IERG4998H Final Year Project 2

Playing Chinese Checkers with Reinforcement Learning

BY
LI, Haocheng
1155047102

A FINAL YEAR PROJECT REPORT SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS
FOR THE DEGREE OF BACHELOR OF INFORMATION ENGINEERING
DEPARTMENT OF INFORMATION ENGINEERING THE CHINESE UNIVERSITY OF HONG KONG

2017 May

I. ABSTRACT

Our last term result was far from satisfactory due to two main reasons. One is that the function approximator we applied in the algorithm is too weak to response to thousands of training data and to precisely represent them in a strategy. The second one lies in the limited memory of agent computer. With the training process undergoing the database storing the board states grows unexpected large and at last we can only reserve few of them while abandon thousands of useful data. In this sense, more than 60 percent data is wasted. To address the above problem, we come up with two innovative ideas. First, we adopt a new heuristic searching algorithm against which the reinforcement learning agent plays. We aim to develop a strong non-learning rival agent which can reflectively elevates the learning efficiency of the learning agent as well as improve its performance. It's turned out that a heuristic searching agent is so powerful that it gains 80 percent winning rate against top players in the world and 99 percent winning rate against other skillful players. Also, some innovative techniques and algorithms for the chess move searching are brought about to accelerate the computing and optimize the performance. After optimization, the computing speed soars to hundred times higher than the original.

Another idea appears to be adopted by most of learning agent nowadays, that is to use neural network as the function approximator instead of direct storing state information which is also known as deep learning. However, it's not easy to measure the performance of the reinforcement learning when it is equipped with a non-linear function approximator such as neural network which is used to represent the action-value function. From our observation, the reasons lie in two main aspects. One is that in each training trajectory the slight change of Q value will totally be amplified and propagated through the whole network which eventually changes the weight distribution, as well as the correlation between Q value and the decision-making value $\mathrm{r} + \gamma max_{a'}\mathrm{Q}(s',\ a')$. To deal with the potential problem, a randomization mechanism called replay is addressed inspired from [1]. Plus, we impose an update on Q value towards target value periodically instead of every time which brings us benefits on reducing the correlation between them. Diverse types of deep reinforcement learning (DRN) have been tested and each has its own strength in either dealing with data or updating value. By comparing, a supervise neural network followed by a deep Q learning network wins out and we feed the multidimensional network with 10000 training trajectories and gain higher wining rate against human experts compared with searching agent.

Keywords—Nega Scout Searching, Heuristic, deep reinforcement learning

II. INTRODUCTION

Our previous work focused on different techniques in the reinforcement learning including Q-learning, Monte Carlo method and TD leaf etc. which were used in the project to teach a computer to learn how to play the Chinese checkers. The artificial intelligence can determine the actions and state transition by learning from the previous experience. The learning process is said to be greedy when the prediction only refers to the maximum reward while during our

training the AI can pick up a choice following a predefine distribution, which can largely improve the robustness of the system. As well, some other techniques like reward-guiding opponents were also introduced to speed up the training and cross validating. We split the algorithm into three divisions which are divisible by different location relationship of each side accordingly. The forward spreading approach is powerful enough to establish a minimax tree for both the first and the second divisions while the last stage adopts the backward propagation approach. We mainly concentrated on develop a model-free leaning mechanism to instruct the AI on action decision making for the first two stages. Also, Monte Carlo Tree Search and alpha-beta pruning are practiced as essential tools to store values of huge amount of tree states in the databases in the early trials. Computing analysis is conducted for the combined algorithm and we make sure the algorithm does not consume too much computing resources to perform normally in real practice. Although the program wins over 99% of the time against a greedy player, we are unsure if it's a challenge for a relatively good human player. Some alternative approaches aroused our interest and get fully researched which may have more impressive impact on balance of exploitation and exploration in tree search, algorithm convergence, etc. We will take a practice on those approaches in next term

However, here are significant limitations for the traditional reinforcement learning, for example, due to the restrained capacity of the current state database, a complete action taking instruction was not yet utilized and invoked during each iteration. What's worse was that we were unable to find a Chinese Checkers agent to generate master moves. Recall the multiple trials undertaken last term the master move we made was purely based on some simple greedy algorithms or UCT, slightly better. To overcome these deficiency, a deep learning model was inspected but we failed to implement it in details, which show the incentive for our continuing work this term. Basically, we do two things this term. First thing is to design a powerful heuristic agent to simulate the master moves while the second thing is to integrate the output master moves to a newly built neural network to refine the action choices. In practice, the outcome of the heuristic system will be stored in an ad hoc decision stack and used as the action space for the network. To be specific, we introduced the Nega scout searching for the current player each turn to explore the best moves with respect to our heuristic board value. To accelerate the searching process, a transportation table and a history heuristic table are put forward to remove the unnecessary moves. In the aspect of computing and evaluation, we inspect that our core language Python seems response slow in each training trajectory without optimization, we come out a supplementary mechanism which leverage some user-defined data structures and baseline algorithms and to improve the learning AI's performance.

III. PRIOR WORKS

In our last term final year project, we mainly focused on developing a decision-making mechanism with traditional reinforcement learning in two players case. The ultimate algorithm adopted SARSA learning plus the alpha-beta pruning to reduce the size of the minimax tree. The immediate rewards for each state was defined by ourselves including 10 attributes indicating the short-term goodness.

3.1 Value iteration

Since we cannot give a heuristic value for every point of the board, we used several functions to evaluate the move.

1. The cumulative distances from all pieces of player *i* to its diagonal corner
2. The maximum vertical advance for a single piece of player *i*
3. The horizontal variance of all pieces to the two sub-central lines
4. The vertical variance of all pieces of player *i*
5. The number of empty slots that player *i* can jump into.

Therefore, the evaluation function is

$$\tilde{V} = w_1(A_2 - A_1) + w_2(B_2 - B_1) + w_3(C_2 - C_1) + w_4(D_2 - D_1) + w_5(E_2 - E_1)$$

Where the weights $\boldsymbol{\omega} = (\omega_1, \omega_2, \omega_3, \omega_4, \omega_5)$ is trained by a function approximator. Then with the calculated value for each piece, we choose the highest six moves for the current player and only consider these moves for the game tree selection. After having a knowledge of the short-term reward, we need to define the utility as the total payoff:

$$\mathrm{E}[\mathrm{R}(s_1, a_1) + \gamma \mathrm{R}(s_2, a_2) + \gamma^2 \mathrm{R}(s_3, a_3) + \cdots]$$

Discount factor is introduced and we wish to traverse the positive value in the chain as soon as possible to keep the expectation large and this makes a fundamental for reinforcement learning.

3.2 SARSA learning

We divided the game flow into three phases as mentioned in Introduction section and the focus is addressed on the first two phases. More specifically, first phase is when each player can only jump through their own chess. Second phase is when the chess of two players encounter and can jump through others' chesses. Third phase is when the chesses of the two players separate and jump towards their home. The work in the first two phases is accomplished in three steps – Estimation of value function, backing up values along real or simulated trajectories, and generalize policy iteration using the approximate optimal policy and the approximate optimal value function to improve each other. Several schemes are suitable to maintain a powerful value function including Q-learning (TD (0)), Monte Carlo method (TD(1)), TD(λ) as well as TD-leaf. However, after testing we find Q-learning is too slow to converge and perform far from satisfactory given limited number of trajectories, say 800. Hereby, we mainly use SARSA(λ) (derivative of TD(λ)) for training where the parameter λ allows a state to learn from different combination of following value functions, efficiently avoid overfitting or overly bias after the parameter tuned. The other three learning algorithms are also researched and proposed to provide an alternative for SARSA(λ) and we will compare their performance next semester. The details are given in Temporal Differences Learning section. Then we applied the SARSA(λ) learning for each phase and tried to improve the total performances for the training agent. The details for SARSA(λ) is given below.

Algorithm 7 SARSA(λ)

```
function SARSA(λ)
    Initialize Q(s, a) randomly
    repeat(for each episode):
        e(s, a) = 0, for all s, a
        Initialize s, a
        repeat(for each step of episode):
            Take action a, observe R(s, a), s'
            Choose a' from s' using policy derived from Q (e.g. ε-greedy, UCT etc.)
            δ ← R(s, a) + γQ(s', a') − Q(s, a)
            e(s, a) ← e(s, a) + 1
            for all a do
                Q(s, a) ← Q(s, a) + αδe(s, a)
                e(s, a) ← γλe(s, a)
            end for
            s ← s', a ← a'
    Until s is terminated
end function
```

With the help of this algorithm, we can record the Q(utility) value of a good move and modify it in the following trainings. We also applied a combination of exploitation and exploration to find out possible good moves. As a result, after more than 800 games of trainings, the learning agent can defeat the simplest one-move look ahead agent.

