Play Othello Games with Improved DQN and MCTS Algorithms

Chengzhe Xu[*]
The School of Microelectronic Science and Engineering
Fudan University
Shanghai, Pudong, Zhangheng Road 825
15307110177@fudan.edu.cn

Hao Xiang[†]
The School of Microelectronic Science and Engineering
Fudan University
Shanghai, Pudong, Zhangheng Road 825
15307130110@fudan.edu.cn

Abstract

This article is about the final project of the implementation of Othello game program. We implemented Othello game programs with both Deep Q-learning algorithm (DQN) and Monte Carlo Tree Search algorithm (MCTS), and discussed the theories and characteristics, experimented performances of the two algorithms. To further improve the performances of the agent, we also discussed and implemented the improved versions of the algorithms, (eg. Double DQN algorithm (D-DQN) and Dueling architecture) and experimented training tricks (eg. Self-training, Data argumentation and Updating with selection) while training.

1 Introduction

Othello, or Reversi, is one of the traditional but challenging strategy board games for two players. In Othello players take turns placing game pieces with their assigned colors, called *disks*, onto the board. During a play, any disks of the opponent's color that are in a straight line or a diagonal 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. When the board is filled, the player whose disks are more that another's would be the winner. (For the situation that the two players have the same number of disks, then the game end in a draw.)

Reinforcement learning algorithms have been showing extraordinary talents in the field of strategy games (eg. Atari 2600). This article is going to summary and discuss our works on implementing a program to play Othello.

This article would be arranged in following sections: *a brief introduction*, *the reinforcement learning algorithm and their improvement used in the project*, *the experimental methods and tricks we applied when training*, *our experiments results and discussions*, and end in *summary*.

[*]student ID: 15307110177
[†]student ID: 15307130110

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

2 Reinforcement Learning Algorithm Used in Othello Project

In this section, we will introduce the algorithms we used to play Othello game and some of the improve methods towards them. This section will concentrate on the mathematical principles of them.

This section would be arranged in following subsections: *basic concepts of reinforcement learning*, *MDP procedure and Bellman equation*, *policy iteration*, *Value Iteration*, *Temporal-Difference Learning*, *Q-learning*, *Deep Q-learning*, *Double deep Q-learning*, *Dueling architecture of deep Q-learning*, and *MCTS algorithm with deep neural network*.

2.1 Basic Concepts of Reinforcement Learning

The reinforcement learning (RL) algorithms have been proved powerful to help agents interact with and learn from the environment, and make decisions according to the environment to maximum the total reward.

There are elements in the progress of decision: *state of the environment* s, *action of the agent* a, *the rules of statesf́ransition* (usually be represented as a probability states' transition), *the decision policy* π and the *reward gained from the environment* r_t.

Suppose that the space of environmentś possible states is S, the space of agentś possible actions is A, and under a specific state s, the space of actions allowed is A_s. With the notations, the progress of interaction between the environment and agent can be described as the decision progress: 1)According to the current state s_t, make decision on which action to take:$a_t = \pi(s_t)$, $a_t \in A_(s_t)$. 2)Take the action a_t, the environmentś state would be change to s_{t+1}, and the agent would get a reward

$$r_t = R(s_t, a_t, s_{t+1})$$

from the environment, according to the state transition and action taken by the agent. 3)Back to 1) until the state s_{t+1} is one of the final states.

The goal is to maximum the total rewards gained along the state transition chain l from initial state s_0 to final state s_T. Because there is always randomness in the transition chain, the total reward are represented by its expectation, obey the distribution of decision transition.

The *state value function* $V(s)$ is defined as the expectation reward of all the decision progress start at state s and *action-state value function* $Q(a, s)$ is defined as the expectation when the progress start at state s and take action a at the beginning.

$$V^{\pi}(s) = E_{l\in\pi}(\sum_{t=1}^{T}(\lambda^{t-1}R(s_t, a_t, s_{t+1})))\|_{s_0=s}$$

$$Q^{\pi}(a, s) = E_{l\in\pi}(\sum_{t=1}^{T}(\lambda^{t-1}R(s_t, a_t, s_{t+1})))\|_{s_0=s, a_0=a}$$

In the equation, λ is the reward discount factor, to make current reward more important than the long-term reward gained in the future, and further more, when the state transition chain is much longer, even has infinite length, the discount factor can ensure the total reward would converge to a finite number.

