IERG4998H Final Year Project I

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

December, 2016

Contents

1 ABSTRACT

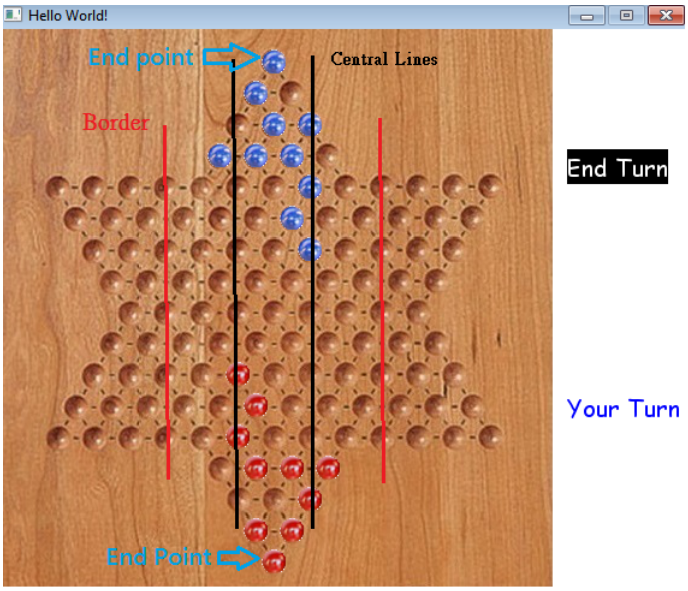

Figure 1: User Interface

This report describes different techniques in the reinforcement learning including Q-learning, Monte Carlo method and TD-leaf etc. which were used in this project to teach a computer to learn how to play the Chinese checkers. Experiments with different kinds of players have been done i.e. greedy and reward-guiding opponents are introduced to train the learning algorithm, on the other hand, the AI repeatedly improves itself by self-playing strategy. The algorithm assembles three sub-algorithms designed for three playing stages respectively. The forward spreading approach is applied for the first two stages while the last stage adopts the backward propagation approach. Both model-based and model-free reinforcement learning are implemented in different stages. Figure 1 shows the user interface of our program where players can select which step to take, rolling or hopping..

Monte Carlo Tree Search and alpha-beta pruning, as good ways to reduce the tree size, are practiced as assist to store values of huge amount of tree states in the databases in the early trials. However, the performance turns out below our expectation. We ultimately abandoned these two methods and instead stored limited number of states which reveals better results. The value of each board state is determined via a minimax tree of depth k, while the value of each leaf is approximated by weights and features extracted from the board. Weights are tuned via function approximation. 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.

1.1 keywords

Reinforcement Learning, MDP, Minimax tree, Chinese Checkers

2 INTRODUCTION

2.1 Reinforcement Learning

Reinforcement learning is an area of machine learning which provides a group of typical algorithms with a reward function, which indicates to the learning agent when it is doing well or poorly. The environment of reinforcement learning is typically formatted as a Markov Decision Process (MDP) which has large range applications in the instant decision making problem. The Chinese Checkers is played by the rule that two players move alternatively and previous actions has little impact on the latter actions take by the same player. The positions of all pieces after each draw can be considered as one state and every draw is an action. Hence, the two players' interaction can be formatted as a MDP.

On the other hand, the reinforcement learning agent can learn without expert supervision which render it suitable to solve some kinds of complex problems where there appears to be no pragrammable solution like Game playing. Because in game playing determining the best move often depends on a number of different factors, and like in Chess, Go or Chinese Checkers the number of possible board states is usually very large. To accurately figure out next step may consume an excess of computing power like Deep Blue chess computer(2006), exhaustively exploiting four layers from each state. However, the reinforcement learning cuts out the need to manually specify the rules between two states but allows the agent to learn from previous experience. The agents can be trained when playing with different opponents. The main concern of reinforcement learning is to maximize some notations of cumulative rewards where the instant reward can be gained before playing. For the Chinese Checkers, we define 5 instant reward function with weights which can be tuned during training. The problem has usually been studied in the field of optimal control and the problem we needs to deal with in this project is to solve bellman equation by stochastic gradient descent.

This project is accomplished by two people, LI, Haocheng and Chen Ken . The details of work division are given in the WORK DIVISION section.