Although the previous algorithm seemed work fine and fit well with a large sate-action pairs set, the training expense is a relatively high compared with traditional model-based artificial intelligence. In practice, we can only apply the first 6 moves with the highest utility as our searching space which is confined and error prone. Moreover, saving the potential moves to database requires many hard disk read and write. To deal with this problem, we design a record system which will record each move in memory and reduce the frequency to inquiry the database. The third problem arises when we find the probability for two moves to collide with each other is extremely small. In other word, it's extremely hard to match a previously recorded board transition during the preceding training. Also, it takes quite long time to compare the current transition with the recorded transition due to a lot of meaningless records are stored in the database. To avoid this happening, we make use of the hash function to exponentially decrease the searching time and increase the collision probability.

3.3 Improvement direction

We adopt the alpha-beta pruning to make a deeper look ahead mechanism. Instead of only looking ahead one layer for the best action which is considered as a slack strategy, we look ahead six layers and use tree search and tree pruning methods to locate the best move. Meanwhile, we modified the original algorithm and data structure to make a compensate of the system-level and the language-level deficiency. After developing, the agent seems to learn more than the case it only looks for more layer and the searching time is also reduced. One more observation is that the agent seems to always make a "good" move and will not learn any strange moves which probably is tentative good.

Another worry is that because we simplify our design for better readability and reliability, the agent seems behave exactly disappointing when competing with skillful human players which is due to the simple but poor original evaluation. This evaluation renders the agent to drop some good but intuitive moves but adopt more moderate one. A suggestion is given that we can create many scripts with different players online simultaneously and each of the script shares the same database and train it repeatedly. The advantage is that but in reality, we find it is awfully hard to implement script and interact with the online Chinese Checkers providers. More information is needed like encryption code, session number will be requested when we design such scripts. For example, the game platform of Tencent doesn't leave any interface for us to circumvent the graphic interface and only transmit data. More difficult lies in that most of online player is not strong enough and their poor move may affect our agent to learn something useless.

IV. METHODOLOGY

4.1 Data structure improvement

For a two-player game that can be simulated by a minimax tree, normally we need to look up the tree nodes and find the optimal move for the current player. There are three node type in the alpha beta tree search and each type has its own characteristics

4.1.1 Node Type

In the alpha-beta tree, all the nodes can be categorized into three types: [2] [3]

Type One: PV-node

The score within this node is exact and should be always larger than alpha and smaller than beta.

These nodes have the characteristic that

- *All root nodes and the leftmost nodes are principal variation node.*
- *During the searching, the nodes are defined as beta-alpha*
- *All siblings of a PV-node are expected to be Cut-node*

Type Two: Cut-node

This node is also called fail-high node, in which beta cutoff is performed. Its score is bounded but not exact.

Characteristics:

- *The children of a Cut-node are always All-mode*
- *The parent of a Cut-node is either a PV-node or an All-node*
- *The layer distance of a Cut-node to its PV-ancestor is always odd.*

Type Three: All-node

This node is also called fail-low node, whose scores are always lower than alpha. Its score is also bounded.

Characteristics:

- *The children of an All-node are Cut-node*

- *The parent of an All-node is But-node*
- *The layer distance of an All-node to its PV-ancestor is even*

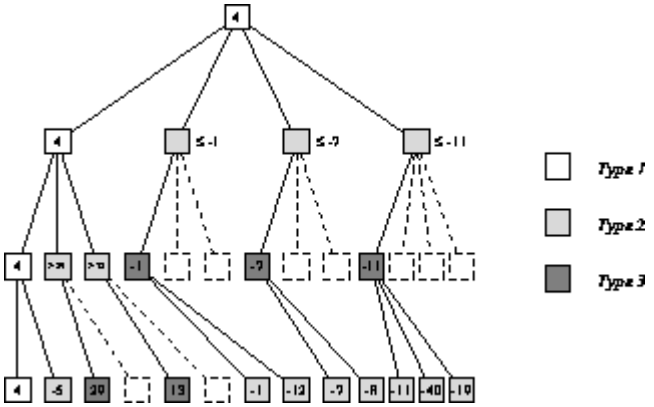


Fig 1. Node Types in a Minimax Tree [4]

4.1.2 Node Statistic

With the node categorization above, we can come up with some useful statistic data for the minimax tree analysis. One of the tree attribute we will use is about the odd and even layer. [5]

Odd-Even Effect

The even layer (start from 0) is the moves made by the current player, and the odd layer is the moves made by the opponent. Then, we can count the node and find out the equation below. Suppose the branching-factor is b and the depth is n, we have

For even depth, the leaf node number of the three types are

$$PV_n = 1$$
$$CUT_n = CUT_{n-1}$$
$$ALL_n = ALL_{n-1} + \ b^{\frac{n}{2}} - b^{\frac{n-2}{2}}$$

For the odd depth

$$PV_n = 1$$
$$CUT_n = CUT_{n-1} + \ b^{\frac{n+1}{2}} - b^{\frac{n-1}{2}}$$
$$ALL_n = ALL_{n-1}$$

Then we can come up with the following table that predicts the best situation of the beta-cutoff, assuming the branching factor of 40.

Depth	Worst case	Best case	PVn	CUTn	ALLn
0	1	1	1	0	0
1	40	40	1	39	0
2	1,600	79	1	39	39
3	64,000	1,639	1	1,599	39

4	2,560,000	3,199	1	1,599	1,599
5	102,400,000	65,599	1	63,999	1,599
6	4,096,000,000	127,999	1	63,999	63,999
7	163,840,000,000	2,623,999	1	2,559,999	63,999
8	6,553,600,000,000	5,119,999	1	2,559,999	2,559,999

4.1.3 Transportation Table

Transportation table is established when we have the following situations, in the process of searching, there is a chance that one chess position has already been explored while the agent does not know. Hence, it will search again. We build a table which caches the previous result of computation and next time the agent will search the table first before tracing the tree. Since comparing board position can also be time-consuming, we employ the hash-table to achieve this job. First, we randomly assign a 32-bit integer to each position of the board and a 64-bit integer to the value of the board. When we make moves, we use exclusive-or to compute the new position and the new key. Notice that comparing the current position with each cached position will only take constant time, the total searching time will be exponentially reduced.

4.1.4 Collision Handling

When using hash table, we need to pay extra attention on the collision, because the collision may direct the agent to move to an unexpected position. Here we may have the index collision(32-bit) and the key collision(64-bit). For 32-bit key, there can be 4.29×10^{9} number of combination and for 64-bit key, there can be 1.84×10^{19} combination. As for the Birthday Paradox, Robert Hyatt and Anthony Cozzie as published in their 2005 paper *Hash Collisions Effect*[7]. They discussed the effect of using the combination of 32-bit key and 64-bit key, and concluded that those signatures are more than sufficient. The probability of hash collision in such case is significantly small.

4.1.5 Nega Scout

In order to get a closer insight into the search process an example will be discussed. Figure 3 shows the smallest uniform game tree in which alpha-beta visits one node (p.2.1.2) which is exempted by Negascout.

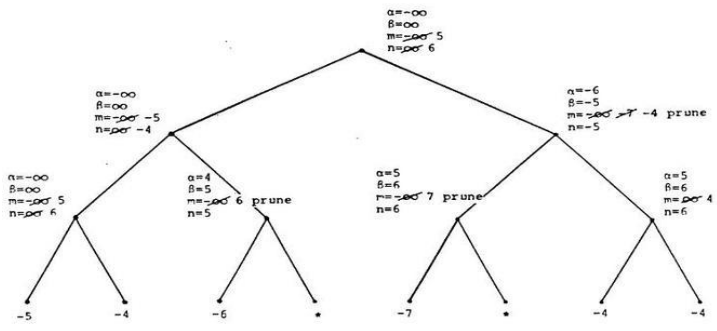


Figure 3.: Smallest uniform tree showing Negascout to advantage

Fig 2. Example of Nega Scout Search[8]

Instead of using the traditional alpha-beta searching mechanism, we used to enhanced version that is called the Nega Scout. It employs the Nega max algorithm, that is

$$\max(a, b) = -\min(-a, b)$$

As a result, we don't need to implement two separate sub-rountines to handle the alpha-player and the beta-player. Besides, this algorithm assumes that all horizon nodes would have the same score for the re-searching, and it employs another round of searching at the same depth with certain condition.[9]

Algorithm Nega Scout Search with Re-Search

1: **Input**: depth, alpha, beta of last layer
2: function
3: $t = -\text{negascout}(\text{depth} - 1, -\text{beta}, -\text{alpha})$
4: if (t > alpha and t < beta and it is not a PV-node)
5: $\text{alpha} = -\text{negascout}(\text{depth} - 1, -\text{beta}, -t)$
6: end function

To elaborate it, the Nega scout searches the first move with an open window, and then every move after that with a null window, which is used to reduce searching space. With the help of that, we can tell the alpha is already improved or not. While re-searching, it used the narrow window of (score, beta), instead of using the traditional window (alpha, beta).[8]

To be noted that, due to the quiescence search, the Nega scout algorithm is actually identical to the principal variation search with the fail-soft implementation.[10]

4.1.6 Fail-High Reductions

Actually, the re-searching in 3.2.2 is also called as fail-high, which is targeted as the reduction for the fail-high nodes(Type Two). As proposed and examined by Rainer Feldmann[11], and implemented in the program Zugzwang, its idea is to search to shallower depth at positions that are seemingly quiescent and where the side to move has established a substantial advantage, according to a static evaluation. The approach is applied at expected Fail-high or Cut-Nodes recursively inside a Nega Scout-framework. When combined with the transportation table, it can store the node types for further reduction. The details are as follows.