The best policy is the one that help the agent get the maximum reward, that is, select a policy π according to:

$$\pi^*(s_t) = argmax_{\pi}(V_{\pi}(s_t))$$

.

It is obvious that the best policy under the state s_t can be also computed with $Q(a, s)$:

$$\pi^*(s_t) = argmax_{a_t}(Q(s_t, a_t))$$

.

And under the optimized policy, the relationship between $V(s)$ and $Q(a, s)$ is:

$$V^* s_t = max_{a_t}(Q^*(a, s))$$

2.2 MDP Procedure and Bellman Equation

For many of the decision procedures (eg. most of the chess games), the probability of the next state only depends on the current state and the action to take on, and those who has this characteristic are called *Markovian Decision Process* (MDP). And for a MDP progress, according to the definition, we have:

$$P(s_{t+1}|s_t, a_t, ..., s_0, a_0) = P(s_{t+1}|s_t, a_t)$$

And in that case, the *state value* and *state-action value* can be write in the iteration form (*the Bellman equation*):

$$V^{\pi}(s_t) = \sum_{a_t \in A(s_t)} (P_{\pi}(a_t|s_t) * \sum_{s_{t+1}} (P(s_{t+1}|a_t, s_t) * (R(s_t, a_t, s_{t+1}) + \lambda * V^{\pi}(s_{t+1}))))$$

$$Q^{\pi}(s_t, a_t) = \sum_{s_{t+1} \in S} (P(s_{t+1}|a_t, s_t) * (R(s_t, a_t, s_{t+1}) + \lambda * V^{\pi}(s_{t+1})))$$

2.3 Policy Iteration

It has been proved that the comparison of two different policies, or the comparison of the state value under the two, is equivalent to the comparison between the state value under the one and the state-action value under the same policy, while the choice of action is based on the another policy (*Theory of Policy Optimization*):

$$sign(V^{\pi}(s) - V^{\pi'}(s)) = sign(V^{\pi}(s) - Q^{\pi}(s, \pi'(s)))$$

So according to the truth that optimization of policy is to maximum the state value, we can also derive an iteration form of the best policy:

$$\begin{aligned} \pi_i^* &= argmax_{\pi}(V^{\pi_{i-1}}(s)) \\ &= argmax_{a_t}(Q_{i-1}^{\pi}(a, s)) \\ &= argmax_{a_t}(\sum_{s_{t+1}} (P(s_{t+1}|a_t, s_t)(R(a_t, s_t, s_{t+1}) + \lambda * V^{\pi_{i-1}}(s)))) \end{aligned}$$

The equation reveals that the iteration of the policy π depends on the state value $V(s)$, and according to the Bellman equation, the iteration of $V(s)$ also depends on the current policy π:

$$V^{\pi}(s_t) = \sum_{a_t \in A(s_t)} (P_{\pi}(a_t|s_t) * \sum_{s_{t+1}} (P(s_{t+1}|a_t, s_t) * (R(s_t, a_t, s_{t+1}) + \lambda * V^{\pi}(s_{t+1}))))$$

$$\pi_i^* = argmax_{a_t}(\sum_{s_{t+1}} (P(s_{t+1}|a_t, s_t)(R(a_t, s_t, s_{t+1}) + \lambda * V^{\pi_{i-1}}(s))))$$

So $V(s)$ and π synergize with each other while co-iterating within the circle of Policy-Evaluation and Policy-Updating, and in a further more simple form, the policy iteration can be described as:

$$V^{\pi}(s_t) = \sum_{a_t} (P(a_t|s_t) * E_{s_{t+1} \in P(s_{t+1}|s_t, a_t)}(R(s_t, a_t, s_{t+1}) + \lambda * V^{\pi}(s_{t+1})))$$

$$\pi(s) = argmax_a(E_{s_{t+1} \in P(s_{t+1}|s_t, a_t)}(R(s_t, a_t, s_{t+1}) + \lambda * V^{\pi}(s_{t+1})))$$

While in practice, it is impossible to get an accurate expectation of total reward because that would require the weighted sum of all the possible state transition chains. The mean of a sampled batch is used to serve as an unbiased, effective and consistent estimation of the expectation:

$$\hat{Q}(a, s) = \frac{1}{N}(\sum_{i=1}^{N}(G(l_i)))$$

.

