Reinforcement Learning in the Game Reversi: Deep Q Network and Monte Carlo Tree Search

YIQING MA
School of Data Science
14300180095@fudan.edu.cn

YUNFAN LI
School of Software Engineer
16302010002@fudan.edu.cn

YUQING CAI
School of Mathematical Economic
14300680092@fudan.edu.cn

Abstract

Board game playing is a popular area within the field of Reinforcement Learning.Past few years have seen many breakthroughs using reinforcement learning to play the Atari games with the help of the Gym environment. This paper compares four strategies in using reinforcement learning to teach the agent to learn to play the game of Reversi. These four strategies are listed following the order from simple to complex: Simple Q-learning network, Double-Simple Q-learning network, Deep Q-learning network and a self-play based algorithm using neural networks for policy estimation and Monte Carlo Search Tree for policy improvement, also called the AlphaZero algorithm. These four reinforcement learning algorithms are trained and tested against two fixed opponents. One is the random agent, the other is the alpha-beta agent. We can obtain a pretty good AI model of playing Reversi after a few hours' training. In view of the game result, we can obtain a gradually growing winning percentage with the improvement of the model step by step.

1 Introduction

Reinforcement Learning : Those interested in the world of machine learning are aware of the capabilities of reinforcement learning-based AI. The past few years have seen many breakthroughs using reinforcement learning(RL). RL is currently excelling in many game environments. In short, Reinforcement learning is a computational approach where an agent interacts with an environment by taking actions in which it tries to maximize an accumulated reward. So that in this condition, RL can guide an AI to choose the optimal decision.

Gym : Gym is a collection of environments designed for testing and developing reinforcement learning algorithms. With the help of Open AI Gym ,one can try different kinds of games without creating the complicated environment of them. Gym is written in Python, and there are multiple environments such as Robot simulation and Atari games.

Othello(Reversi) : Reversi is a prefect-information, zero-sum game played on an 8*8 board.There are sixty-four identical game pieces called disks.There are two players placing at alternate turns one disc at a time. During a play, any disks of the opponent's color that are in straight line 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. Reversi is much simple than go and chess. Its calculations are relatively small, and the possible moves are gradually limited as the game progresses. So that Reversi provide a good test

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

to apply reinforcement learning. We can focus on the training scheme of RL.

Approach : There are many existing approaches for designing systems to play games. Generally speaking, it can be divided into two categories. Firstly, the system relies on availability of expert domain knowledge to train the model on and evaluate non-terminal states. Secondly, the agent like AlphaGo Zero is trained through self-play without guidance from humans. In our work, we mainly extract ideas from the Reinforcement Learning and Alpha Zero, and apply them to the game of Reversi. We use board size of 8*8 and we tried training models from pure simple Q-learning networks, double simple Q-learning networks, Deep Q-learning networks and Monte Carlo Tree Search. For evaluation, we compare our trained agents to random baselines, and ab-pruning minimax agent with evaluation functions and found our model achieve good performance very quickly.

2 Methods

In this part, firstly we summarize the Reinforcement Learning as basic knowledge in order to provide a good warm-up for the latter part of our model. With this groundwork laying, then we detailed introduced the core methods we used in the four agents step by step. In the introduction, we also attach the optimization and the trick we added for our agent to achieve better superhuman performance.

Reinforcement Learning

Reinforcement learning is defined as a class of problems learning how to map situations to actions to maximize the numerical reward signal.The basic RL is modeled as a Markov decision process. The process of RL can be partitioned into several discrete time steps. The procedure how the agent interacts with its environment is described as following. At each time step, the agent receives an observation and chooses an action from the set of available actions, which is subsequently sent to the environment.A policy defines the learning agent's way of behaving at a given time. A reward function defines the goal in a reinforcement learning problem. A value function specifies what is good in the long run. A model of the environment mimics the behavior of the environment. The environment then moves to a new state and the reward associated with the transition is determined.

One of the challenges in RL is the tradeoff between exploration and exploitation. Exploration refers to the taking the actions that haven't been performed before, so as to explore more possibilities. Exploitation refers to the actions that have been tried in the past and found to be effective, which is preferred to improve the model. An agent has to exploit what it already knows to get reward. On the other hand, it also needs to explore in order to make better action selections.