Algorithm Fail-High Reduction

1: **Input**: depth, alpha, beta of last layer, board position
2: **Output**: type
3: function
4: is_exact = False
5: $t = -\text{negascout}(\text{depth} - 1, -\text{beta}, -\text{alpha})$
6: if (t > alpha and t < beta and it is not a PV-node)
7: $\text{alpha} = -\text{negascout}(\text{depth} - 1, -\text{beta}, -t)$
8: is_exact = True

```
 9:            type = exact
10:     if (alpha < t)
11:            type = exact
12:            is_exact = True
13:     if (alpha >= beta)
14:            type = lower bound
15:     if (is_exact != True)
16:            type = upper bound
17: end function
```

4.1.7 Entry Storage

As discussed above, we can simply store the previous computed result of one board position, yet in the real application, it is not sufficient to merely store the value. We also store the depth, three nodes types and their score, since we can do further pruning as we meet the refutation move.[12]

The traditional technique to reduce the searching space is to use the alpha-beta pruning, which prunes off the minimax tree branches when we have found the best move within a layer so that the opponent will never make the parent node in the upper layer. Although tons of branches can be cut off during this process, it is still inefficient for the practical programming searching. As a result, we include the refutation move that further prunes off the branches in one layer.

The **refutation move** in the context of search algorithms like Alpha-Beta or Principal Variation Search is a move which fails high at Cut-nodes. It is not necessarily the best move, but good enough to refute opponent previous move. Since it is the best move found so far, in the context of transposition table, we can perform many operation of these refutation move. [8]

Algorithm Entry Return

```
 1: Input: depth, alpha, beta of last layer, hash key x, transportation table TT
 2: Output: score
 3: function
 4:     if (TT[(x,depth)]["type"] == exact)
 5:         return TT["score"]
 6:     else if (TT[(x,depth)]["type"] == lower bound && TT["score"] >= beta)
 7:         return TT["score"]
 8:     else if (TT[(x,depth)]["type"] == upper bound && TT["score"] <= alpha)
 9:         return TT["score"]
10:     end if
11: end function
```

4.1.8 History Heuristics

It is evident that, when the minimax tree is sorted, with the most optimal move arranged at the first, the alpha-beta cutoff will reach its best performance. Therefore, to sort the searching

sequence, we use the method of history heuristic table.

The history heuristic table works like the concept of iterative deepening. First, it performs some shallow pre-searching like 2 or 4 layer, which can be return very quickly. During these searching, the values of the table are updated. When a cutoff occurs either in shallow layer search or in the formal searching, we increment the counter of the history heuristic table, addressed by [from][to] [13]. And the increased value is typically 2 ^ depth. Then, after generating all moves for a given position, values retrieved from this table are assigned to these moves and used to sort them.

Algorithm History Heuristic Table Assignment

1: **Input**: from position, to position, depth, alpha, beta of last layer, history heuristic table HH
2: **Output**: history heuristic table
3: function
4: $t = -\text{negascout}(\text{depth} - 1, -\text{beta}, -\text{alpha})$
5: if (t > alpha and t < beta and it is not a PV-node)
6: $\text{alpha} = -\text{negascout}(\text{depth} - 1, -\text{beta}, -t)$
7: HH[from][to] += 2 ^ (depth + 1)
8: if (alpha >= beta)
9: HH[from][to] += 2 ^ (depth + 1)
10: end function

4.2 Optimization

4.2.1 Language level optimization

The running time reduces to a considerable extent after we apply the above algorithm. However, the system calling time and file I/O time still maintains long, which we think is probably due to the threading calling of Python. Recall that Python in a script language which means all the code will be converted directly to the machine code without compilation and it has more instructions than compile-level language like C++. Hence some compiling optimization is in vain for Python and we need to use techniques to reduce the programming redundant computation. Another issue is that, Python has such a relatively low address referencing mechanism that we always need to reduce the use of function, global variable and it will be very inefficient to use recursive in Python since it does not support cut-tail mechanism. Moreover, Python does not support short-circuit evaluation and the type binding is dynamic which makes it hard for error detection. Without the support from some powerful online source like Tensorflow or Keras, one of the methods we come up with is insert a piece of code from C in Python to resolve the above concerns.

4.2.2 Algorithm level optimization

Originally, we used the depth-first search to generate the reachable position space. In that case, a backward move may also appear along with forward move while it does us no good to make a backward move in a real practice. Hence, we come up with a new generating algorithm called upward generation with link list data structure which take this concern into account. Also, this algorithm resolve another problem is that the depth-first searching algorithm may generate the same state on different tree branches, which is a considerable time cost. While the upward algorithm regenerate a state after searching the existing state space to avoid redundancy.

We found out that the board can roughly be categorized into four zones: A, B, C, D where each zone represents an action field that for every stone that can perform a jump action, it can only jump into the same zone, say, one stone can only jump from one B position to the other B position. It's observed that none of these zones is inner connected.

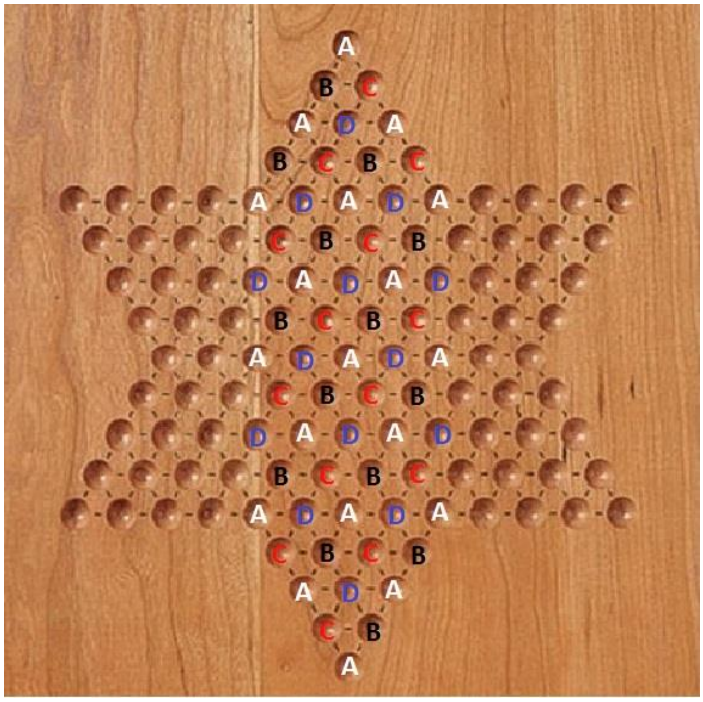


Fig 3. Board Attribute

Hence, for every round and searching layer, we first classify the jump zone for each stone and record their relative position. Then we can apply the upward generation algorithm to record the moving trace for each stone with the aid of linked list. More specifically, each time we follow the bound linked list of the stone until reach the nil and then we add a new position into this linked list if such position is encountered. After developing, the space for all jumpable position for one piece is completed and with the help of this algorithm, we can save duplicate search for many moves and preserve the tick-shape jump as well. After implementation, this method is roughly 15% faster than the original one. The algorithm is presented ass follows.

Algorithm Upwards Moves Searching

1: **Input**: four groups of positions from ten pieces of the current player
2: **Output**: unidirectional linked list(dictionary in Python) for each of the four group
3: function
4: **for** each group:
5: mark all pieces as unvisited

```
    initialize the stack
  while not all pieces have been visited:
      start from the most left behind unvisited piece and push it into the stack
    mark it as visited
    while the stack is not empty
        pop one position and check its six surrounding jumpable positions
        if the new position is empty
          add a link from the current to the new position, and push new into stack
        end if
        if the new position is occupied by current player
          mark it as visited and push it into stack
            if the current position is empty
                add a link from the new position to the current position
            end if
        else
            add a link from the current position to empty
      end if
end function
```

4.2.3 Hub-Channel Mode

First, we need to define the hub and the channel. For the blue chess in red ring it can jump into the red circle areas, which we call as channel of that chess. And the chess with star sign is called as hub, which enables other chess to jump through.