2.2 Markov Decision Process

Markov decision process is a tuple $(S, A, \{P_{sa}\}, R)$, where:

- S is set of states.
  i.e. in the Chinese Checkers, S refers to all possible situations where each of them corresponds with a group of positions all on-board pieces occupy.
- A is a set of actions.
  i.e. The set of possible moves one player may choose from a state.
- $\{P_{sa}\}$ are the stae transition probabilities.
  For each state $s \in S$ and action $a \in A$, $\{P_{sa}\}$ is a distribution over the state space which indicates the probability from one state s to another state s' via action a. While in the game setting, the probability is deterministic which means this attribute can be igonored in the following computation.

- $\gamma \in [0, 1)$ is the discount factor.
  which reduces the impact of long term reward to the current state.

- $R : S \times A \mapsto \mathbb{R}$ is the reward function.
  Here we select the following ten reward functions for the Chinese Checkers:

  1. the cumulative distances from all pieces of player i to the 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 of player i.
  4. the vertical variance of all pieces of player i.
  5. the number of bridges player i has. (a bridge refers to a situation that one piece of player i can hop(jump over the pieces))

  with $i = 1, 2$. The first two reward functions are extracted from [16] which has been testified by He, Hu & Yin as good parameters and the third one was an improvement from one in [1] and we select the two 'sub-central' lines (the lines besides the central line)(Figure 1)instead of the central line as our baselines because we wish to search for more possible 'good draws' by relaxing our restriction on the dispersion. Plus, the border lines are drawn such that each draw is limited in a certain region. By doing this, the AI is not likely to learn from the bad move(one out of the region). The last two are developed by ourselves to measure the profit each player can gain after the opponents' turn. The square of the distance is used in the 4^{th} and 5^{th} reward function to penalize the trailing pieces, which aims to galvanize each player to make the piece cohesively and promote hopping. The ten reward functions are picked from a large number of set such that the overfitting of the model is avoided.
  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 $\boldsymbol{w} = (w_1, w_2, w_3, w_4, w_5)$ is trained by a function approximator.

The entire MDP proceeds like

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} s4 \xrightarrow{a_4} ...$$

where s_i refers to a state. The long term reward function for state s_1 is given by the following with the discount

$$R(s_1, a_1) + \gamma R(s_2, a_2) + \gamma^2 R(s_3, a_3) + ...$$

Our goal in this project is to choose actions over time in order to maximize the expected value of the total payoff:

$$E[R(s_1, a_1) + \gamma R(s_2, a_2) + \gamma^2 R(s_3, a_3) + ...]$$

Note that the discount factor is introduced and we wish to traverse the positive value in the chain as soon as possible to keep the expectation large. A policy is a strategy π that one player adopts such that given a specific state s the player can decide which

action to take according to π. Given a fixed policy π, its value function V^π satisfies the Bellman equations:

$$V^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s')} V^\pi(s', a')$$

Suppose the best strategy is π^* then the Bellman equation with the optimal value function is:

$$V^*(s,a) = R(s,a) + \gamma \max_{a' \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{s\pi^*} V^*(s', a')$$

with

$$\pi^*(s) = arg \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{s\pi(s')} V^*(s')$$

A good way to approach the optimal value and policy V^* and π^* for s_0 is by iteration that find the optimal V given a policy π and find a optimal π given a value V

$$V(s_0,a) \leftarrow V(s_0,a) + R(s_0,a) + \gamma \max_{a' \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P_{s\pi} V^*(s', a')$$

which is the fundamental assignment function in Q-learning.[2]

2.2.1 Value Iteration

the value iteration of RL in Chinese Checkers is given by Algorithm 1 where $P_{sa}(s) \equiv 1$ in accordence with the game setting.

Algorithm 1 Value Iteration

1: **function** VI($R(s,a), \gamma$) ▷ Where $R(s,a)$ - reward for state s, γ - discount
2: For each state s except the final state, initialize $V(s,a) := 0$
3: **while** not convergent **do**
4: **for** $\forall s$ in $\mathcal{S}$ **do**
5: update $V(s,a) := R(s,a) + \max_{a' \in \mathcal{A}} \gamma V(s', a')$
6: **end for**
7: **end while**
8: **end function**

The current value function is repeatedly updated as different policies are chosen. In practice 800 trajectories can guarantee the error between the trained value V and the optimal value V^* smaller than a threshold 0.1. Also there is another convergent algorithm - policy iteration given in algorithm 2 which can also provide an approximation of V^* and π^*, while the inner-loop function for policy iteration is not linear. Providing that the Chinese Checkers has a large states space, solving the nonlinear equation system in each iteration may deplete huge amount of computing resources and sharply increase the time complexity by O(the number of states following one trajectory). As a result, we use the value iteration as our core in all the reinforcement learning algorithm.