There are three canonical RL algorithms, TD-learning, Q-learning and Sarsa. TD-learning learns by sampling the environment according to some policy, and is related to dynamic programming techniques as it approximates its current estimate based on previously learned estimates. Q-learning works by learning an action-value function, which ultimately gives the expected utility of taking a given action in a given state and following an optimal policy there after. The difference between Q-learning and SARSA is that when the algorithm is updated, the next state and action of Q-learning is still uncertain, while the state and action of SARAS has been determined. The algorithm we choose here is Q-network, an upgraded version of Q-network, which will be introduced later.

2.1 Agent 1.Simple Q-learning Network

In this first agent, we decide to simply implement Q-learning. Since Q-learning is a table of values for every state (row) and action (column) possible in the environment. Cause the state and action in the particular Atari Game "Reversi" is fixed. So a Q-learning table is enough for the agent to decide the policy and choose the best action within a given state.

We start by initializing the table to be uniform (random choose around zeros) and than we observe the rewards we obtain for various actions, then we can update the table accordingly using the function called "Bellman equation" this states that the expected long-term reward for a given action is equal to the immediate reward from the current action combined with the expected reward from the best future action taken at the following state. We also reuse this Q-table when estimating how to update

our table for future actions. By updating the Q-value In this way, the table slowly begins to obtain accurate measures of the expected future reward for a given action in a given state.

This equation is:

$$Q(s, a) = r + \gamma(max(Q(\acute{s}, \acute{a}))$$

Next we consider add the neural networks into our models. Because in the real-world environment, the possible states is nearly infinitely larger and the actions may not be fixed in certain state. We instead need some way to take a description of our state and produce Q-values for actions without a table :that is where neural networks com in. We can take any number of possible states that can be represented as a vector and learn to map them to Q-values.The method of updating is a little different as well. In the network we use a back propagation and a loss function.

Last we use the framework Tensorflow for constructing our neural network model. The tensorflow is easy when training a neural network cause it has included every function needed in the process of update and back propagation.

we mainly complete the following four things in this Simple Q-learning model:

- Construct a Q-value table for every state (row) and action (column).
- Use "Bellman equation" to renew the value of Q-table.
- Construct a Neural Network to substitute the Q-value table.
- Combine the Tensorflow framework.

2.2 Agent 2. Double Simple Q-learning network

There are two strategies when training a neural network :Learning by self-play, Learning from playing against a fixed opponent. When we train the Simple Q-learning network, it attempts to answer the following question: How does the performance after learning through self-play compare to the performance after playing against a fixed opponent, whether paying attention to its opponent's moves or just its own?So we do one modify to our strategy and our Q-learning agent goes through these following steps:

Algorithm 1 learning by play with agent algorithm

1) Observe the current state s_t
2) For all possible action a_t in s_t use NN to compute $Q(s_t, \acute{a_t})$
3) Select an action at using a policy π
4) According to (8) compute the target calue of the previous state-action pair $Q^{new}(s_{t-1}, a_{t-1})$
5) Use NN to compute the current estimate of the value of the previous state action pair $(Q(s_{t-1}, a_{t-1}))$
6) Adjust the NN by back propagating the error $Q^{new}(s_{t-1}, a_{t-1}) - \hat{Q}(s_{t-1}, a_{t-1})$
7) $s_{t-1} < -s_t, a_{t-1} < -a_t$ 8) Execute action a_t
9) Compute the target value of the opponent's previous state action pair $Q^{new}(s_{t-1}, a_{t-1})$ 10) Use NN to compute the current estimate of the value of the opponent's previous state action pair $\hat{Q}(s_{t-1}, a_{t-1})$
11) Adjust the NN by back propagating the difference between the target and the estimate.

Besides, in order to apply our new strategy for training , we do one modification to our tensorflow model. Black weight and White weight and their Q values are separately set. Because the black agent and the white agent are faced with different situations and two separate weight parameters are beneficial for the agent to gain better rate of winning.

2.3 Agent 3. Deep Q-learning network.

Since the ordinary Q-network was barely able to perform as well as the alpha-beta agent in a simple game environment, we are starting to improve it . Learning from Google DeepMind team, who

achieved superhuman performance on dozens of Atari games using their DQN agent, we found that Deep Q-Networks are much more capable, and the wining rate begin to soar after we took this approach of transforming an ordinary Q-Network into a DQN.