So the algorithm of policy iteration can be described as:

In the algorithm, the policy used while sampling is the ϵ-greedy policy, not the optimal policy we want. The use of ϵ-greedy policy helps to prevent the over-fitting on the policy (it can be regard as some kinds of regularization) and introduce more exploration into the policy. And the algorithm is in fact model-free, can be applied to the problems whether MDPs or not.

Algorithm 1 Policy Iteration

Input: The environment of the game, current state s;
Output: $\pi(a|s)$; Initial policy $\pi(a|s)$; $\pi^{\epsilon}(a|s) = \pi(a|s)$ with probability ϵ else random;

1: **for** $i = 1$ to T **do**
2: do Monte Carlo sampling under the ϵ-greedy policy π^{ϵ} with given s_t, and get state transition road l_1, l_2, ... l_N;
3: $\hat{Q}(a, s) = \frac{1}{N}(\sum_{i=1}^{N}(G(l_i)))$
4: $\pi(a|s) = argmax_{a'}(\hat{Q}(a^{'}, s))$;
5: $\pi^{\epsilon}(a|s) = \pi(a|s)$ with probability ϵ else random;
6: **end for**

2.4 Value Iteration

Similar to policy iteration, the value iteration algorithm also uses iterations to estimate the accurate values and then make decisions, based on the dynamic programming algorithm. Differently, value iteration focus on the state value under the optimal policy, rather than the co-optimization process of state value and policy.

Under the optimal policy, there are relations:

$$Q^*(a_t, s_t) = \sum_{s_{t+1}}(P(s_{t+1}|a_t, s_t) * (R(s_t, a_t, s_{t+1}) + \lambda * V^*(s_{t+1})))$$

$$\pi^*(s_t) = argmax_{a_t}(Q^*(a_t, s_t))$$

And the Bellman equation under the optimal policy can be derived as:

$$V^*_{t+1}(s) = max_{a_t}(\sum_{s_{t+1}}(P(s_{t+1}|a_t, s_t) * (R(s_t, a_t, s_{t+1}) + \lambda * V^*_t(s_{t+1}))))$$

So with dynamic programming, the estimation of state value can be computed, and then the optimal policy can be computed by:

$$\pi^*(s_t) = argmax_{a_t}(\sum_{s_{t+1}}(P(s_{t+1}|a_t, s_t) * (R(s_t, a_t, s_{t+1}) + \lambda * V^*(s_{t+1}))))$$

The value iteration algorithm is:

Algorithm 2 Value Iteration

Input: The environment of the game, current state s;
Output: $\pi(a|s)$; Initial state value $V(s)$;

1: **for** $i = 1$ to T **do**
2:
$$V^*_{t+1}(s) = max_{a_t}(\sum_{s_{t+1}}(P(s_{t+1}|a_t, s_t) * (R(s_t, a_t, s_{t+1}) + \lambda * V^*_t(s_{t+1}))))$$
3: **end for**

$$\pi^*(s_t) = argmax_{a_t}(\sum_{s_{t+1}}(P(s_{t+1}|a_t, s_t) * (R(s_t, a_t, s_{t+1}) + \lambda * V^*(s_{t+1}))))$$

2.5 Temporal-Difference Learning

In policy iteration algorithm, the state-action value Q is estimated by sample mean:

$$\hat{Q}(a, s) = \frac{1}{N}(\sum_{i=1}^{N}(G(l_i)))$$

. More minutely, we can derive that:

$$\hat{Q}^N(a,s) = \frac{1}{N}(\sum_{i=1}^{N}(G(l_i))) \tag{1}$$

$$= \hat{Q}^{N-1}(a,s) + \frac{1}{N}(G^N(a,s) - \hat{Q}^{N-1}(a,s)) \tag{2}$$

Where $G^N(a,s)$ represent the real total reward when do N times samplings. That means the mean value of N times sampling equals to the mean value of $N-1$ times plus the simulation error of $N-1$ times. Replace the factor $\frac{1}{N}$ by a constant α to prevent the decay of the importance of simulation error, or in other words, use slip average to replace mean. Then we have another estimation:

$$\hat{Q}^N(a,s) = \hat{Q}^{N-1}(a,s) + \alpha * (G^N(a,s) - \hat{Q}^{N-1}(a,s))$$

