**Deep Reinforcement Learning**

**For Othello (Reversi)**

Final Project

DATA130011.01  
Neural Network and Deep Learning  
School of Data Sciences  
Fudan University

**Rong Ye**

**Yuqing Xie**   
[14307130361@fudan.edu.cn](mailto:14307130361@fudan.edu.cn)  
Mathematics and Applied Mathematics  
School of Mathematical Sciences

**Abstract**

**Introduction**

**The Othello Game**

**Rule**

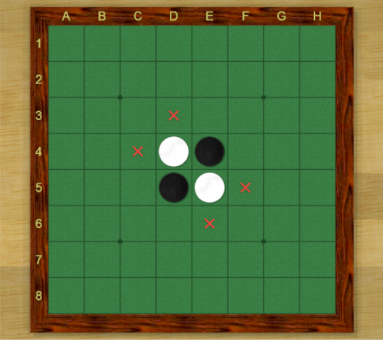
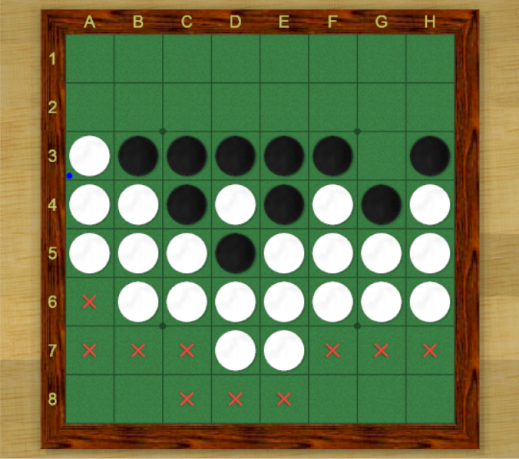
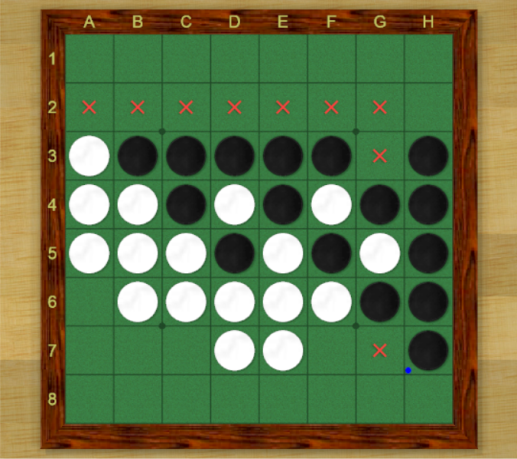
Othello is a two-player game on a 8 \* 8 board. There are 64 identical pieces which are white on one side and black on the other. The game begins with each player having two pieces placed diagonally in the center of the board **(Figure 1)**. The black player moves first. A move is legal if the newly placed piece is adjacent to an opponent’s piece and causes one or more of the opponent’s pieces to become enclosed from both sides on a horizontal, vertical or diagonal line. The enclosed pieces are then flipped. The black player's first move can be any one of four Red Cross locations, in **(Figure 1)**. Players alternate placing pieces on the board. If and only if a player does not have any legal move the player passes. The game ends when neither player has a legal move (of course when the board is filled with pieces), and the player with more pieces on the board wins. If both players have the same number of pieces, the game ends in a draw.

Figure 1 Othello play board and initial state.

**Features & Difficulty**

Othello is a perfect information, zero-sum, two-player strategy game. Despite simple rules, the game is far from trivial. The number of legal positions is approximately and the game tree has in the order of nodes [5], which precludes any exhaustive search method. Othello is also characterized by a high temporal volatility: a high number of pieces can be flipped in a single move, dramatically changing the board state.

**The GYM package:**

Gym is a Python package that offers the basic game settings of Othello. Useful information can be received from the env module.

**Related work**

[1] Introduced CNN and using professional game records for training.

This paper verifies whether CNN-based move predictors prove effective for Othello. They compare several CNN architectures and board encodings, augment them with state-of-the-art extensions, train on an extensive database of experts’ moves, and examine them with respect to move prediction accuracy and playing strength.