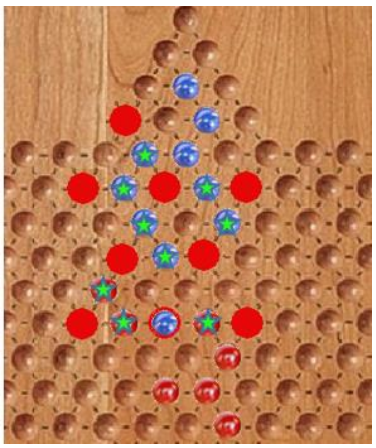


Fig 4. Channel-Hub Model

When one chess moves to a different zone, say from A zone to B zone, the hub and channel are totally changed so we need to re-generate the moves again. When one chess jumps, then its zone status remains the same, so that we only need to perform one operation for the channel and re-generate other three channels. To be specific, we can save the generation process in these two scenarios.

Suppose these black lines belong to channel A.

Scenario One

When the blue makes move, we can find that for that central channel, red cannot no longer

jump that far. But we don't need to re-generate all steps for that routine, and only clear the linked list for that point, and only take O (1) for that channel.

However, we need to re-generate the channels for other pieces because the new blue build a new hub for other three channel B, C, and D.

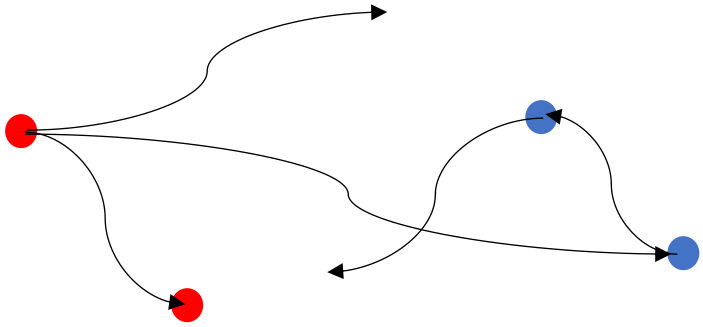


Fig 5. Before Blue Making Move

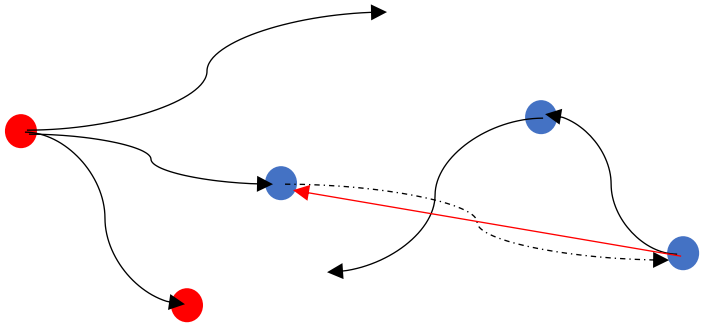


Fig 6. After Blue Making Move

Scenario Two

For this scenario, when the blue makes a move that is after the most left-over red piece, we can simply reuse the whole four red channels. To be noted that, when the blue chess keeps abreast with the last red chess, we cannot re-use the channel since the red can make tick move. After implementing these two scenarios, the average speed is faster about 10%.

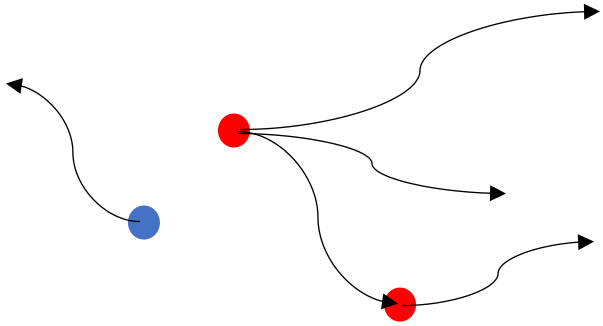


Fig 7. Before Blue Making Move

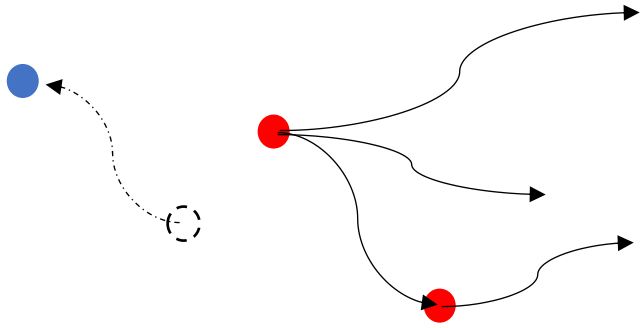


Fig 8. After Blue Making Move

5.1 Deep Reinforcement Learning

Most of deep reinforcement learning applied in gaming agent, big data center and even in the mechanical industry is deep Q network(DQN) which is the first deep reinforcement learning model and has high capacity of finding optimal or near optimal policy for complicated systems or agents. Based on deep Q network, people developed model-free control over extremely large dataset. The advantage of deep Q network over traditional multiple layers perceptron is that it can select policy and handle decision making problems which exactly meets the requirement of our cases. The purpose to load deep learning on the reinforcement learning is that the traditional one always runs into huge trouble when handling high dimension data, say in our case, a nine-by-nine binary matrix. With the development of feature detection using deep learning, such limitation vanishes with the combination of these two. Traditional reinforcement learning has two classes – policy iteration and value iteration. On the other hand, it can be divided into off-policy and on-policy where off-policy refers to updating the utility (target) with the optimal value, while on-policy permits the updating to go over current value. This distinction gives birth to various reinforcement learning algorithms like Q learning, SARSA learning, TD (λ) and so on. Accordingly, we can also combine deep learning with these basic models. To the best of our knowledge, different deep learning model has its own strengths and the selection may always base on the current situation. For example, deep Q learning has the drawbacks like low efficient sampling process as well as those limitations of off-policy reinforcement learning, while SARSA network has better tolerance during sampling. On the other hand, the convergence speed for deep Q network is seemingly faster than that of SARSA network

5.1.1 Deep Q learning

Recall the major equation for Q learning also known as Bellman equation.

$$Q^*(s,a) = E[r + \gamma max_{a'} Q^*(s',a')|s,a]$$

The basic idea behind the Bellman equation is to estimate the action given the state information as well as the binding reward. We have confidence to believe that the value update function $Q^*_{i+1}(s,a) = E[r + \gamma max_{a'} Q^*_i(s',a')|s,a]$derived from the Bellman equation will ultimately converges if the value iteration is adopted. However, this action-value function is estimated separately for each trajectory without any generalization. Instead, people tend to find a function approximator to estimate the action and transition which is typically linear function. With the aid of deep learning, the sophisticated but delicate non-linear function is introduced and the

target is to train the weights in Q-network. Like other neural network, the loss function instructs how the weight can change.

$$Loss(\theta_i) = E[(y_i - Q(s, a; \theta_i))^2]$$

Where y_i is the target value like the label in supervise network.

$$y_i = E[r + \gamma max_{a'} \mathrm{Q}(s', a'; \theta_{i-1})|s, a]$$

The weight θ_{i-1} is held fixed during the prediction while y_i depends on the network weight which means it will change and is different for each iteration which is mainly different from supervise network. We both use stochastic gradient descent (SGD) and batch gradient descent (BGD) to approach the optimal. When their performance is compared, BGD has better over sight for weight change and fast converge speed. On the other hand, the BGD's computational cost per iteration is proportional to the data set while the SGD has rather low constant cost per iteration. In our final design, we adopt SGD to release the computing power, which can earn us more time to train more data and gain better understanding of the whole game.

There are several ways of parameterizing Q using neural network. The architecture used in [14] includes only one input with the state representation and multiple output, the Q value and the corresponding action. The main advantage of this type of architecture is the ability to compute the target value within the whole state space with only a single forward pass through the network. However, we endorse another type of architecture that it takes both the state representation and the action as input and output the Q value, and then by comparing the Q value from different action, we pick up the largest one as our choice. The reason is that we believe more information is plugged into the system, the more robust the system will grow. To achieve the idea, we develop a strong heuristic algorithm to do initial selection of actions and input actions to the network chosen in such narrow range to avoid overflow of the data processing.

Noteworthily, the deep Q learning is the combination of reinforcement learning and deep learning but not modify the structure of Q learning. Thus, it is still a model-free learner without explicitly estimate the environment, and it is also an off-policy learning. Another point is in the replay memory. Learning directly from consecutive sample is inefficient, owing to strong correlations between samples. Unwanted feedback loops may arise and the weight may be stuck in local optimal if no randomization is introduced to action-transition selection. More details about the replay memory will be discussed in the following section.