To make the estimation converge faster while the sample times N increasing, and to make the estimation error to be as small as possible, the error term should be minimized:

$$MinimumTarget = G^N(a,s) - \hat{Q}^{N-1}(a,s)$$

Using the conclusions derived before, the sample term in the minimum target can be estimate by the result of the samplings before, to simplify the sampling process, reduce the sampling times and so simplify the complexity of the algorithm.

$$G^N(a_t,s_t) = R(s_t,a_t,s_{t+1}) + \lambda * G^N(s_{t+1},a_{t+1}) \tag{3}$$

$$= R(s_t,a_t,s_{t+1}) + \lambda * G^N(s_{t+1},a_{t+1}) \tag{4}$$

$$= R(s_t,a_t,s_{t+1}) + \lambda * \hat{Q}^{N-1}(a_{t+1},s_{t+1}) \tag{5}$$

Then the iteration equation in policy iteration algorithm is changed into:

$$\hat{Q}^N(a_t,s_t) = \hat{Q}^{N-1}(a_t,s_t) + \alpha * (R(s_t,a_t,s_{t+1}) + \lambda * \hat{Q}^{N-1}(a_{t+1},s_{t+1}) - \hat{Q}^{N-1}(a_t,s_t))$$

That is the core idea of temporal-difference learning algorithm (TD). Compared with policy iteration, TD focuses on reducing the complexity of the sampling process, trying to make sampling times as small as possible. Although the approximating in TD would, of course, have bad influence on the iteration accuracy, the low-complexity makes it possible to do much more circles of iteration than policy iteration algorithm, leading to the compensation on estimation accuracy and with further more iteration steps, it is even possible to reach a higher accuracy.

The TD algorithm is showed in details below:

Algorithm 3 Temporal-Difference Learning

Input: The environment of the game, current state s;
Output: $\pi(a|s)$; Initial policy $\pi(a|s)$ and $Q(a,s)$;

for $i = 1$ to T **do**
  select a state s;
  select an action a by ϵ-greedy policy;
  while s is not one of the final states **do**
    Sample with state s and action a, get reward r, next state$s^{'}$;
    select an action $a^{'}$ by ϵ-greedy policy under the state $s^{'}$;
    $Q(a,s) = Q(a,s) + \alpha * (r + \lambda * Q(a^{'},s^{'}) - Q(a,s))$;
    $\pi^*(s) = argmax_a(Q(a,s))$;
    $s = s^{'}$; $a = a^{'}$;
  end while
end for

2.6 Q-Learning

If say TD algorithm is somehow the improvement version of policy iteration, then Q-learning algorithm can be regarded as an improved value iteration. With the core idea to reduce the sampling times, same to TD, Q-learning focus on estimating the $Q(s,a)$ under the optimal policy π^*.

Under the optimal policy, the sampling equation is:

$$\hat{Q^*}^N(a,s) = \hat{Q^*}^{N-1}(a,s) + \alpha * ((G^*)^N(a,s) - \hat{Q^*}^{N-1}(a,s))$$

Estimate the long-term reword with the sampling result before, similar to TD algorithm:

$$\hat{Q^*}^N(a,s) = \hat{Q^*}^{N-1}(a,s) + \alpha * ((G^*)^N(a,s) - \hat{Q^*}^{N-1}(a,s)) \tag{6}$$

$$= \hat{Q^*}^{N-1}(a,s) + \alpha * ((G^*)^N(a,s) - \hat{Q^*}^{N-1}(a,s)) \tag{7}$$

Then the core equation of Q-learning, based on the dynamic programming, is:

$$\hat{Q^*}^N(a_t,s_t) = \hat{Q^*}^{N-1}(a_t,s_t) + \alpha * (R(s_t,a_t,s_{t+1}) + \lambda * max_{a_{t+1}}(\hat{Q^*}^{N-1}(a_{t+1},s_{t+1})) - \hat{Q^*}^{N-1}(a_t,s_t))$$

The complete algorithm of Q-learning:

Algorithm 4 Q-learning

Input: The environment of the game, current state s;
Output: $\pi(a|s)$; Initial policy $\pi(a|s)$ and $Q(a,s)$;

for $i = 1$ to T **do**
  select a state s;
  while s is not one of the final states **do**
    select an action a by ϵ-greedy policy
    Sample with state s and action a, get reward r, next state $s^{'}$;
    $Q(a,s) = Q(a,s) + \alpha * (r + \lambda * max_{a'}(Q(a^{'}, s^{'})) - Q(a,s))$;
    $\pi^*(s) = argmax_a(Q(a,s))$;
    $s = s^{'}$;
  end while
end for $\pi(s) = argmax_a(Q(a,s))$