2.2.2 Learning algorithm

Now we can learn the model of RL from MDPs and use the data learned from MDP to train the value function and policy. The model for each state including possible actions,

Algorithm 2 Policy Iteration

function PI($V(s,a),\gamma$) ▷ Where $V(s,a)$ - value for state s, γ - discount
  while not convergent **do**
    assign $V := V^{\pi}$
    for $\forall$ s in $\mathcal{S}$ **do**
      $\pi(s) := arg\max_{a in \mathcal{A}} V(s',a')$
    end for
  end while
end function

the transition probability (in our case is always 1) and the rewards are first initialized. During the procedure, we can update the value function of state s each time we explore it with learning rate α which will be specified in the temporal different learning section. Note that, the more experience we gained the better the algorithm performance is, which indicates that our algorithm will monotonously improve as more and more games are played. The general algorithm is given as bellows:

Algorithm 3 Learning in an MDP

function LEARNING
  Initialize π randomly
  while not convergent **do**
    Execute π in MDP for trails with the depth of k
    Use VI function to update the traversed models
    Update π by some standards (e.g. $\epsilon - greedy$)
  end while
end function

3 BACKGROUNG AND RELATIVE WORK

3.1 Gridword, Chess and Go

3.1.1 Gridworld

Gridworld is a course project of Udacity – an on-line course provider. It is an one player game such that the target for the player is to arrive at the destination with positive reward as soon as possible. However, there are two types of objects, the wall and the black hole which are designed to hold the player back. The wall blocks the path while the black hole will drag the player to the starting point. A minimax tree where each parent will lead to four different children with fixed transition probabilities can be established throughout the game playing. The traditional Q-learning is suitable to train the data. Besides, the instant reward for each state is set to be negative which enables us to avoid the long path. This technique is also applied in our design for agent of the Chinese Checkers.

On the other hand, the algorithm can be developed without learning due to the limited size of the game board. Some techniques like alpha-beta pruning and Monte Carlo tree

search are applicable to find the sub-optimal move directly. The complexity of Gridworld programming is relatively smaller than that of the chess and the Chinese Checkers but the framework of our learning algorithm was deployed in Gridworld and tested for performance in the early stage of this project.

3.1.2 Chess and Go

The chess and the Go are much more complex than the Gridworld such that it is impossible to explore every possible move via the game engine. Most of traditional algorithms like alpha-beta pruning and minimax strategy all serve to seek for the best move based on some brute-force strategies. In knowledge of the state space of the next 'n' possible moves where the size of the space $O(n)$ is contingent on the capability of the game engine, the algorithms incorporate intuitive discarding strategies to evaluate the next best move and reduce the tree size.

The reinforcement learning algorithms have a few prominent features over traditional ones. They focus more on the value function rather than the state itself where the value is used to guide the algorithm to pick the best action in a model.[3] Second, the game plays are conducted again and again during when the value are updated along the trajectories to reinforce the knowledge. Under this setting, the agent improves well to mimic the best human player's behavior in a complex system. Thirdly, some techniques like artificial or convolutional neural network can be embedded to enable us to train a new engine from scratch, which entails maintaining of two things, an approximate value function and an approximation policy. There is no need to use some feature extraction techniques because the learning process renders us to teach a real novice and the system is believed to improve in an exponential manner.

Some outstanding chess programming like Knighcap was developed earlier in this century which learns weights using TD-leaf and searches the tree by Alpha-beta search with standard enhancement [4]. However, TD-leaf was not sufficient to yield the expected result until a breakthrough was finally achieved through Gerald's Tesauros work on backgammon. A new programming with combination of the classification of board state and the reinforcement learning was brought up by Block, Bader at el. [5]. A clear enhancement of the chess game engine's learning and playing skills were reached in their design. In our design for the Chinese Checker, the methodology of classification is also implemented and proves to work very well.

3.2 Chinese Checkers

The objectives of the Chinese Checkers is to be first to race one's pieces across the hexagram-shaped board into the 'home' - the corner of the star opposite the starting corner by hopping(jumping over the pieces) or rolling(one step moving).The agent for this game aims to make the largest use of the opponents' pieces to maximize its advance distance in each round as well as to break down the opponent's bridges. The basic strategy is to create or find the longest hopping path that leads closest to home, which directs us to design the following RL algorithm.