Improvement 1: Convolutional Layers

Instead of considering each pixel independently, convolutional layers allow DQN to consider regions of an image, and maintain spatial relationships between the objects on the screen as we send information up to higher levels of the network. In this way, they act similarly to human receptive fields.In Tensorflow, we can utilize the

convolutionlayer =tf.contrib.layers.convolution2d(inputs,numoutputs,kernelsize,stride,padding)

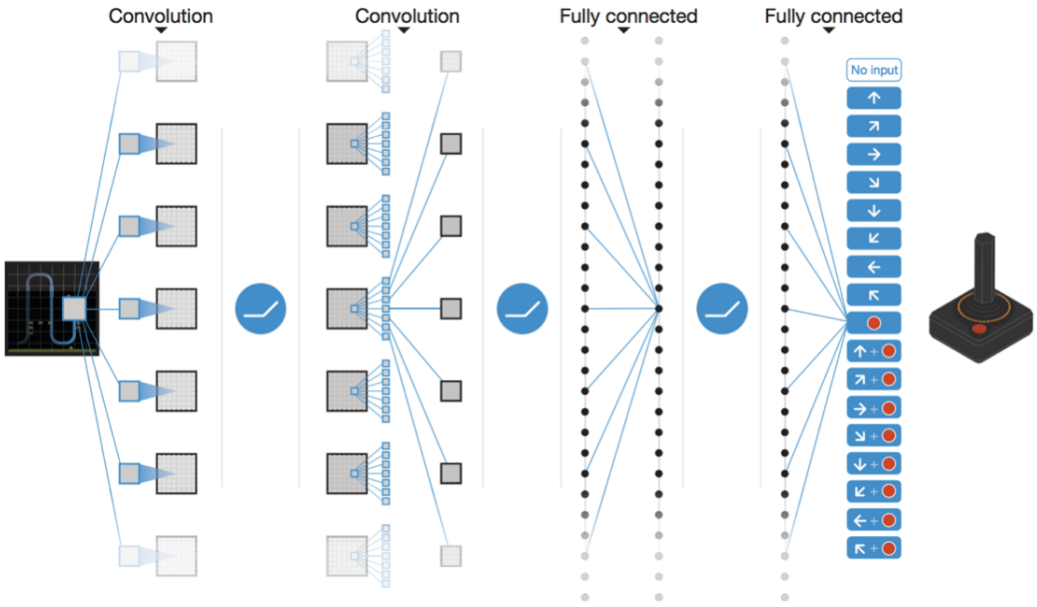

Figure 1: Convolution Neural Network.

Improvement 2: Experience Replay

The second major addition to make DQNs work is Experience Replay. The basic idea is that by storing an agent's experiences, and then randomly drawing batches of them to train the network, we can more robustly learn to perform well in the task. By keeping the experiences we draw random, we prevent the network from only learning about what it is immediately doing in the environment, and allow it to learn from a more varied array of past experiences. In the experiment, experiences are stored as a tuple of <state,action,reward,next state>.we build a simple buffer to store them by Tensorflow .When the time comes to train, we simply draw a uniform batch of random memories from the buffer, and train our network with them.

Improvement 3: Separate TargetNetwork

The third major addition to the DQN that makes it unique is the utilization of a second network during the training procedure. This second network is used to generate the target-Q values that will be used to compute the loss for every action during training.Why not use just use one network for both estimations? The issue is that at every step of training, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target network's weights are fixed, and only periodically or slowly updated to the primary Q-networks values. In this way training can proceed in a more stable manner.

Improvement 4: Double DQN and Dueling DQN

Double DQN is approached to avoid overestimated values, resulting in overoptimistic value estimates. Dueling DQN can more quickly identify the correct action during policy evaluation as redundant or similar actions are added to the learning problem.