2.7 Deep Q-learning (DQN)

Deep neural network have been proved to have great computational ability to simulate mappings and progressions. It can be inferred that applying deep neural network into reinforcement learning would significantly improve the performance of agents, even simply regarding reinforcement problems (eg. Chess games) as a mapping from current states to actions to take and applying a deep CNN to simulate the classification function would have great performance (Liskowski et al., 2017). More commonly, deep neural networks are used as one part of the traditional dynamic programming-based algorithms like Q-learning, to take responsible to compute. Guaranteed by the computational power, Q-learning with deep neural networks improve the performance dramatically (Mnih et al., 2015).

In traditional algorithm of Q-learning, the important state-action value $Q(a,s)$, used to generate the best policy, is computed by the iteration method, using *Bellman equation* and Dynamic programming. While with the help of deep neural network, we can just rely on its power. So the computation of $Q(a,s)$ become simple, what we need is just clarifying which mapping we want our network to learn. So similar to that in Q-learning, we put state s into the network, and of course, we want the network learning to simulate Q, so we feed the labels:

$$Y_t = r_t + \lambda * max_{a'}(Q(s^{'}, a^{'}))$$

Attention, is $s^{'}$ is one of the final states, then Y_t should be just r_t.

While there are still something should be noticed: for a network, it is not a good phenomenon to have the training instances associating with each other. That would severely violate the assumption of i.i.d. and of course ruin the training. So the trick of experience replay, in which we introduce a experience pool to store the sampling result, is introduced. For every epoches, we sample once and get one sampling instance $(s, a, r, s^{'})$, and the store it into the pool, when training, we randomly pick a batch from the pool. By this method, the time-domain association within the states is broken down.

And we would introduce a time-delay target network, whose parameter would be updated to be equals to the evaluation network, only for every N epoches, to select $a^{'}$ and evaluate $max_{a'}(Q)$. That means the target network used to evaluate the optimal target is a time-delayed version of the main network. The reason to do so would be stated in *chapter "Double DQN"*.

2.8 Double DQN

When applying the deep Q-learning algorithm to perform reinforcement learning, a deep neural network is used to estimate the state-action value $Q(s, a)$, taking the place of dynamic programming. And the optimizing target helping neural network fit the distribution is used as:

$$Y_t^Q = R_{t+1} + \lambda * max_a(Q_{\hat{\theta}}(s_{t+1}, a))$$

The idea has been proved effective, especially when the trick of experience replay is introduced, which would dramatically improve the performance of the algorithm (Mnih et al., 2015).

While there are still details need to be reconsidered. In standard Q-learning or DQN algorithm, the maximum operator uses the same state-action value, both to select and evaluate an action, that would make the selected action to over fit the selected Q value. That would make the algorithm more likely to select overestimated values, leading to overoptimistic value estimation. It has been proved that the overoptimistic in large-scale problems, even deterministic, are very common and severe in practice (van Hasselt et al., 2015).

To prevent this, a general idea is to simply decouple the selection of Q-network to do selection and evaluation, and this is the core idea behind Double Q-learning (van Hasselt, 2010).

In the original Double Qlearning algorithm, two identical networks, or two different sets of parameter θ and $\theta^{'}$, are used to estimate the update targets and train the both networks. That is, when training the network θ, we still use the maximum of greedy policyś state-action value, while the action will be selected by network θ and estimated by $\theta^{'}$. And when update the parameters of $\theta^{'}$, we use $\theta^{'}$ to select the best decision of greedy policy and evaluate it by θ. By decoupling the selection network and evaluation network apart, the overestimation is somehow reduced. In addition, van Hasselt (2010) argued that noise in the environment can also leads to overestimation, and Double Q-learning algorithm is also effective towards that.

In practice, the time-delayed target network can serves as evaluation network. Although it is not completely identical from the Q-network, in fact for every N epochs, it will inherit the parameters of Q-network, the time delay between the two would serve well to help reduce the overestimation. And whatś more important, using target network to do estimation means there is no need to introduce another network, make it much more simple and similar to DQN. And it has been proved that using target network as a nature candidate to implement Double Q-learning resulting in more stable and reliable result. And D-DQN can find better policies and obtained state-of-the-art result on the Atari 2600 domain (van Hasselt et al., 2015).