There are fundamental differences between the Chinese Checker and the low level board

game like Tic Tac Toe, Gomoku. For example, small changes in the position of those games will only lead to small changes in the position evaluation while the changes for the Chinese Checkers may have large impact in the evaluation. This reveal a huge challenge for the Chinese Checkers programming where the main focus of the search lies in choosing the future tactic. Of course, a fast evaluation of the position is required which makes the application of neural network inadvisable. However, as small differences between two figure locations may result in paramount differences in evaluation, the minimax strategy combined with the temporal differences learning are used to modified the input data of the neural network to guarantee the stable performance of the algorithm.

4 METHODOLOGY

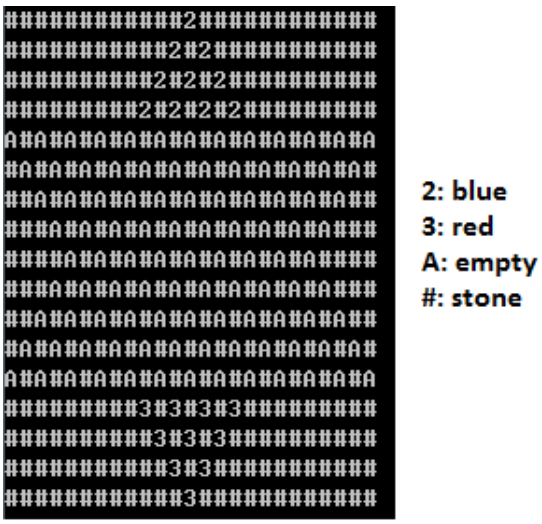

Figure 2: Board Representation

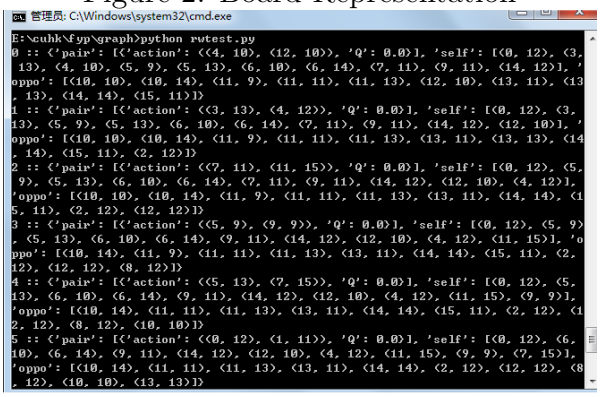

Figure 3: State Entry

In the first semester, we only considered the two-player case. Figure 2 illustrates the board representation od each game round where '2' represents the blue player while '3' for the red one. Figure 3 shows the state entries in the form of dictionary which is extracted from each board representation and stored in our database.

Based on the fact that the agent needs to behave differently before and after encountering the opponent's pieces - more attributes like blocking the opponent's path, breaking the bridge and horizontal variance should be taken into the consideration. The algorithm is then classified into three phases. The first phase ends up the first time two player collide with each other, the second phase ends when the pieces of both sides are totally separated and the third phase follows until the end of the game. The algorithm in the first two is similar where both emulate a minimax tree and conduct a top-down tree search for each trajectory. Nevertheless, the last phase adopts backward propagation idea which adds two nodes in the backward tree each round of game.

The value function serves as a connection between two conservative phases. The rewards of the intermediate states of the first two phases are set to zero while the leaves of the two minimax trees have rewards guided by the reward function. The balance of exploration vs. exploitation is taken into the consideration and several selection stan-

dards are employed to ensure good ratio between them. Plus, a threshold is used to speed up the learning procedure. In the effect of the discount, the feedback of a long path is so small and insignificant that we prefer not to add it into our database – we are prone to believe every trajectory with value function below the threshold is a bad one and should be avoided in the future selection. While the time for convergence may increase due to this partial effective learning process, the performance is saliently improved. Another thought is given to the negative rewards assignment as mentioned in the Gridworld. However, as the state in the trajectory will be only updated once during each iteration in Q-learning, the negative effect will accumulate in one state rapidly which requires quite a long time to offset and the convergent time will increase in a large degree. We finally choose the threshold method to direct our selection.

4.1 The 1^{st} and 2^{nd} phases

The main 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 choice for SARSA(λ) and we will compare their performance next semester. The details are given in Temporal Differences Learning section.