For Double-DQN. the authors of DDQN paper propose a simple trick: instead of taking the max over Q-values when computing the target-Q value for our training step, we use our primary network to chose an action, and our target network to generate the target Q-value for that action. Thus we are able to substantially reduce the overestimation, and train faster and more reliably. Below is the new DDQN equation for updating the target value.

$$Q - Target = r + \gamma Q(s', \underset{a}{argmax}(Q(s', a; \theta), \theta'))$$

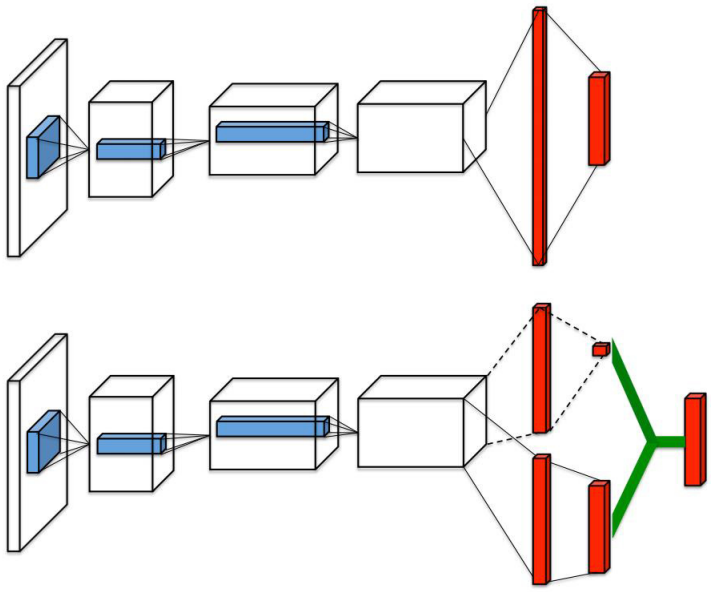

Figure 2: Above: Regular DQN with a single stream for Q-values. Below: Dueling DQN where the value and advantage are calculated separately and then combined only at the final layer into a Q value.

For Dueling DQN. The goal of Dueling DQN is to have a network that separately computes the value and advantage functions.The first is the value function V(s), which says simple how good it is to be in any given state. The second is the advantage function A(a), which tells how much better taking a certain action would be compared to the others. We can then think of Q as being the combination of V and A. More formally:

$$Q(s, a) = V(s) + A(a)$$

we mainly complete the following four things in this Deep q-learning model:

- Upgrading from a single-layer network to a multi-layer convolutional network.
- Implementing Experience Replay, which will allow our network to train itself using stored memories from it's experience.
- Utilizing a second "target" network, which we will use to compute target Q-values during our updates.

- Drawing into the Double DQN and Dueling DQN to avoid over estimate and achieve more robust estimates of state value by decoupling the Q-value into advantage and value functions.

2.4 Agent 4. AlphaGo Zero Monte Carlo Tree Search.

The framework of the first three models used Reinforcement learning and Q-learning neural network, they can be summarized into one category. While this forth agent is totally the different. It used pure MCTS. Which is a searching method similar to alpha-beta minimax search agent. Next is the introduction of MCTS.

The Monte Carlo Tree Search (MCTS) can be viewed as a kind of enhanced learning. It gradually builds an asymmetric tree that can be broken down into four steps and iteratively repeated

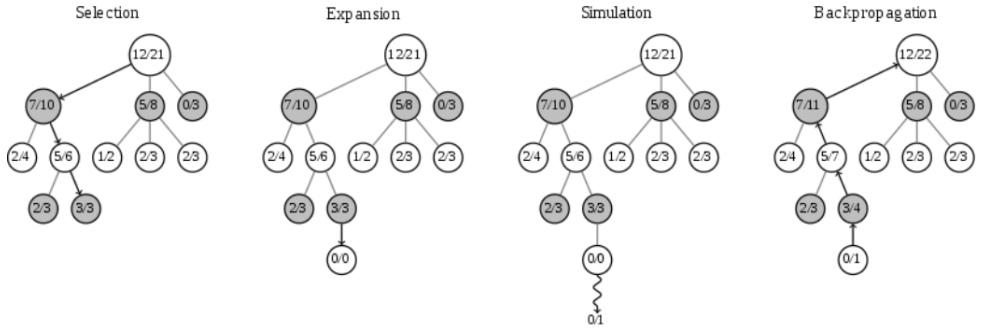

Figure 3: Monte Carlo Tree.

1.Selection Starting from the root node, select the best child node under the Tree Policy, until we reach the leaf node. Tree Policy is the strategy of selecting nodes, its quality has a direct impact on the quality of the search. The strategy is UCT, Upper Confidence Bounds for Trees.The basic idea of UCB is not to select the highest average return, but to select the highest confidence limit of the machine.we find a feasible move with the maximum UCB using a UCB formula, and once again check the moves of the new situation.