[2] Mainly compares different reinforcement learning methods' performance

This paper compares three strategies: Learning by self-play, learning from playing against a fixed opponent, and learning from playing against a fixed opponent while learning from the opponent’s moves as well in using reinforcement learning algorithms, to let an artificial agent learn to play the game of Othello, using algorithms Q-learning, Sarsa and TD-learning. It is found that the best strategy of learning differs per algorithm. Q-learning and Sarsa perform best when trained against the fixed opponent they are also tested against, whereas TD-learning performs best when trained through self-play. Learning from the opponent’s moves leads to worse results compared to learning only from the learning agent’s own moves.

[3] The best known Othello playing program is LOGISTELLO.

This paper surveys the evaluation and **search techniques** utilized by the strongest Othello programs of their time during the past twenty years. In this time span computer Othello has experienced considerable progress which culminated in the convincing 6-0 victory of LOGISTELLO against the then World-champion Takeshi Murakami in 1997. The focus of this article is the evolution of Othello evaluation functions and **heuristic search techniques** which quite nicely reflect the general Artificial Intelligence trend of replacing slow and error-prone manual tuning by automated machine learning approaches.

**Reinforcement learning model**

**Basic reinforcement learning model**

In reinforcement learning, the learner is a decision making agent that takes actions in an environment and receives a reward (or penalty) for its actions in trying to solve a problem. After a set of trial-and-error runs it should learn the best policy, which is the sequence of actions that maximize the total reward.

We define our problem as a Markov decision process.  
(1) A finite set of states ; In Othello, s means a certain broad state and S contains all possible states of the game, approximately states.  
(2) A finite set of actions ; In Othello, a means a legal position to place piece.  
(3) The transition function is simply to place a piece chosen by action a after state s.  
(4) A reward function , providing the reward the agent will receive for executing action a in state s, where denotes the reward obtained at time t; In Othello, a reward is received only when the whole game comes to an end. Also in our project, official judger does not consider how many pieces are wined, but only care whether black player win the game. We will try this reward function and also consider other choice to improve the agent's performance.  
(5) A discount factor which discounts later rewards compared to immediate rewards.

Policy is the decision an agent make at each state of the game. Usually, policies are classified as deterministic policy or stochastic policy. Deterministic policy is a fixed function from S to A. Stochastic policy gives the possibility of choosing a certain action at a certain state. According to the huge state space, we will use stochastic policy in our learning process.

We also have to define value functions:   
We want our agent to learn an optimal policy for mapping states to actions. The action to be taken in any state s is. The value of a policy , , is the expected cummulative reward that will be received when the agent follows the policy starting at state s. It is defined as: . The optimal policy is to achieve largest state value in all states. We sometimes also use Q-value () instead of value function.

**Learning algorithm framework**

To compute value function in contentious state space or action space, as well as huge state or action space, we can use approximation function , for example a neural network depending on parameter .

During training process we update Q-value using this formula:  
 ,   
where*α* is the learning rate

To prevent high correlation between samples because of learning goal’s dependency on parameters, we could use experience replay method, meaning construct a experience pool to de-correlation.

Following is a detailed pseudocode for the basic algorithm using network to learn the Q-value.

Initializing:

**States space S** (3\*8\*8): Where S[0] and S[1] represent positions of black pieces and white pieces respectively, S[2] represents the spaces where has no piece in it.  
**Actions space A**: From 0 to 63 along with 65. 0 to 63 represent positions on the board and 65 represents none legal move is available.  
Set super parameters: **discount rate** , **learning rate** Initialize the Q-network and its parameters  
Initialize the experience pool

Training:

Repeat:

Set S to initial state  
 Repeat:

# Black player’s turn  
 State\_black\_t = State\_white\_(t-1)’  
 Get legal positions at State\_black\_t for black player  
 Try all legal actions a and get the rewards\_black\_t, and new state\_black\_t’  
 Add reward and new state into experience pool D\_black  
 Choose action according to Q-value\_black  
 Execute action a

# White player’s turn  
 State\_white\_t = State\_black\_t’  
 Get legal positions at State\_white\_t for white player  
 Execute opponent’s policy to get reward and new state\_white\_t’

Until Game over  
sampling from experience pool D\_black sample\_state\_black, sample\_action\_black, sample\_reward\_black and sample\_new\_state\_black’  
compute loss function:   
 , where y=reward when new-state is terminal state, or update otherwise  
update black player’s Q-value network parameters to minimize loss function

Until convergence

**Different opponents:**

The algorithms' performance also vary when the opponent change. We have fixed policy opponents, experts playing records, best players in the world and we also tried self-play opponent inspired by Alpha-Go.