The algorithm is presented as

Algorithm Deep Q Learning

1: **function** Deep Q Learning
2: initialize replay memory D with size of N and parameters of neural network
3: **for** episode=1 to M **do**
4: initialize state s_1
5: select a_1 using policy derived from Q (e.g. ε-greedy, UCT etc.)
6: **for** t=1 to T **do**
7: take action a_t, observe next state s_{t+1} and the reward r_t
8: store data (s_t, a_t, r_t, s_{t+1}) into D
9: sample data from memory D with some specific distribution (uniform, Poisson, Priority Sweeping)

10: select a' using policy derived from Q (e.g. ε-greedy, UCT etc.)
11: $y_j = \begin{cases} r_j \ if \ episode \ terminates \ at \ step \ j+1 \\ r_j + \ \gamma max_{a'} Q(s_{j+1}, a';\ \theta) \ otherwise \end{cases}$
12: Using stochastic gradient descent to optimize the loss function $L(\theta_i)$
13: $a_t \leftarrow a'$
14: **end for**
15: **end for**
16: **end function**

It turns out that the deep Q learning behave extraordinarily when small scale of low-dimensional input data. In our case, the input dimension is a 9-by 9 binary matrices which is under deep Q learning's capability. In such range, it can always gain higher performance and convergence speed than other deep reinforcement learning model.

5.1.2 Deep SARSA network

In SARSA learning, the training data is quintuple ($\mathrm{s, a, r}, s', a'$) rather than ($\mathrm{s, a}, argmax_{(s',a')} R$) which is adopted by Q learning. In every update process, this quintuple will be fetched from replay database and take effect in decision making sequence both for the current state and the proceeding state. The update major equation for SARSA learning is presented as follows.

$$\mathrm{Q(s,a)} \leftarrow \mathrm{Q(s,a)} + \ \alpha[r + \ \gamma \ Q(s', a') - Q(s, a)]$$

SARSA network estimates the current optimal state and action depending on

$$Q^*(\mathrm{s,a}) = E[r + \gamma Q(s', a')|s, a]$$

where a' is the action selected by one of our selection rules. Like deep Q learning, SARSA network leverages the loss function between ideal update value and the predicted target value.

$$\mathrm{Loss}(\theta_i) = (y_i - Q(s, a; \theta_i))^2$$

Here we adopt Minkovsky distance with factor 2 in the measurement. Of course, other types of measurement can be applied while the performance of 2-Minkovsky distance wins over after testing.

5.1.2.1 review on SARSA learning

The SARSA(λ) algorithm is an on-policy algorithm whose main difference from Q(λ) is that the new action a is selected following the policy which also determines the original action a. Hence, the reward used to update the state s is not necessarily be the max one among the proceeding states. In practice, Q(λ) can always learn the optimal path during the game play but also risk reaching a lousy solution while the SARSA(λ) may learn a 'safe' path because it takes the action selection method e.g. ε-greedy into consideration. As a result, SARSA(λ) gives out better average result than Q(λ). The algorithm is presented as follow: [7]

Algorithm SARSA(λ)

1: **function** SARSA(λ)
2: Initialize Q(s, a) randomly
3: **repeat** (for each episode):
4: e(s, a) = 0, for all s, a

5: Initialize s, a
6: **repeat** (for each step of episode):
7: Take action a, observe R(s, a), s
8: Choose a' from s' using policy derived from Q (e.g. ε-greedy, UCT etc.)
9: $\delta \leftarrow R(s, a) + \gamma\, Q(s', a') - Q(s, a)$
10: $e(s, a) \leftarrow e(s, a) + 1$
11: **for** all a **do**
12: $Q(s, a) \leftarrow Q(s, a) + \alpha\, \delta\, e(s, a)$
13: $e(s, a) \leftarrow \gamma\, \lambda\, e(s, a)$
14: **end for**
15: $s \leftarrow s'$, $a \leftarrow a'$
16: Until s is terminated
17: **end function**

5.1.2.2 SARSA network

The main objective of SARSA network is to minimize the loss function. We may use the gradient descent to approach the optimal weight and the differentiating is given below

$$\nabla L(\theta_i) = (r + \gamma\ Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))\nabla Q(s, a; \theta_i)$$

The gradient descent is presented as

$$\partial\theta_i = \ \alpha\nabla L(\theta_i) + \beta\theta_i$$

Where α is learning rate and the scale is set to be 1 out of 200 with respect of $\nabla L(\theta_i)$. β is the momentum to conserve the tendency of previous weight change. Specifically, we hereby use the stochastic gradient descent (SGD) to avoid the overflow of data processing. It's noted that during the whole training process and action selection is not greedy, hence the network can learn how to deal with diverse types of players with different strategies (Human, heuristic agent and so on) if the data for those strategies is imported. The algorithm is given by

---

Algorithm Deep SARSA Learning

1: **function** Deep SARSA Learning
2: initialize replay memory D with size of N and parameters of neural network
3: **for** episode=1 to M **do**
4: initialize state s_1
5: select a_1 using policy derived from Q (e.g. ε-greedy, UCT etc.)
6: **for** t=1 to T **do**
7: take action a_t, observe next state s_{t+1} and the reward r_t
8: store data (s_t, a_t, r_t, s_{t+1}) into D
9: sample data from memory D with some specific distribution (uniform, Poisson, Priority Sweeping)
10: select a' using policy derived from Q (e.g. ε-greedy, UCT etc.)
11: $y_j = \begin{cases} r_j \ \ if\ \ episode\ \ terminates\ \ at\ \ step\ \ j+1 \\ r_j + \ \gamma Q(s_{j+1}, a';\ \theta) \ \ otherwise \end{cases}$
12: Using stochastic gradient descent to optimize the loss function $L(\theta_i)$
13: $a_t \ \leftarrow \ a'$

14: **end for**
15: **end for**
16: **end function**

We absorb 10000 training trajectories into deep SARSA network training process while the result turns out far from satisfactory. In our guess, it may result from the limited size of training set. In practice, we usually need ten times more data when training a neural network compared with the traditional reinforcement learning. Confronted with the deficiency on computing power we do not rely on deep SARSA learning as our final design.

5.1.3 Combined model

Not only the deep Q learning but also the deep SARSA learning has a drawback that it requires massive storage and extraordinary computing power to process tens of thousands of data. The main effort we made in this part is to figure out a methodology to reduce this burden. Noted that after binding the replay memory to the deep reinforcement learning, the system not only needs to store the state information, but also requires separating a division to maintain the temporary tuples. We consider minimizing this replay experience storage by limiting the number of effective tuples.

First, we establish a supervise network to detect the goodness of tuples, or more precisely, to restrain the possible actions a state may lead to. The network works together with the heuristic algorithm introduced in the above sections. The input of the network is the state representation and the output is the board value which will be used by the heuristic algorithm. We endorse the soft-max function as the classifier to restrain the range for board value between 0 and 1. After each iteration a vector of decimal element of size n is obtained where n stands for all possible actions from a fixed state. The label of this network is the wining-losing status for the players on each side and the update can only be conducted until the end of one game. Hence, to make the training go through well we concatenate the result for each iteration as the overall prediction. The final prediction vector has the shape of $n \times k$ where k refers to the number of non-terminal states before the game ends. Accordingly, we extend the label to $n \times k$ by multiplying itself for several times where 1 indicates the wining status and 0 indicates losing.

The loss function is defined as

$$Loss(\theta_i) = E[||\boldsymbol{Y}_i - \boldsymbol{B}_i||^2]$$

Where $\boldsymbol{Y}_i$ is the label vector while $\boldsymbol{B}_i$ is the prediction vector.

In practice, we have tried double-side updating. double-side updating is a technique to accelerate the updating process. It means two players share the same network and database, and they will decide based on the same information. However, when one game ends, the label for one player is the counterpart for that of another player (0 converts to 1 and 1 converts to 0). Nevertheless, after training for hundreds of data, we find double-side updating in this scenario tends to overfit, hence we abandon this method and returns to one-side updating.

Similar to the above neural network, we use stochastic gradient descent to do approximation. The algorithm is given by

Algorithm Action Selection Network

1: **function** Action Selection Network

2: initialize the parameters of neural network
3: **for** episode=1 to M **do**
4: initialize state s_1
5: Compute the board value $\boldsymbol{B}_1$ for all $a_1 \in action\ space\ of\ s_1$
6: **for** t=1 to T **do**
7: take actions $a_t \in action\ space\ of\ s_t$ and compute the corresponding board value $\boldsymbol{B}_t$
8: store data $(s_t, a_t, \boldsymbol{B}_t)$ into a temporary storage
9: Concatenate $\boldsymbol{B}_t$ with $\boldsymbol{B}_{t-1}$
10: **end for**
11: Based on the length of $\boldsymbol{B}_T$ decide the label $\boldsymbol{Y}_i$
12: Using stochastic gradient descent to optimize the loss function $L(\theta_i)$
13: **end for**
14: **end function**

After the ad hoc selection on action space **A**, we pop out the first 6 optimal actions for each state into a new action space $\boldsymbol{A}'$. Next, we load the deep Q learning model with the action space $\boldsymbol{A}'$ to train the target value Q. It turns out the performance of this combined network is much better than when we only use a single deep reinforcement learning model. The result will be shared and discussed in next section.