2.Expansion Extend the leaf node to add one or more viable children as the child node of the leaf node. This sub-node is obtained by taking move under situation .

3.Simulation According to the Default Policy from the extended position of chess to the final. Starting from current situation, both sides began to randomly play the game. Eventually we get a win or loss info, and update the node's winning rate. Of course completely random nature is relatively weak, by adding some prior knowledge and other methods to improve this part can be more accurate estimation of the value of the child, increasing the chess power of the program.

4.Back propagation After the simulation of has ended, its parent and all its ancestors update their winning rate. The winning rate of a node is defined as the average winning rate of all child nodes of this node. we propagates back to the root node R, such that the winning rate of all the nodes in the path can be updated.

After that, iterate the process from step 1. The more situations we include, the higher the victory rate we obtain. Finally, we choose the move that has the highest winning rate. In practice, MCTS can go along with a lot of improvements. Most of the famous artificial intelligence now has made a lot of improvements on this foundation.

3 Experiments Design

Experiment : In this section, I introduce the whole process of the experiment. In order to evaluate the result of our agent. we let them to compete with a given agent. This given agent could be random

Algorithm 2 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

agent, and an alpha-beta minimax search agent. The purpose of the experiment is to train a set of parameters of the neural network to let our agent learn superhuman performance. We are required to adjust the parameters, add tricks like experience replay and do optimization to our model. When upload our agent to the server, we are required to win the server's agent of level 1, 2, 3 and reach the winning rate larger than 90%.

For local test, we wrote a test for random.py and a test for alpha beta.py. We view the result of competing with the random agent as a baseline.While we view the result of competing with the alpha-beta agent with 1-6 level search as an evaluation of the experimental result. Since the alpha-beta search and Q-learning are fixed policy after the training process is done. As for the alpha-beta agent, we only need to run one epoch of test for each search depth. As the higher the number of alpha-beta agent search layer is , the more accurate the search result is. So beating the 6-layer alpha-beta agent is our goal. Meanwhile, since playing chess is a zero sum game, in the process of our experiment, we need to consider the differences between the black agent and the white agent.

Setting : In this section, I describe my 4 agents' setting and env setting.For Reversi environment, It's a board of 8*8. Through the command "observation, reward, done, info = env.step (action)", It means that one can execute an action and obtain the current state of the env. The reward equals to 1 for black wins and the reward equals to -1 for white wins. Action returns as a integer of 0-63, corresponding to the 64 states of the board. If the current plater has no possible move, the action returns as 65.

For Reinforcement Learning, the agent SQN, the Tensorflow placeholder's input's size is [1,64] transformed from the [8*8*3], in which the black site equals to 1, the white site equals to -1. The Tensorflow placeholder weight to be optimized is a [64*65] matrix. For the agent DQN, the Tensorflow placeholder's input's size is directly the original state's size [8*8*3], and it passes through 4 convolution layers and 1 fully connected layers.The batch size for experience replay is 32, with a replay buffer of 10000 trajectories. In both models the learning rate is setting to 0.001 and the training episode sets to 2000. For the agent MCTS, the key is the number of simulation. This is a trade off of time and accuracy. We choose 200 in our MCTS. In the following text, we denote the four agent as SQN, D.SQN,DQN, MCTS.

4 Result Analysis

We'd like to show our experiment result briefly through table and picture.

