNet, or AlphaZero-like agent.

Table 1: Win rates against opponents of 3 levels using MCTS-PVNet

Level	Games won/played on black	Games won/played on white
1	15/15	15/15
2	20/20	20/20
3	11/15	12/15

Table 2: Win rates against opponents of alpha-beta with depth from 4 to 6

Depth	Games won/played on black	Games won/played on white
4	14/20	18/20
5	13/20	15/20
6	15/20	16/20

Basic Results We briefly report experiment results in Table 1 (AlphaZero-like with 100 rollout agent performance). Note that when using AlphaZero-like agent, our testing procedure assigns 1.0 to temperature parameter during the first 5 step, so the test result is with randomness and may not be reproduced accurately. When training, we find that the loss term drops rapidly in the early stage, with only 5 times of update from 5 to 1, and then the speed slows down, like the performance reported on the original AlphaGo Zero paper. From Table 2(AlphaZero-like with 100 rollout agent performance against alpha-beta), we can learn that the trained model is not so robust. To understand specific strategies adopted by the agent, we examine details of games. As both of use are not perfect Othello

Table 3: Win rates against opponents of 3 levels Using pure MCTS

Level	Games won/played on black	Games won/played on white
1	15/15	15/15
2	20/20	20/20
3	11/15	14/15

Table 4: Win rates against opponents of 3 levels Using pure policy-value network

Level	Games won/played on black	Games won/played on white
1	12/15	12/15
2	20/20	20/20
3	1/15	1/15

player, we can just look for some reasonable patterns learned by the agent. We find that a remarkable lead in the early stage often leads to be reversed, because a great gap also means a potential thoroughly flip, so it maybe not a good policy to greedily move according to the immediate state of the board. When using both pure MCTS or MCTS with policy-value network against alpha-beta, the agent would allow some disadvantages, which somehow proves our guess.

Ablation Experiment To understand the role of MCTS in the whole model, we segment the model to use the trained policy-value network and the pure MCTS respectively as a direct access of our policy, and test against opponents over FTP. Table 3 (pure MCTS with 200 rollout agent performance) and Table 4 (policy-value network agent performance) show the results of this ablation experiment. The result shows that the performance of MCTS-PVNet is worse than the pure MCTS, possibly due to insufficient training. However, it is far faster than pure MCTS. Besides, with the result of Table 4, where the policy-value network performs well against random agent but poorly against alpha-beta agent, we can safely come to a conclusion that MCTS performs as a tool of looking ahead for the RL agent, making it more suitable for a board game which needs longer term prediction of board state, enhancing the policy.

5 Conclusions

We implement an agent that learns to play Othello solely by self-play, starting from random play, without any supervision or use of human data. Resulting in convincing win rates, it proves the feasibility of training by self-play to reach high level with no domain knowledge beyond basic rules. Meanwhile, the generic algorithm framework we adopt can be extensively deployed in other artificial intelligence scenarios.

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," vol. 550, pp. 354–359, 10 2017.

[3] V. Makris and D. Kalles, "Evolving multi-layer neural networks for othello," in *Proceedings of the 9th Hellenic Conference on Artificial Intelligence*, SETN '16, (New York, NY, USA), pp. 26:1–26:6, ACM, 2016.

[4] M. Buro, "An evaluation function for othello based on statistics," 1997.

[5] S. M. Lucas and T. P. Runarsson, "Temporal difference learning versus co-evolution for acquiring othello position evaluation," in *in IEEE Symposium on Computational Intelligence and Games*, 2006.

[6] P. Liskowski, W. Jaskowski, and K. Krawiec, "Learning to play othello with deep neural networks," *CoRR*, vol. abs/1711.06583, 2017.

[7] S. Gelly and D. Silver, "Achieving master level play in 9 x 9 computer go.," 01 2008.

[8] E. A. Heinz, "New self-play results in computer chess," 10 2000.

[9] M. Wiering, "Self-play and using an expert to learn to play backgammon with temporal difference learning," vol. 2, pp. 57–68, 01 2010.

[10] M. Szubert, W. Jaśkowski, and K. Krawiec, "On scalability, generalization, and hybridization of coevolutionary learning: A case study for othello," 04 2013.

[11] J. a. m. Nijssen, "Playing othello using monte carlo," 01 2018.

[12] M. Ree and M. Wiering, "Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play," 04 2013.

[13] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, pp. 1–43, March 2012.