5.2 Applicable techniques for deep reinforcement learning

In traditional reinforcement learning, the learning and updating should continue in sequence. In other world, each iteration can only update one trajectory in the minimax game tree. The updating process hence, is considered rather slow within a scalable learning environment. With the aid of deep learning, we wish to accelerate this process by introducing some innovative ideas like experience replay, repeated iteration and so on. Upon implementing these techniques, the efficiency of data usage is elevated. The weights are calibrated precisely and converge quickly and the computing and storing burden is greatly released as well.

5.2.2 Replay memory

No matter is the sequence representation of deep Q learning or the quadruple $(\mathrm{s}, \mathrm{a}, \mathrm{r}, s', a')$ of deep SARSA learning is stored inside a static memory which is called replay memory and we will access this memory in each iteration. During the training, we randomly fetch one sample from the replay memory with respect to some probability distribution, say uniform distribution, Poisson distribution (lay stress on most frequently used state) and so on. In our trial, we find the uniform sampling gains the best result which is probably because the two consecutive samples have less correlation and all possible state can be trained during the iteration. For example, if the maximizing action for one state is to jump to the left, then the distribution of the whole replay stack will tilt to the left oriented if Gaussian or Poisson distribution is adopted.

In practice, our algorithm only stores the last N (N=100) experience tuples because we make assumption that the average time of the tuples being trained before leaving the replay memory is over 10N and 1000 training experience for one state is thought to be enough. On the other hand, this rule can also release unnecessary consumption of computer memory. However,

the approach is in some sense limited since the memory buffer does not differentiate important transitions and always rewrite on recent transitions due to the finite memory size. We have researched more [17] complicated strategy which have the emphasis on which tuple we should learn more which is called priority sweeping algorithm. Below is a brief introduction of priority sweeping algorithm.

First, we need to build a maximum likelihood model of the state and q_{ij} is defined as

$$q_{ij} = \frac{Number\ of\ transition\ from\ i\ to\ j}{Occurence\ in\ state\ i}$$

Also, we define a terminal state k which indicates the end of game (in our case, the set of terminal states fall into two categories each including one side finishing the game) and non-terminal states as other states. Suppose a trajectory starting from a non-terminal state I, let π_{ik} be the probability that it ends at a terminal state k and $\hat{\pi}_{ik}[t]$ denote the estimated value of π_{ik} after the system has been running for t state-transition observations. Then a transition model can be established as

$$\pi_{1k} = q_{1k} + q_{11}\pi_{1k} + q_{12}\pi_{2k} + \cdots + q_{1S_{nt}}\pi_{S_{nt}k}$$
$$\pi_{2k} = q_{2k} + q_{21}\pi_{2k} + q_{22}\pi_{2k} + \cdots + q_{2S_{nt}}\pi_{S_{nt}k}$$
$$\cdots$$
$$\pi_{S_{nt}k} = q_{S_{nt}k} + q_{S_{nt}1}\pi_{1k} + q_{S_{nt}2}\pi_{2k} + \cdots + q_{S_{nt}S_{nt}}\pi_{S_{nt}k}$$

where S_{nt} is the number of non-terminals. We attempt to approach the probabilities for each state transition as precise as possible in a dynamic manner and here is the major equation.

$$\hat{\pi}_{ik}[t+1] = \hat{q}_{ik} + \Sigma\hat{q}_{ij}\hat{\pi}_{ij}[t+1]$$

Where state j is a successor of state i.

The priority sweeping algorithm performs once after every real-world observation in a stochastic behavior and in this sense, it can be considered as a heuristic algorithm with almost no randomization. Plus, It can be proved that the solution converges.

Algorithm Priority Sweeping

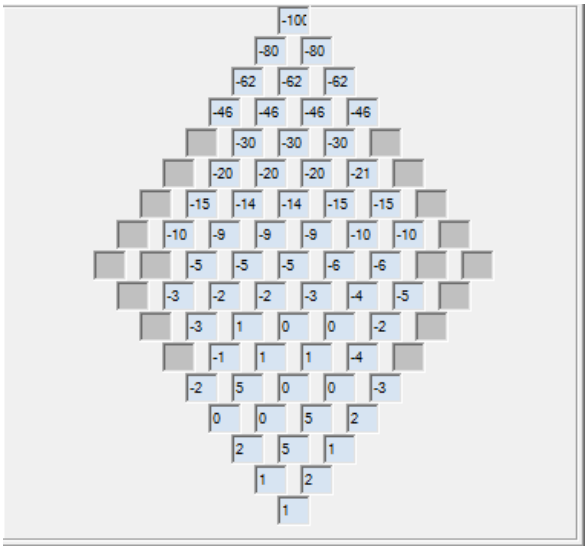
```
1: function Priority Sweeping
2:     for each non-terminal state I
3:         for each terminal state k:
4:         π_ik := π̂_ik[t]
5:     repeat
6:         Define the precision Δ_max := 0
7:         for each non-terminal state i
8:                 for each terminal state k:
9:                 π_new = q̂_ik + Σq̂_ij π_ik
10:                Δ := |π_new − π_ik|
11:                π_ik := π_new
12:                Δ_max := max (Δ_max, Δ)
13:            end for
14:            end for
15:    until Δ_max < ε
16:    for each non-terminal state I
17:        for each terminal state k:
```

18: $\pi_{ik} := \hat{\pi}_{ik}[t]$
19: **end function**

By applying the priority sweeping algorithm, the prediction for distribution of state transition normally provides excellent result. However, this algorithm requires massive computing power to store the probabilities for each transition and when a most frequent observation is encountered, especially when it has never been involved in previous iteration, more than S_{nt} extra iterations are needed for convergence. Other estimation algorithms seem all applicable in this case, like Bayesian network, Gauss-Seidel algorithm but a common characteristic for them is they are all computing expensive. After several attempts, we eventually use uniform mechanism to select entries from the replay memory which can still achieve some good result.

V. RESULT

We design three types of moves generation agents which look ahead 2, 3 and 4 layers per step



respectively and we assume that the more layers the agent look ahead the better the action it will take. Even for the same type of agent it can adopt different direction policies and the case will happen that the expected value for different policies is the same. Therefore, we also have two sub-type of agent to tackle this situation disparately --- the fixed type only take the same move while the randomized one will take moves with the same reward following some distribution each time. Notice that our searching algorithm relies on the board value which can be treated as modification weight for the neural network. The change of the board value will affect the action making. In the real-life training, we intend to optimize this board game by train it in a supervise network. It's also interesting to notice we can potentially change the strategy accepted by the agent by changing the manipulation of the board value. In other words, we can adjust the board value to make the agent more offensive or more defensive. The following is an example of the board game.

Fig 9. Board value

5.1 deep Q learning

The network structure for our deep Q learning is inherited from the master deep Q learning. The input to the network is a nine-by-nine binary state matrix with 2 standing for stone of the training side, 1 for stones from the opposite side and 0 for nil. The whole network consists of 3 hidden layers and 2 nonlinear layers with the last one to be fully connected linear layer. The first one is a convolutional layer with a kernel size four-by-four with stride 2 and the second layer is also a convolutional layer with kernel size of two-by-two with stride 2. One rectifier nonlinear layer strictly follows each convolutional layer. The outcome form the final fully connected layer with six outputs represent our legal actions. The following is the training result with x-axis as the training times and y-axis represents the wining status and the Q(target) value.