2.9 Dueling architecture

The DQN and D-DQN algorithms regard all the possible states equally, computing the best state-action values of the states and use them to update the networkś parameters. But in practice, not all the states have the same value to be paid attention to. Take Othello game as an example, maybe for the first few steps, it does not matters much that which action to take. (Especially for the first step of the black player.) So we can somehow omit the decision of the first few steps. While when the game comes to middle, maybe only one little mistake would lead to dramatic change of the board. So different state should be take differently, and standard DQN or D-DQN or other bootstrapping algorithms are proved to be disabled to satisfy that demand (Wang et al., 2016).

To handle with the demand, a new architecture of the neural network, called Dueling network, is introduced by Google Deepmind in 2016. In the architecture, two outputs would be estimated by the network: the value of state $V(s)$ evaluating the expectation of reward when current state is s, and the advantage of action $a(s)$. More abstractly, $V(s)$ evaluates how good it is to be in the state s, the goal we want, the state-action value $Q(s,a)$ evaluates how wise it is to take action $a(s)$ when in state s, and the advantage subtracts the value of state out of $Q(s,a)$ to obtain an unbiased measurement of each action:

$$A^{\pi}(a,s) = Q^{\pi}(a,s) - V^{\pi}(s)$$
$$E_{a\ \pi(s)}(A^{\pi}(a,s)) = 0$$

With the $V(s)$ and $a(s,a)$ being estimated separately by the network, we can evaluate the value of each action by its advantage, and with the state value we can know how much attention we should pay towards the evaluation and selection of the actions. And that would make it possible to evaluate actions with pertinence.

Still, the final estimation target is $Q(s,a)$, the same as that in standard DQN network (Mnih et al., 2015), and the state value and action advantage should be added back. In Dueling architecture (Wang et al., 2016), the networks estimating $V(s)$ and $a(s,a)$ shared the feature extractor, the outputs from convolution layers are separated into two streams, one is sent to fully connected layers to compute $V(s)$, the other used by another fully connected layers to compute $a(s,a)$, and finally the two streams join together.

However, when back propagating, the equation cannot tell how to propagate the loss of $Q(a,s)$ to $V(s)$ and $A(a,s)$ clearly: a restriction is needed for the equation to do so. In the architecture (Wang et al., 2016), two restrictions are mentioned:

$$(A^{\pi}(a,s) - max_a(A(a,s))) + V^{\pi}(s) = Q^{\pi}(a,s)$$
$$(A^{\pi}(a,s) - mean_a(A(a,s))) + V^{\pi}(s) = Q^{\pi}(a,s)$$

The advantage of the best choice of actions is forced to be zero in the first method, and in two, the mean of advantages is forced to be zero. The former one is more theoretically interpretable because the best decision has zero advantage. As for the latter one, in fact the $V(s)$ is biased value of state, as well as advantage. Although lack in interpretation, it improves the stability because the bias only need to change with the mean of advantages, not with every instance of that.

Because the ports of the network are all the same with (Mnih et al., 2015), so Dueling architecture can be transferred to either DQN, D-DQN or any other algorithm, and improves the performance dramatically (Wang et al., 2016).

2.10 Monte Carlo Tree Search Algorithm(MCTS)

For further improvement, we choose to implement one famous algorithm, MCTS, which is used by Deepmind in their AlphaGoZero. We tried to implement the Monte Carlo Tree Search algorithm with neural network

We use the MCTS to improve the policy. MCTS can also take advantages of exploitation and exploration so that the agent can find the best policy without stopping learning when it finds one local optimum. MCTS use the tree structures. Each node in the graph represents a state from the point of view of the current player. Thus the state can be represented in a canonical form which means that the pieces of the current player at current state are 1 while -1 for the opponent player. The advantage of this kind of representation is that we can uniquely code our boards and easy to determine the reward. If an action can be taken so that the state can transfer from s_i to s_j, then the two nodes have an edge. Using these nodes and edges, we can store the information we need to represent the MCTS structures. We store $Q(s,a)$ which is the expected reward of edges (s,a) and $N(s,a)$ is the number of times we visit this edge. Besides, we use the CNN to estimate the probability of taking action an at state s. Then use these to calculate the upper confidence bound.