One way for storing values is mentioned above by using neural network. When a Q-factor is needed, it can be fetched from its neural network and when a Q-factor is to be updated, the new Q-factor can be used to update the neural network itself. The number of hidden layers and perceptrons play an important role in simulation. We attempt to design three neural networks with one hidden layer containing 10,30,50 nodes respectively while as far as we've tried, the performance of the network with one hidden layer and 10 neurons is below our expectation. We wish to conduct more practice with more delicate system in next semester. Whereas failing in emulating the neural network, we have another intuition to reduce the tree size and store the model of each state in memory directly. Hereby, we attempt to undertake the Monte Carlo tree search before the learning and only maintain the states with its first five largest reward actions. Besides, in few years ago Winands at el. developed an advanced version for MCTS as Monte Carlo tree search solver [6] which we still study on and hope to make use of in our programming design.

4.1.1 Temporal Differences Learning

The strategy used in the backgammon engine [5] consists of selection action a from a known state s, which minimizes the chance the opponent increases the evaluation in the

minimax tree:

$$a(x) = arg \min_{a \in \mathcal{A}_x} J'(x'_a, \omega)$$

where x_a denotes the position reached after the action a has been performed on x, and the $J'(x'_a, \omega)$ denotes the evaluation value of the position x. While in our algorithm, we introduce the reward function to replace the min-max strategy and assign the opponent reward to be negative which represents the state in the min layer of the minimax tree. In this way, the reward value of a state will be uploaded to its grandparent skipping the opponent's layer and all the value of function will be trained in 2-generation manner. Besides, as which one is on the offensive is not determined it is possible for us to complete the value iteration and train the data in both odd and even layers.

Q-Learning Q-learning is the basic step to study on other more complex RL algorithm. In Q-learning, the reward of each state is predicted only by the successive state, which causes slow converge speed. However, Q-learning keeps a relative high performance during the learning process and the wining ratio is monotonous increasing with more trajectories inserted. We do not include this learning algorithm in the programming but use it as a benchmark to test the performance of other advanced algorithm. The algorithm is presented as follows where α is the learning rate which will be trained in the for loop.

Algorithm 4 Q-learning

function Q-LEARNING
    Initialize $Q(s, a)$ randomly
    for α in $(0, 1)$ with step of 0.05 **do**
        repeat(for each episode):
            Initialize s, a
            repeat(for each step of episode):
                Take action a with $\max Q(s, a)$, observe $R(s, a), s'$
                $Q(s, a) := Q(s, a) + \alpha[R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    end for
end function

In advance, Q-learning can be extended to $Q(\lambda)$ with bootrapping the value of a given state to the value of proceeding states leveraging the idea of temporal differences learning. One of the famous $Q(\lambda)$ algorithm is Watkins's $Q(\lambda)$ which sacrifices much more computing resources than the traditional Q-learning but attain the result with more stable performance. We picked this algorithm as our learning machine which yields quite good results. The performance will be shown in the Result section.

Algorithm 5 Watkins's $Q(\lambda)$

```
1: function WATKINS'S Q(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:             a* ← arg max_b Q(s', b) (if a ties for the max, then a* ← a')
10:            δ ← R(s, a) + γQ(s', a') − Q(s, a*)
11:            e(s, a) ← e(s, a) + 1
12:            for all a do
13:                Q(s, a) ← Q(s, a) + αδe(s, a)
14:                if a' = a* then
15:                    e(s, a) ← γλe(s, a)
16:                else3
17:                    e(s, a) ← 0
18:                end if
19:            end for
20:            s ← s', a ← a'
21:    Until s is terminated
22: end function
```

Monte Carlo Method The value of the state s_t is estimated as the average reward with discount of the rest episode after visiting s_t. With a constant learning rate α $V(s_t)$ is updated once in each episode.

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t - V(s_t)]$$

where r_t is a long term reward return following the state s_t. r_t can not be accessed until the trajectory reaches the bottom of the tree, which keeps the time complexity of the algorithm relative large due to the repetitive calculation. However, techniques like dynamic programming may resolve this problem by calculating bottom-up. So far, we have not tried this method yet and left it to be a task in the next semester.