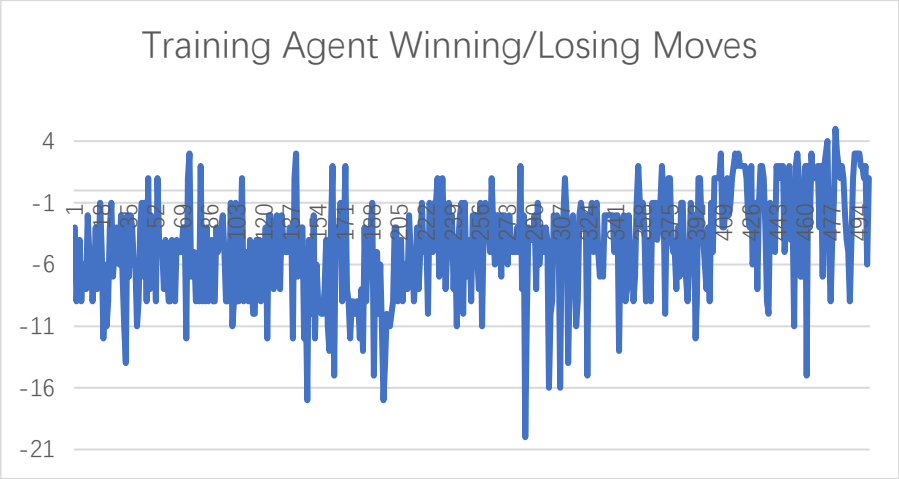


Fig 10. Wining status against training rounds

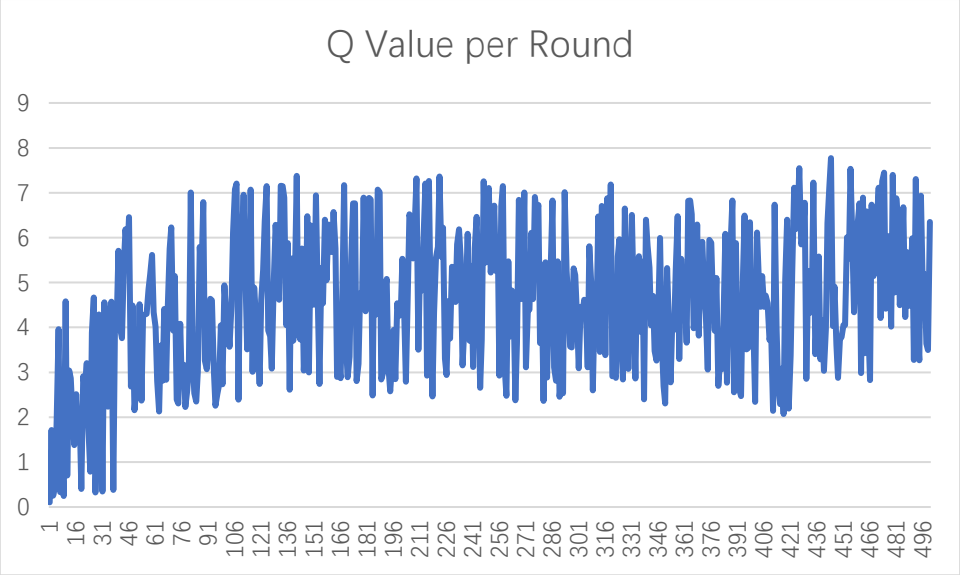


Fig 11. Q value against training rounds

We notice that the deep Q learning requires a highly stable environment, say the training target should adopt a specific strategy. While in rea-life this condition is not promised and can be variable with different counter policy. On the other hand, as the action generation is also unpredictable for the Chinese Checkers, we cannot simply move the stone arbitrarily or to a fixed location. To meet the requirement of deep Q learning, we make our agent to select the actions with a fixed policy as well as in a predetermined cation pool. The pool contains actions with the first 6 highest target score. From the above chart we can observe, the learning agent can gradually improve its performance and beat the agent who adopt the model-based policy. We use the wining steps to indicate the learning outcome. However, this method lacks of flexibility that if we have another opponent who adopts totally different policy, then the performance of this learning agent shrinks down quickly. This is probably due to the limited training resource we have right now. Without more general players and situations, the agent cannot develop further.

5.2 Deep SARSA network

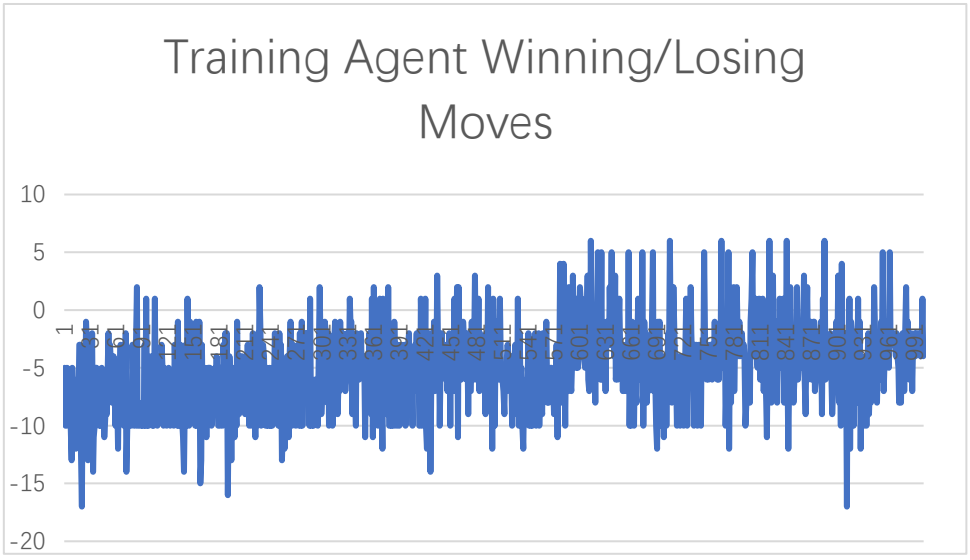


Fig 12. Wining status against training rounds

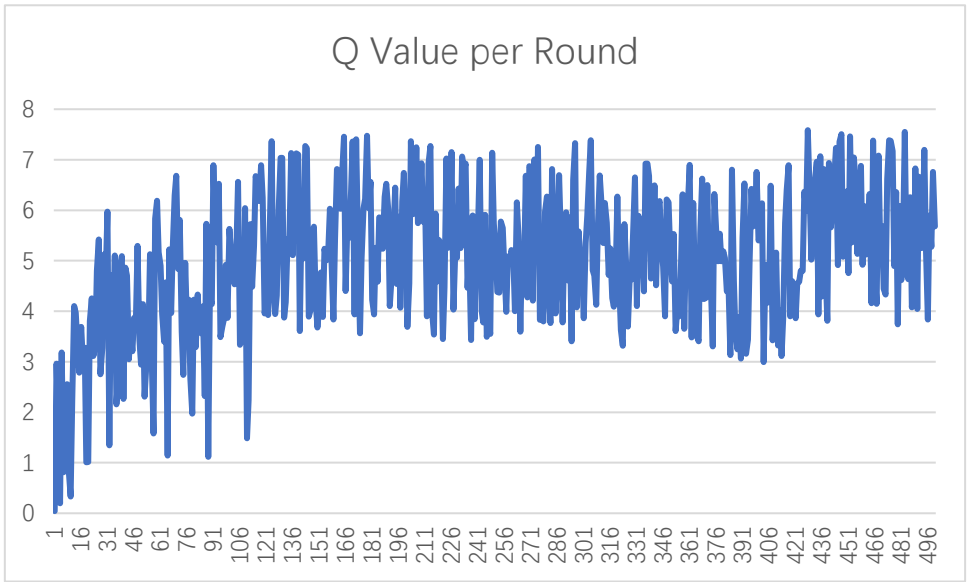


Fig 13. Q value against training rounds

Deep SARSA network has similar performance with deep Q network.

5.3 Combined network

We apply two hidden layers in the supervise network which is aimed to train the pre-processing action space. The two layers both have 729 fully connected nodes and each is followed by a rectified nonlinearity. Because the convolution layer is exempted from the network, hence, no feature map will be involved in which is considered to be advantageous to our design. As we observe, the network with the convolutional layer can learn fast as ad hoc stage but cannot reach a good level of accuracy or in other words some good single move will be ignored by the convolutional layer. Hence, the design without convolutional layer indeed help the robustness of the network. Another characteristic for this network is that we apply normalization for each hidden layer output to prevent the overfitting problem. The following is the visualized result to show the improvement the normalization brings to us.

