CS303A Project1 ReversiV

Xu Botian 11810424
dept. Computer Science and Engineering
Southern Univ. of Science and Technology

***Abstract*—AI for board games has been a well researched topic ever since computer was invented. This project applied a highly-simplified but rather effective variant of the reinforcement learning framework, AlphaZero [1] by Deepmind, to our board game, Reversi. With a little prior knowledge besides the basic rules, the proposed method, ReversiV(V for value), was able to achieve an exciting level of playing with possibly minimum parameters and training time.**

I. Introduction

A. Reversi

Reversi, or Othello, is a strategy board game for two players, played on an 8×8 uncheckered board. There are sixty-four identical game pieces called disks (often spelled "discs"), which are light on one side and dark on the other. Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color. The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

Compared with other games, Reversi has the following properties: it is completely observable, episodic and deterministic, which make it a perfect task to study simple machine learning-based approaches. The key idea of reinforcement learning is to estimate from data the expected reward of taking an action from some state. A general and effective framework is desirable because it doesn't rely on hand-crafted evaluation function as traditional adversarial search strategies do, and can be easily applied to new tasks with little human knowledge.

B. Implementation

This project is written in Python3.8 with the learning process done by Pytroch. Trained parameters are retrieved afterwards to re-implemented the neural network with Numpy for submission.

II. Methodology

A. Related Work

Deepmind proposed in 2017, a generalized and efficient RL framework, Alpha(Go)Zero [1] [2], which can be applied to almost all chess games and convincingly beaten its precedent, AlphaGo with just days of training, compared to months needed by the latter.

The agent combines a neural network with Monte-Carlo Tree Search. The network takes in the state and outputs a policy $\vec{p}(s)$ which represents the probability of choosing action a at state s, and an evaluation $v(s) \in [-1, 1]$.

For the tree search, the following are maintained:

- $Q(s, a)$: the expected reward of taking action a at state s
- $N(s, a)$: the number of times we took action a from state s
- $P(s, \cdot) = \vec{p}(s)$: the initial estimate of taking an action from the state s according to the policy returned by the current neural network across simulations

from which we can calculate the upper confidence bound value $U(s, a)$ used for the selection step in MCTS:

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

where c_{puct} is a hyperparameter that controls the degree of exploration.

When a leaf node is encountered, instead of doing a rollout as in normal MCTS, the network is called to predict $\vec{p}(s)$ and $v(s)$. Then $v(s)$ is propagated up along the tree as return value.

For the learning process, at the end of each game of self-play, training examples of the form $(s_t, \vec{\pi}_t, z_t)$ are gathered, where $\vec{\pi}_t$ is an estimate of the policy from state s_t and $z_t \in \{-1, 1\}$ represents the estimate of the final outcome from the current player's perspective($+1$ for a win and -1 for the opposite). Then the neural network is trained to minimize the following loss:

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}_t \cdot \log\left(\vec{p}_\theta(s_t)\right)$$

So the intuition is to learn an evaluation of states and make decisions basing on the evaluation. Every time a new model is obtained, the agent is pit against its previous version to ensure improvement. Only models that can achieve a win rate over some threshold(55% in Deepmind' work)are accepted.

A difference between AlphaGo Zero and AlphaZero is that AlphaZero always uses the latest model to generate training examples, while AlphaGo Zero sticks to the best model so far if improvement is not being made.

Further introduction can be found in Deepmind's paper.

B. ReversiV

1) Simplification: In theory, since $U(s, a)$ asymptotically converges to $Q(s, a)$, MCTS still works if we have only $v(s)$. So I removed the policy part from the network and simply set $P(s, a) = \frac{1}{|A|}$ where A is the set of all valid moves at state s. Despite of the problems introduced with robustness,

the simplification saved the network from the heavy burden of having to learn a complex non-linearity and thus drastically reduced the number of parameters needed.

Assume even exploration over all the valid moves, we have

$$U(s,a) = Q(s,a) + c_{puct} \cdot \frac{1}{|A|} \frac{\sqrt{N(s)}}{1 + \frac{N(s)}{|A|}}$$

where $N(s) = \sum_a N(s,a)$ is the total number of expansions made at state s. Typically, more than 2000 iterations can be performed from the root state, $|A| \ll N(s)$, the second term is less than 0.05 with c_{puct} set to a constant near 1.

2) Network Model: RevNet: A state is represented as a 8×8 matrix. Disks belong to the current player are represented as 1, those of the opponent's as -1, and 0 otherwise. From observation and intuition, it can be evaluated primarily by a weighted sum of occupation, i.e.assign different scores to each position on the board. Although simply using a static weight matrix could give reasonable performance already, the scores should be dynamically adjusted with respect to the current situation to obtain a better evaluation.

AlphaZero and other Deep Reinforcement Learning usually employ deep and complicated convolutional neural networks that are capable of learning vary complex mappings. But here, with quite limited number of parameters allowed, a fairly simple network is proposed following straightly from the intuition.