Algorithm 6 Monte Carlo

function MONTE CARLO
  Initialize $V(s)$ randomly
  repeat(for each episode):
    $e(s) = 0$, for all s
    Initialize s
    repeat(for each step of episode):
      Take action a given by π, observe $R(s), s'$
      Choose a' from s' using policy derived from Q (e.g. ϵ-greedy, UCT etc.)
      $\delta \leftarrow R(s) + \gamma V(s') - V(s)$
      $e(s) \leftarrow e(s) + 1$
      for all s in the trace **do**
        $V(s) \leftarrow V(s) + \alpha\delta e(s)$
        $e(s) \leftarrow \gamma e(s)$
      end for
      $s \leftarrow s'$
  Until s is terminated
end function

SARSA(λ) The SARSA(λ) algorithm is an on-policy algorithm whose main difference from $Q(\lambda)$ 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, $Q(\lambda)$ 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 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.)
      $\delta \leftarrow R(s, a) + \gamma Q(s', a') - Q(s, a)$
      $e(s, a) \leftarrow e(s, a) + 1$
      for all a **do**
        $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
        $e(s, a) \leftarrow \gamma\lambda e(s, a)$
      end for
      $s \leftarrow s', a \leftarrow a'$
  Until s is terminated
end function

TD-leaf TD-leaf is a combination of min-max lookahead and TD-learning. Following the minimax strategy the best action for each state is selected and at last the leaves reward will backward propagate to the state s_t dominated by

$$V(s_t) \leftarrow V(s_t) + \alpha[r + \gamma V(s_{leaf}) - V(s_t)]$$

[3]Unfortunately, as the learning process is separated into three parts the leaves reward in each phase may not be a dominate attribute to guide the separate tree developing. By now we did not figure out a good way to coordinate the leaves reward of the three phases. While in other board game like the chess[3], TD-leaf is proved to be an efficacious algorithm reaching a winning percentage of more than 95%.

4.1.2 Neural Network

The artificial neural network technique is utilized in this project and we extract the graphical characteristics from the figure locations. The input are 20 position indicator pairs of the current state and the output is the new value of next state. neurons in adjacent layer are lined together with a weight w_i and the activation a of each neuron is represented as a vector function of neurons from last layer $A = \{a_1, a_2, ...\}$

$$a = f(\sum_i w_i a_i)$$

Here we take the sigmoid function $g(z)$ as $f(.)$ to restrict the variation of the activation to $(0, 1)$ and transform the linear Q-factor into a non-linear mapping in manner of logistic regression

$$g(z) = \frac{1}{1 + \exp(-z)}$$

Backprop is a universal function approximator and ideally fit any Q-function.[8] We use this technique to approxiamte each weight. In this semester, We only tried a simple network with one hidden layer and 10 intermediate neurons. The performance after training is given in the Result section.
The convolutional neural network is more applicable in our case for its ingrained support for image analysis. The input are the geographical characteristics of current state and the output is those of next state. However, we did not try this method due to its high cost in computing resources. We will go back to it in next semester with the aid of GPU if possible.

4.2 The 3^{rd} phase

The main difference between the third phase and the first two is that the player needs not to take care of each action the opponent takes. Hence, only 5 out of 10 reward functions need to be taken into account and the mini-max tree is shrunk down to a single strategy tree. As the second phase ends at a large set of states which will lead to a single sink in the third phase, we establish the strategy tree in bottom-up manner and after each game round we add two nodes, one learned from the opponent behavior and the other added by searching the current tree.

On the other hand, as the states space in this phase is considerable smaller than the first two, the model based reinforcement learning may provide a favorable solution which has

a strong capability in balancing the exploration and exploitation.[9] E^3 algorithm is one of them which introduces a threshold ϵ ensuring the algorithm to probe a limited but promising region of search space while avoid being trapped in the local optima.

Model-Based Reinforcement Learning Model-Based Reinforcement Learning algorithm studies the model from each trial and attempts to build up the models for each state with simulated reward and transition probability. In E^3 algorithm, we define a known space containing all the states with refined value function and we treat the unknown state space as one single state with

$$R(N^C) = \max_k R(s_k)$$

where $s_k \in N^C$ after the approximated value function of state s performs good enough within the known space the algorithm returns and searches for a new path. The depth of the new path is inverse proportional to the learning rate α which also control the converge speed of value function. The pseudo-code fro E^3 algorithm is given below:[10][11]

Algorithm 8 E^3 Algorithm

Inputs: $\mathcal{S}, \mathcal{A}, \mathcal{N}, \alpha, \epsilon$ ▷ Where $\mathcal{S}$ = states space, $\mathcal{A}$ = action set, $\mathcal{N}$ = known set, γ = learning rate and ϵ = threshold
Outputs: V, π^* ▷ Where V = expected value after training, π^* = optimal policy
function E^3
    Initialize $\mathcal{N} = \emptyset$
    repeat until $\mathcal{N} == \mathcal{S}$
        Pick s randomly
        if $s \notin N$ **then**
            Take action $a(s)$ with fewest chosen times at s
        else
            repeat until $V_{\mathcal{N}}(s, a) \leq V^*(s, a) + \frac{\epsilon}{2}$
                update $V(s, a)$ by learning from trajectories (Q-learning)
                return s and $\pi^*_{\mathcal{N}}$
            Follow policy $\pi_{\mathcal{N}}$ for T steps where $T = \frac{1}{1-\alpha}$
        end if
end function

4.3 Policy

The policy is highly relevant to the issue of exploration vs. exploitation. We experience different policies greedy, ϵ-greedy and UCT in this project and examine their effect on the exploration ratio. Noted that the tree size is so large that only a small portion will be visited, we use the instant reward function to guide us for selection in a never visited state which in practice proves to be much better than selecting randomly.