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

Where c is a hyper-parameter. This first term is the expected reward when following edge (s,a) which corresponds to the exploitation term. The second term is for exploration strategy. When

$P(s,a)$ is very small and $N(s,a)$ is more likely to be very small. But $\frac{\sqrt{\sum_b(N(s,b))}}{1+N(s,a)}$ is going to be big and accordingly, the edge is becoming more inviting for the agent. Besides the UCB and ϵ-greedy method are all for balancing exploration and exploitation. During training procedure, we choose actions that lead to the maximum UCB while during self-play we choose the actions based on the stochastic policy which is estimated form recursively calling simulation method.

The algorithm of MCTS is shown as below:

Algorithm 5 MCTS

Input: The environment of the game, current state s;
Output: v; Procedure $MCTS(s,\theta)$;
if s is terminal **then**
  return $game_result$;
end if
if s not in $Tree$ **then**
  $Tree$ append s;
  $Q(s,) = 0$;
  $N(s,) = 0$;
  $P(s,) = p_\theta(s)$;
  return $v_\theta(s)$;
else
  $a = argmax_{a'}(U(s,a^{'}))$;
  $s^{'} = GetNextState(s,a)$;
  $v = MCTS(s^{'})$;
  $Q(s,a) = \frac{v+N(s,a)*Q(s,a)}{1+N(s,a)}$;
  $N(s,a) = N(s,a)+1$;
  return s;
end if

First, we initialize the model neural networks. And set s as the root node, and calculate UCB. Then choose the action that has the maximum UCB values. If the next state $s^{'}$ is not visited, we add it to the tree and initial p and v which are evaluated by the neural network and return v. If the next state $s^{'}$ is the terminal state which means that the game ends at this state, we return the true value of the reward v. If the node $s^{'}$ has been visited, then we just choose the next state and update the values of the nodes and edges and then return v. In another word, when a node is a leaf node or a terminal node, we start back propagating v up till the root s. and then we choose next action and then set the next state as the root state.

The main idea behind MCTS is using $N(s,a)$ to estimate the policy $pi(s)$. And after training, the neural network can give use better estimation of the $p(s)$ and $v(s)$ by using the following loss function:

$$LL = \sum_t ((v_\theta(s_t) - z_t)^2 + \pi_t * ln(p_\theta(s_t)))$$

We can use the evaluated $pi(s)$ to guide our action choice as a result. And we build a deque which is very similar to the replay memory mentioned in the other part which stores (s, pi_s, r). The deque or the replay memory can efficiently reduce the dependency between each batch of examples. Therefore it can help the model have more generability and somehow prevent over fitting.

3 Experimental Methods and Tricks

While training, we applied several tricks to help the training and improve the performance of the agent, including *self-play*, *survival of the fittest*, *experience replay*, *representation of the game*.

3.1 Self-play

When training DQN we try to use the random opponent to train our model as first. And then use the self-play i.e. competing against itself and learning from self-play. Competing with random can somehow give us some confidence that our model has learned from a random player. Therefore, when doing self-play, the model may be less likely to over fit themselves which means that both of the players are becoming better and better as training proceeds. To further prevent this question, we introduce the next trick that is very useful.

3.2 Survival of the Fittest

After training, we let the old model which uses the previous parameters compete with the current model which uses the updated parameters. Both of them would play black and white. And we count the scores of them. If the current model can outperform the old one, then we keep the current model. Conversely, if the current model loses, then we keep the old model as the current model. Use this trick, we can guarantee that our model would make progress each time through self-play. After using this trick our DQN model can outperform the random opponent and can reach the 70 80% winning rates comparing with the old wining rates around 50 60%.

3.3 Experience Replay

We use the experience replay to store the previous training examples called experience. And during the training process, we randomly sample a batch of training examples to train our models. This trick can help model learn from experience. Hence, somehow it can help the model to be less likely to over fit the examples which are generated by the same playing process. Our DQN, DoubleDQN, DuelingDQN and DQN with MCTS have all used this thick.