The 8×8 matrix is flattened to a state vector v and fed into a 64×64 fully-connected layer. The output is then multiplied by v, and passed on to several more feed-forward layers. Or alternatively it can be considered as a convolution layer with 8×8 kernel and 64 channels. This serves as the basic block and can be duplicated(2 in my final implementation)to improve performance. The last FC layer produces a score normalized into $(-1, 1)$ by taking a $\tanh$ activation.

Several criteria were used to verify the capacity of the network:

- ability to overfit on small or special data,
- training loss,
- ability to improve in the learning process.

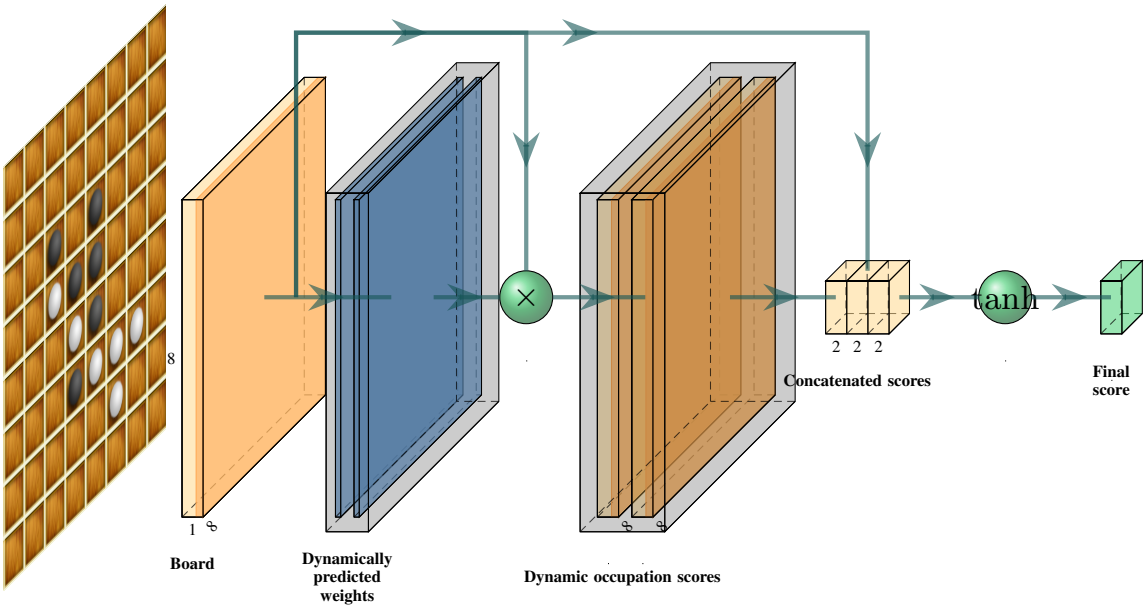
3) Fast and Stable Training: AlphaZero improves itself through pure self-play, which is more exploratory but less stable. To speedup the training process and increase stability, ReversiZero was trained simultaneously with several baselines and itself. That is, part of the training examples are gathered from playing against baselines, and the rest from self-play where additional randomness(temperature and noise) is added. A new model is accepted only when it performs no worse than before when tested against all its opponents and the previous version of itself. With good enough baselines, we can assume pareto improvement is being made.

4) Over All Strategy: ReversiV combines MCTS and Minimax search to ensure definite victory when possible. We define turn to be the number of disks occupied by either sides minus 3 so that the first move taken is at turn=1. Use of the two methods are controlled by a parameter cut. MCTS is used to find the best move until turn$\leq cut$, and Minimax takes over afterwards.

Minimax here doesn't look for maximum utility possible, but just search all the way down to the terminal state for a win. That is, we don't actually care how big an edge can we have over the opponent as long as the move does lead to victory.

And fortunately, $Q(s,a)$s(or $N(s,a)$s) estimated by MCTS can help with deciding the order to perform subsequent Minimax search at each state. At each state s, the valid moves $a \in A$ are sorted in descending order by $Q(s,a)$ to reduce the

Fig. 1. Network



number of states needed to be searched.

5) Algorithm Implementation:

a) Data Structure: The following are maintained throughout the game(but not across games):

- $Q(s,a)$: the expected reward of taking a at s
- $N(s,a)$: number of search performed by taking a from s
- $N(s) = \sum N(s,\cdot)$
- $E(s)$: whether s is a terminate state.
- $V(s)$: all the valid moves at s

b) Pseudo Code:

Algorithm 1: MCTS with Neural Network

Input: Board State s
Output: Best Move a
Function *MCTS(State s)* $\rightarrow$ *Action*:
  initialization;
  while *not timed out* **do**
    $Search(s)$;
  end
  $a := \arg\max N(s,a)$;
  return a
end

Function *Search(State s)* $\rightarrow$ *Value*:
  if *s is a terminal state* **then**
    return $GameResult(s)$;
  end
  $V(s) := GetValidMoves(s)$;
  if *s is a leaf node* **then**
    $v := Network(s)$;
  else
    $a := \max_{a \in V(s)} UCB(a)$;
    $s' := GetNextState(s,a)$;
    $v := Search(s')$;
    $Update(Q(s,a), N(s,a))$
  end
  return $-v$
end

III. Empirical Result

A. Data Collection

1) Dateset1: A first dataset contains states where there are 54 disks occupied, then the outcome is definitely determined by minimax search and labeled. 2000+ games were played using a random strategy, the states were gathered and augmented using symmetry to obtain about 100,000 examples.