First we consider fixed-opponents:

1. DQN vs. random: white player choose action randomly from legal positions
2. DQN vs. greedy: white player choose the legal position that reverse most black pieces
3. DQN vs. minimax: white player choose action according to pruning minimax search algorithm
4. DQN vs. experts: using professional game database’s recorded as training samples.

Then we also consider to use **self-play opponent** for training – just replace the white player as another same agent as the black player. Training process is a little different with the previous algorithm:

Repeat:

Set S to initial state  
 Repeat:

# Black player’s turn  
 State\_black\_t = State\_white\_(t-1)’  
 Get legal positions at State\_black\_t for black player  
 Try all legal actions a and get the rewards\_black\_t, and new state\_black\_t’  
 Add reward and new state into experience pool D\_black  
 Choose action according to Q-value\_black  
 Execute action a

# White player’s turn  
 State\_white\_t = State\_black\_t’  
 Get legal positions at State\_white\_t for white player  
 Try all legal actions a and get the rewards\_white\_t, and new state\_white\_t’  
 Add reward and new state into experience pool D\_white  
 Choose action according to Q-value\_white  
 Execute action a

Until Game over  
sampling from experience pool D\_black sample\_state\_black, sample\_action\_black, sample\_reward\_black and sample\_new\_state\_black’  
compute loss function:   
 , where y=reward when new-state is terminal state, or update otherwise  
update black player’s Q-value network parameters to minimize loss function

sampling from experience pool D\_white sample\_state\_white, sample\_action\_white, sample\_reward\_white and sample\_new\_state\_white’  
compute loss function:   
 , where y=reward when new-state is terminal state, or update otherwise  
update white player’s Q-value network parameters to minimize loss function

Until convergence

**Different neural network structure**

According to the huge amount of states, we have to learn the policy using statistic methods and network structure may affect learning proficiency greatly.

We used different neural network for Q-value function: Full connected linear network and Convolutional neural network.

Full connected linear network is clear:

The input layer is the same as size of S:3\*8\*8  
The hidden layer is a full connected linear net with 2\*8\*8 neurons

The output layer is 8\*8 in size, represent the possibility in each position for next action.

For Convolutional neural network:

**Here are some points that we should focus on:**2) The reword function is also critical for learning ability  
  
5) We can also consider policy gradient and actor-critic to train the agent.

**Experiment**

DQNselfplay训练结果：

Iterate201次：黑棋胜87，白棋胜113（43.5%，57.5%）

多次尝试都是后下的白棋更好

经验回放DQN和minimaxalpha-beta剪枝对抗结果：

黑棋（先手minimax）vs白棋（后手DQN）：

黑棋（先手DQN）vs白棋（后手minimax）：

6.

黑子（先手）DQNvs Random play训练：

设置一层全连接，reward0+/-1，iteration200次

黑胜（DQN）：128，白棋（rnd）：62

Test：

Vs. rand

Vs. ab

Vs. self-play(白)

**Compare& analysis**

**Reference**

[1] Learning to Play Othello with Deep Neural Networks  
[2] Reinforcement Learning in the Game of Othello: Learning Against a Fixed Opponent and Learning from Self-Play  
[3] ADVERSARIALADVANTAGE ACTOR-CRITIC MODEL FOR TASK-COMPLETION DIALOGUE POLICY LEARNING（最后修改value学习部分可以参考这个）  
[4]M. Buro, “The evolution of strong Othello programs,” in Entertainment Computing - Technology and Applications, R. Nakatsu and J. Hoshino,Eds. Kluwer, 2003, pp. 81–88. <https://skatgame.net/mburo/>  
<https://skatgame.net/mburo/log.html>  
[5]Victor L Allis. *Searching for solutions in games and artificial intelligence*. PhD thesis, University of Limburg, Maastricht, The Netherlands,1994