3.4 Representation of the Game

a) Reward. In the DQN we have tried 2 different ways to represent reward. Strategy 1 is to use one, zero to give rewards, which means that when current player wins we give 1 while the player loses, we give 0. Strategy 2 is to use one and negative 1. Similarly, if wining, player gets one the opponent gets negative 1. We believe that strategy one is more aggressive. The player would prefer to find ways that would win since the only useful information can be received under the condition that the player wins. But the model may be too aggressive that it focus on finding ways to win that it over fit the training sets. While using strategy 2 would make the player focus both on playing and wining. Hence, we believe strategy 2 may be better. And we have tried 2 strategies when running DQN. The one using strategy 2 is slightly better than the one using strategy 1.

b) Canonical form which use 1 and -1 represents black and white and 0 otherwise. In the *reverse.py*, the board is presented by 3 channels representing pieces of black and white player and whether a state is occupied by a piece. When training DQN we use this form, but after reading the paper learning to play Othello without human knowledge. We find that the canonical form may be a better representation of the board since this form is more condense. Therefore, the neural network may extract the useful information more easily comparing with the one used in the reverse.py. But due to the limits of the time, we do not compare the DQN or MCTS with NN with and without canonical form. But we use each one strategy separately in 2 models. But we believe it is part of the reason that MCTS with NN can learn better.

c) Other tricks that may be powerful. Due to the symmetry of the Othello, we can use this to do data augmentation. We can use the mirror symmetry form, the rotated form and so on. But due to the limits of time, we do not implement this trick, but we believe this would definitely make our model more powerful.

4 Experiments

During training, we first train our model against random opponent and then we use our model to train against itself. For DQN case, we build two neural networks for the player 1 and player 2 with 2 convolution layers and 2 fully connected layers. The board represented by 3 channels. We use

the GTX1080Ti to train our model for 2 hours. The winning rates of agent when playing white and black against random opponent are 66.7% and 60% separately due to the fact that we use 2 sets of neural networks. As we can see, DQN can perform slightly better than the random baseline, but it is very hard to train since the real rewards can be given only when the agents are at the last state. And then, we must use this to reward to evaluate our policy or Q-value. We have used self-play and model selection tricks, the results can be improved to about 5 to 10 percent after running several times.

We next experimented Double DQN and Dueling architecture of DQN, while not get the expected results. We found that there was not significant improvement compared with standard DQN. Their winning rates against random opponents are around 60% and 70% and the losses are lower than 0.5. We think that it is because we failed to find the best architectures and hyper-parameters of the two models.

As the DQNs performance is not so good, we decide to implement MTCS with neural networks. Our goal is to implement a simpler version of AlphaGo Zero with 3 convolution layers with the filter size of $3 * 3$ and the channel depth 512, 512 and 256, and 2 fully connected layers with 1024 and 512 neurons. We use GTX1080Ti to train our model for 5 hours. And we use dropout and batch normalization. During training we would track the process. After each iteration, we would evaluate the performance of the agent. And fine tuning the hyper parameters manually, if the loss is no longer decreasing, we would try to increase or decrease the learning rate. Note that though the low loss cannot mean the model would perform good, but we believe that this fact means model is no longer learning anything which means the model is good enough or it is on the local minimum. So, we would test the model with random opponent and decide which situation our agent is in. The final performance of our agent is 87.5% competing with random agents. And when competing with alpha beta with 6 searching depth, my agents when playing black can outperform alpha beta while losing when playing white.

5 Summary

In that project, we discussed and analysised several reinforcement learning algorithms like DQN, Double DQN, Dueling DQN and MCTS with deep neural network, based on the understanding of the algorithms, we implemented several different agents to play Othello games and tested their performances. Finally we discussed their performance and came up with ideas to improve the training procedure.

References

[1] Hado van Hasselt, Arthur Guez & David Silver (2015) Deep Reinforcement Learning with Double Q-learning

[2] Shantanu Thakoor, Surag Nair & Megha Jhunjhunwala (1995) Learning to Play Othello Without Human Knowledge

[3] ZiyuWang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot & Nando de Freitas (2016) Dueling Network Architectures for Deep Reinforcement Learning

[4] Tom Schaul, John Quan, Ioannis Antonoglou & David Silver (2016) Prioritized Experience Replay

[5] Surag Nair (2018) A Simple Alpha(Go) Zero Toturial

[6] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez & Thomas Hubert Mastering the game of Go without human knowledge

[7] Michiel van der Ree & Marco Wiering Reinforcement Learning in the Game of Othello: Learning Against a Fixed Opponent and Learning from Self-Play

[8] Pawe Liskowski, Wojciech Ja skowski & Krzysztof Krawiec (2017) Learning to Play Othello with Deep Neural Networks