4.3.1 Greedy

For each state, $a(s)$ is chosen by the agent which leads to the largest possible value. Thus, with this policy, no exploration is performed and at each step, the agent is prone

to go as far as he can leveraging his knowledge of maximum. However, in our real test, the greedy policy behaves not so bad probably attributable to the high randomness of the game environment. The game tree grows large bunches of branches which disperse each trajectories and the agent has high probability to enter new local situations and explore new states in each game round.

4.3.2 ϵ-greedy

ϵ-greedy policy is an improvement of the greedy one with an affiliated geometric distribution. For each state visited by the agent, the probability each action is chosen is in subordinate with a geometric distribution with parameter ϵ. Experiments we run for $\epsilon = 0.1$ and $\epsilon = 0.5$. In both cases, there is no significant improvement during the learning process which indicates that the randomness of the game environment provides so sufficient exploration opportunities to the agent that no more strategies are entailed.

4.3.3 UCT

UCT method was first developed by Levente and Csaba and applied to multi-stage decision making models like MDP by Chang, Fu at el. For each state the algorithm opts to select the action with the highest value defined by the formula that

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

where

- w_i represents the winning times after passing the i^{th} node.
- n_i represents the visiting times of the i^{th} node.
- c is the exploration rate which is theoretically set to $\sqrt{2}$ but chosen empirically in practice
- t represents the total number of trajectories.

The first component refers to the exploitation while the second component focuses on the exploration. However, the result shows that the UCT does not only improves the previous result but even makes it worse. What's worse, the converge speed was also slow down by UCT. The reason is still not testified and we will keep a sight on that in the future research.

Above all, the greedy policy seems to defeat other policies and gain the best performance. Hence, the project use the greedy policy for all practices.

4.4 Players' Strategy

In this section, we will describe the experiment we've conducted with two different types of players. One uses the greedy strategy while another operates with the same strategy as we adopt but with the weights trained by last game round. Notwithstanding the random player is always considered as primitive rivalry in the board games, we think it's not appropriate to train our data by playing with it. As the Chinese Checkers runs

on a hexagonal board with large horizontal range, it's easy for a random player to get trapped in edge corners. However, this situation does not seemingly take place in real practice against human player. Hence, random player in our case is so weak that playing with it is not likely enhance the performance but even increment the burden on storing value function in database.

4.4.1 One taking Greedy Action

The agent learning with SARSA(λ) algorithm is trained by playing with the greedy player who chooses the action with greatest reward for each state. Without the aid of neural network, the capability of storing value of function is restricted by the computer memory. We generate 800 episodes totally and as we observe this training process revamps the performance of the algorithm quickly but also always lead it to learn lousy trajectories. The outcome is positive and at last the algorithm could defeat more than 90% greedy players.

4.4.2 Self-play

On the other hand, a dynamic self-improved opponent is designed which will simulate our algorithm's previous action in each game and improve itself once a new value of function is updated in our database. Our algorithm is believed to learn quickly in this manner because its opponent grows stronger and stronger during the process. The learning speed may stay slow in the early stage but rise sharply shortly afterwards.

5 RESULTS

In this section we discuss the outcomes we obtained after the data training (829 trajectories in total). Particularly, two measures of the performance are mainly concerned where one is the reward after the 1^{st} phase and another is the total steps achieved by one player in a game round. Figure 4 shows the outcome of self-play where two players use the same learning strategy with the same learning rate. As we observe in Figure 5 and Figure 6, the capability for each players grows simultaneously during the process and the result largely depends on the condition that who takes the first step. in our representation, the blue one takes the first step and the red one follows.