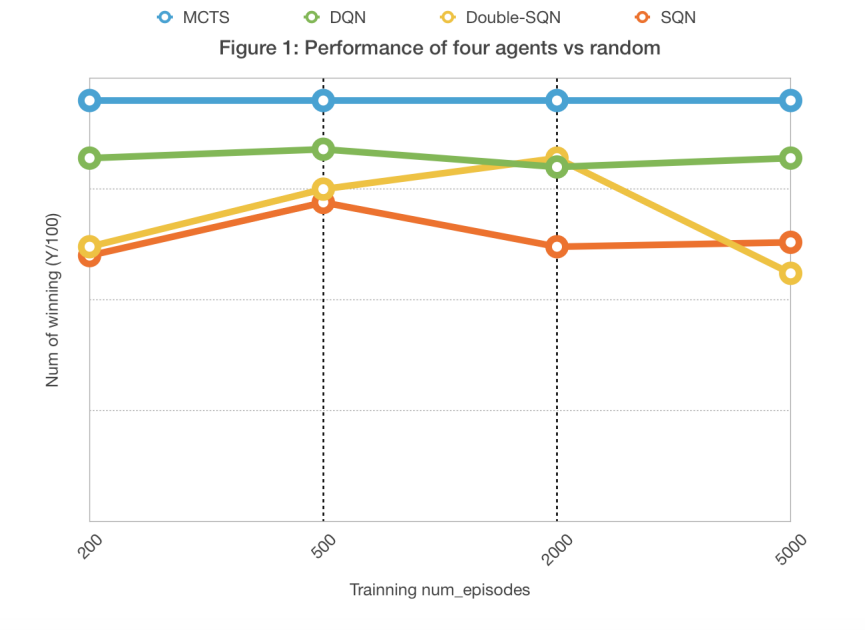

Figure 4: The performance of SQN DSQN DQN MCTS with random agent

Table 1: The performance of SQN DSQN DQN MCTS with alpha-beta agent with depth 1-6

Agent	Depth1	Depth2	Depth3	Depth4	Depth5	Depth6
SQN	O	X	X	X	X	X
D.SQN	O	O	X	O	X	X
DQN	O	X	X	X	O	X
MCTS	O	O	O	O	O	O

From Figure 1 (4 agents' performance with random agent) and Table 1 (4 agents' performance with alpha-beta agent with 1-6 depth), Note that the SQN agent's performance is worse than the other two RL agents in both tests.This is because the SQN 's state input is a [1,64] vector, comparing to the DQN's [3*8*8] dimension input, it drops the features' high dimension information in the procedure of training. The study also shows that DQN ranks the first, DSQN rank the second, and the SQN rank the third among the three RL agents as we expected.We can see the agent becomes more and more intelligent with more training episodes,as well as adding tricks like experience replay and convolution networks. We find that the experience replay comes into prominence when compared with other tricks. Meanwhile the agent MCTS performances well with the alpha-beta as the white player than the three RL agent. Than I uploaded the baseline model SQN and MCTS to the server and got the result listed in Table 2 and Table 3.

Table 2: Win rates against level 1-3 Using baseline SQN

Level	Games won/played on black	Games won/played on white
1	2/15	3/15
2	5/15	3/15
3	6/20	13/20

Table 3: Win rates against level 1-3 Using MCTS

Level	Games won/played on black	Games won/played on white
1	10/15	11/15
2	15/15	15/15
3	20/15	20/20

5 Conclusion

We implement four agents learning to play Reversi using SQN,DSQN,DQN and MCTS and resulting in convincing winning rates. For further study, based on the code provided in this job, we can recurrent AlphaGo Zero which is a combination of MCTS and Reinforcement Learning. This method can robustly obtain the higher winning rate. The original implementation is provided by DeepMind with a large computational power on industry hardware. In our work ,we show that it is possible to train an agent with superhuman performance on commodity hardware.

References

[1] Michiel van der Ree & Marco Wiering Reinforcement Learning in the Game of Othello:Learning Against a Fixed Opponent and Learning from Self-Play *Institute of Artificial Intelligence and Cognitive Engineering Faculty of Mathematics and Natural Sciences University of Groningen, The Netherlands*

[2] Paweł Liskowski &Wojciech Jaskowski& Krzysztof Krawiec Learning to Play Othello with Deep Neural Networks.

[3] Shantanu Thakoor & Surag Nair&Megha Jhunjhunwala Learning to Play Othello Without Human Knowledge.

[4] R. Sutton & A. Barto Reinforcement learning: An introduction. The MIT press, Cambridge MA, A Bradford Book, 1998

[5] Ioffe, S. & Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International Conference on Machine Learning, 448–456.

[6] Nijssen, J. 2007. Playing othello using monte carlo. Strategies 1–9.

[7]Silver, D. & Hubert, T.& Schrittwieser, J.& Antonoglou, I.& Lai, M.& Guez, A.& Lanctot, M.& Sifre, L.& Kumaran, D.& Graepel, T.& Lillicrap, T.& Simonyan, K.& Hassabis, D. 2017a. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXive-prints*