Dataset1 is used to test the capacity of the neural network as a classification task. The network is adjusted to predict the label(+1 or -1) according to the sign of its score. The network proposed can achieve over 90% training and testing accuracy within 20 epochs.

2) Training Data: The actual data used for training an agent is gathered dynamically by playing against several baselines(ranked around 10 in the final week) and itself as described before. The states during each game are recorded from the corresponding player's view and later labeled with the final outcome.

Although theoretically the evaluation learned over time is an unbiased estimate of the expected reward, high variance is introduced due to self-conflicting examples. That is, because a strategy is initially weak and there is randomness in the process, the same(or similar) state can lead to either a win or a loss across different games. Even when it get stronger, the problem still exists and the network may get confused by the data.

To relieve this problem, states are only gathered from the middle stage of the game(for example, after turn=12). Besides, since the estimation is under the current policy, the data that are too old are discarded periodically.

B. Performance Measure

Arenas were built as both training and testing platform. The number of search iterations can be performed at each step on my machine with timeout=5.0s is shown below:

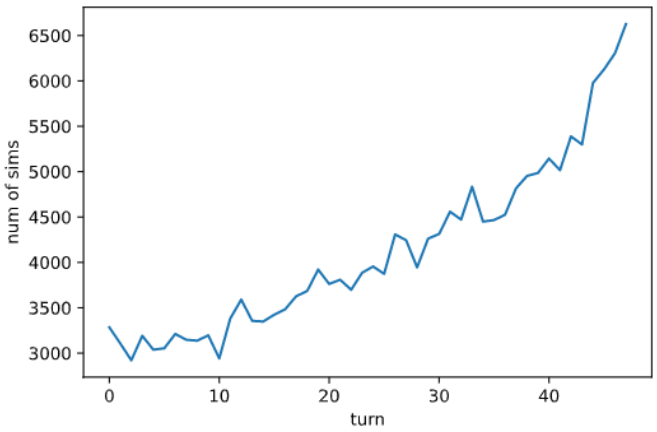


Fig. 8. Number of simulations(iterations) performed at each turn

One thing good about MCTS is that it always utilizes the time to the fullest and never times out.

C. Training Process

Training was done on the server parallelly with pipelined play-train-test process. For each iteration, 80˜120 games are played. It took less than 6 hours for the trained version to defeat the baseline with 10-0. And I terminated the process on iteration 50 which took 10 hours in total.

A serious problem is overfitting. The first to achieve a 10-0 was proved to be very weak against other opponents. While the later version refined by self-playing is far more robust.

D. Hyperparameters

The coefficient c_{puct} is set to 1.2 during training for exploration and 1.1 when actually used for stability.

To ensure victory when possible, Minimax algorithm takes over at the final stage where turn$> cut$. Generally, the better strategy you use in the early and middle stage, the easier it is to find a definite win in the final stage. In my implementation, cut is set to 48. But in optimistic situations it can be further reduced to 44 or even less.

E. Results

The following results are achieved:

- Passed all utility test.
- Ranked 5 in the points race.
- Ranked 8 in the round robin.

which shows that the evaluation learned by the agent is generalizable. With further training a higher level was achieved by a new version and is expected to do better than the version submitted.

Note that the testing environments, especially that of the round robin, is unfair to ReversiV due to performance issues, i.e.with many matches going on simultaneously the number of search iterations can be performed is affected and its behavior may become unstable.

F. Conclusion

This project demonstrated the most basic idea of how to get an intelligent agent via a data-driven approach, that is, to find and fit the underlying distribution. While somehow encouraging, the limitations are also obvious, which is why I didn't named it ReversiZero:

- The evaluation used, i.e.the neural network is not expressive enough. It is derived from straight intuition, and may fail to capture the complex patterns among the enormous state space.
- Eliminating the policy part from the model introduces serious problem with robustness. Working with a well-fitted ideal v does give the best strategy, but the problem is that at training time, there is no explicit improvement made to the policy. And because there is no $\vec{p}$ giving an initial policy, the behavior of the model becomes more sensitive to the number of search iterations can be performed, which means when the testing environment changes, especially in performance, the behavior can be far off what is expected.
- There is no guarantee for convergence of any kind. Even for a simple board game like Reversi, it's very hard to determine whether an agent is "improving". Merely defeating the previous version and several baselines is not a sufficient proof. Chances are that it may stuck in some local extremum, or falls into a "cycle" and get confused. But the need for more carefully managed training and testing contradicts with our motivation that we want a general framework which requires little or no human knowledge.

Nevertheless, it remains as an approach worth trying in terms of obtaining intelligence .

ACKNOWLEDGMENT

Inspired by Simple Alpha Zero.

REFERENCES

[1] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[2] 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(7676):354–359, 2017.