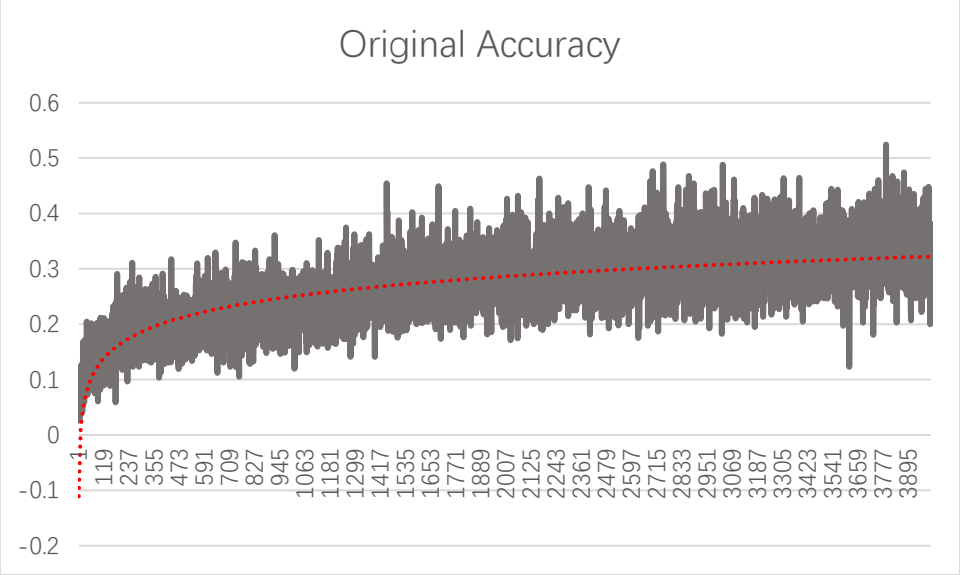


Fig 14. Original Accuracy

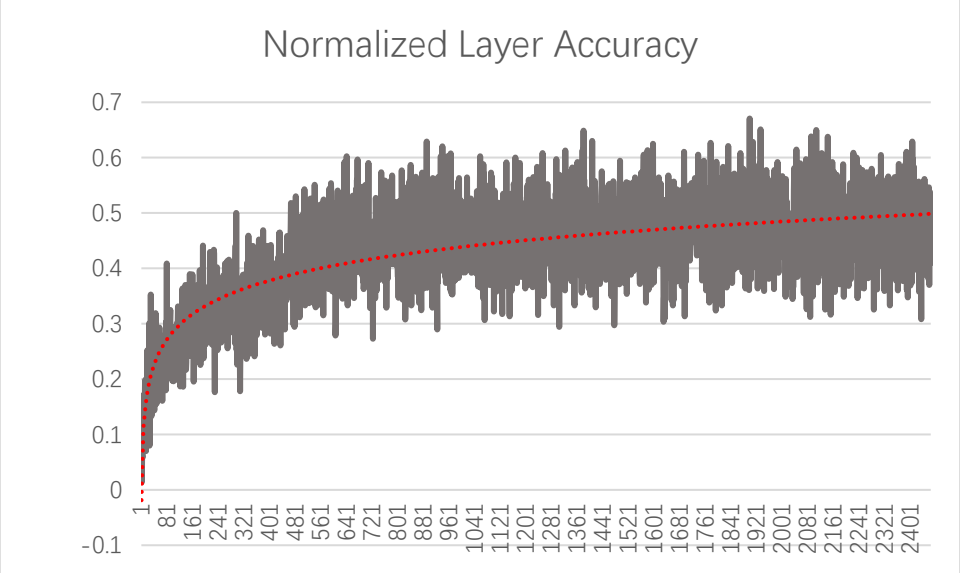


Fig 15. Accuracy after normalization

The last improvement lies in that originally, we assign 2 as the state value for the training side, while 1 as the state value for the opposite side. After modifying the weight, we change the pair to be (1, -1) where 1 for training side while -1 for the opposite side and the outcome is improved and can be observed from Fig 16 and Fig 17.

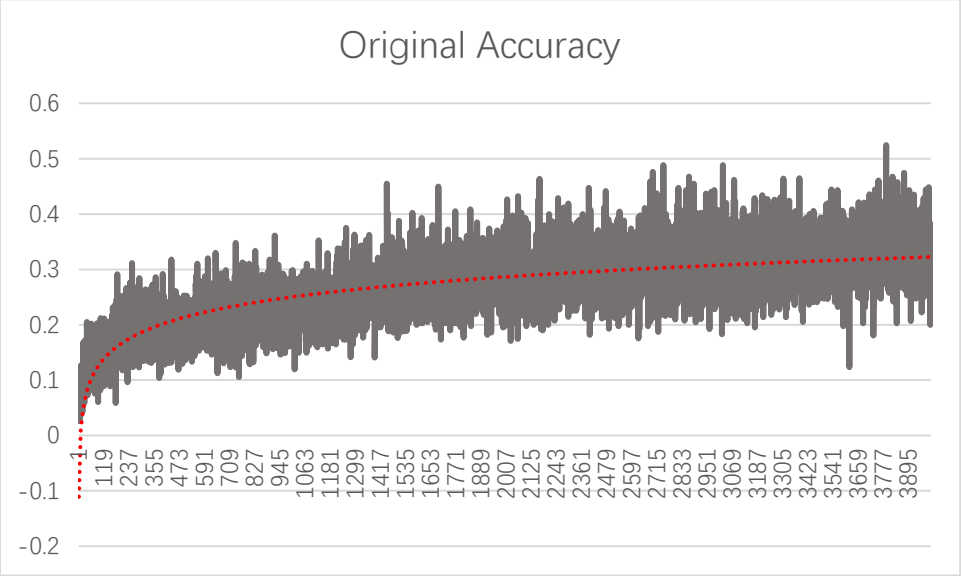


Fig 16. Original Accuracy

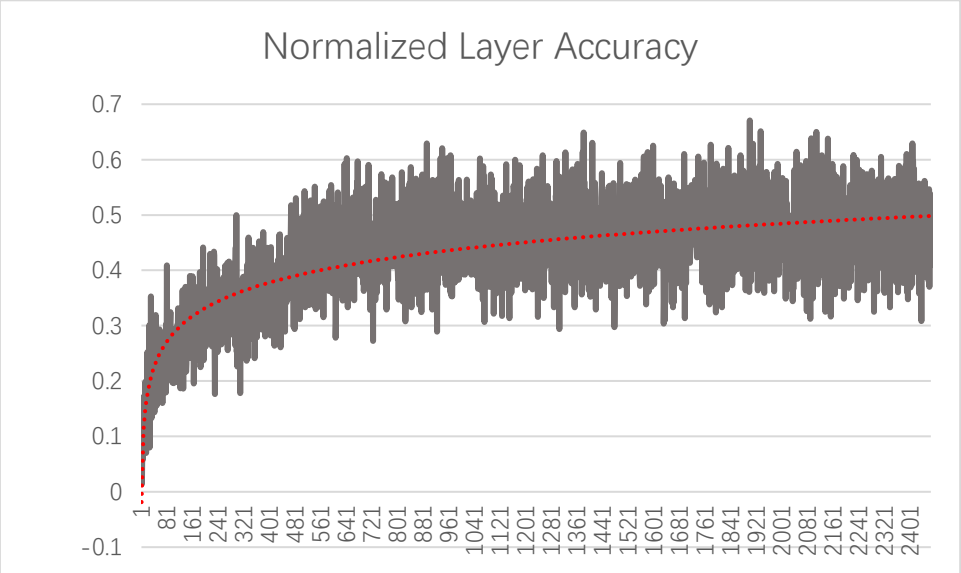


Fig 17. Accuracy after normalization

6 FUTURE DIRECTION

Because of limited knowledge of how to design a robust and powerful neural network, our learning network is still very weak to simulate a heuristic searching agent. Even in some time the network may not converge in finite steps. Recall the method we used last term, we divide the whole learning process into three phases which can give us a clear representation of the game. If we divide the network in the same way and only train the learning strategy in the first two phases there is a chance that the outcome can be better. Also, 4 layers look ahead is not strong enough compared with 6 and 8 layers look ahead, which limit the performance of our network. However, applying higher level look ahead strategy is not easy to implement due to our restrained computing power. We may find an optimization method to reduce the time and storage complexity taken by those higher levels look ahead strategy.

Furthermore, we can also try to dive into the multi-player game, that is three or six players play at the same time. In this case, the previous algorithm for two players is not applicable any more since it requires a lot deeper searching depth and a complicated value board but our computing resources is limited. For this case, we can employ back our Q-learning technique or using a heuristic strategy can be helpful.

7 ACKNOWLEDGEMENT

We are grateful to professor Lau Wing-Cheong for his guide in providing us feasible solutions to train the neural network especially with large scale database. Also, we're appreciated the help from AlphaGo designer from Google DeepMind for their previous results of applying deep reinforcement learning method on the board game.

REFERENCE

[1] M. Martin, "Reinforcement learning monte carlo and td learning", *LEARNING IN AGENTS AND MULTIAGENTS SYSTEMS,* 2014

[2] M. Winands, J. Herik, J. Uiterwijk, E. Werf, "Enhanced Forward Pruning", 2003

[3] T. Marsland, F. Popowich, "Parallel Game-Tree Search", *IEEE Transaactions on Pattern Analysis and Machine Intelligence*, Vol. 7, No. 4, pp. 442-452, 1985

[4] "Analysis of Alpha-Beta Pruning", *Parallel Computing Works*

[5] D. Shawul, "CUT/ALL nodes ratio", *CCC,* 2013

[6] R. Greenblatt, D. Eastlake, S. Crockerm, *The Greenblatt Chess Program*. Proceedings of the AfiPs Fall Joint Computer Conference, Vol. 31, pp. 801-810. Reprinted , **1988**

[7] R. Hyatt, A. Cozzie, *The Effect of Hash Signature Collisions in a Chess Program*. ICGA Journal, Vol. 28, No. 3, 2005

[8] A. Reinefeld, "Smallest uniform Tree showing Negascout to Advatage", *An Improvement to*

the Scout Tree-Search Algorithm. ICCA Journal, Vol. 6, No. 4, **1983**

[9] W. Smith, *Fixed Point for Negamaxing Probability Distributions on Regular Trees,* NEC Research Institute, 1992

[10] D. Breuker, *Ph.D. thesis: Memory versus Search in Games, 1998*

[11] Rainer Feldmann, *Fail-High Reductions,* Advances in Computer Chess 8, 1997

[12] D. Breuker, J. Uiterwijk, J. Herik, *Information in Transposition Tables*. Advances in Computer Chess 8, 1997

[13] J. Uiterwijk, *Memory Efficiency in some Heuristics*. ICCA Journal, Vol. 15, No. 2, 1992