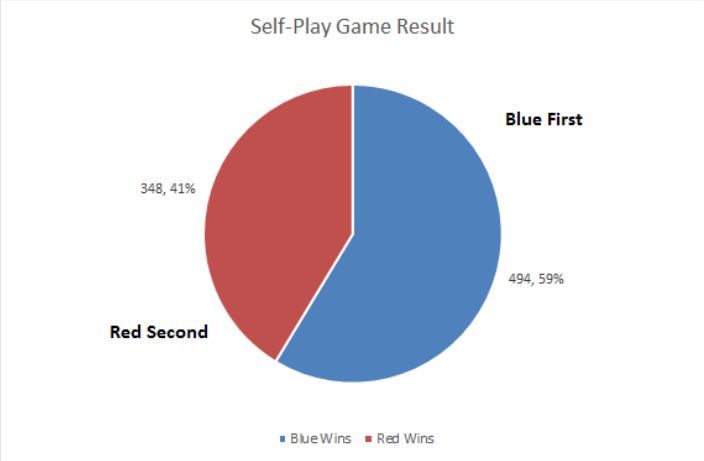

Figure 4: Final result for self-play

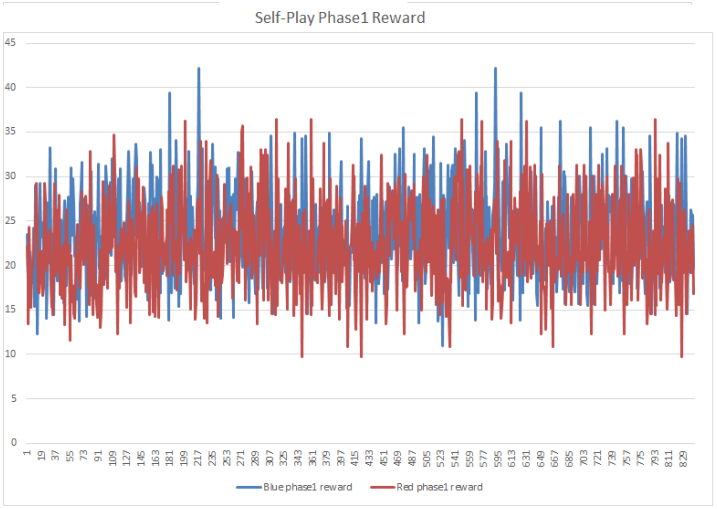

Figure 5: The 1^{st} phase reward in self-play

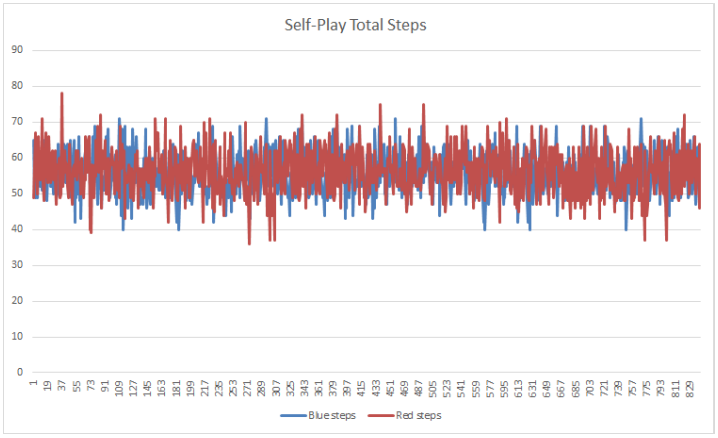

Figure 6: Total steps in self-play

Also we modified the our algorithm by playing it with a greedy one. Figure 7 and Figure 8 clearly show the improvement in the AI's performance and up to around 500 matches the AI overtakes the greedy player and behaves better and better as the process goes.

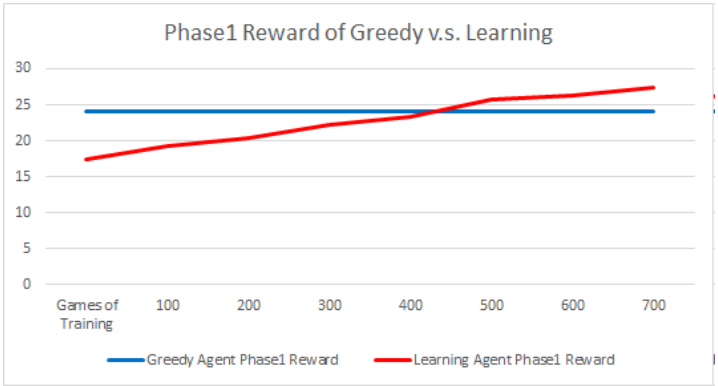

Figure 7: The 1^{st} phase reward in greedy-play

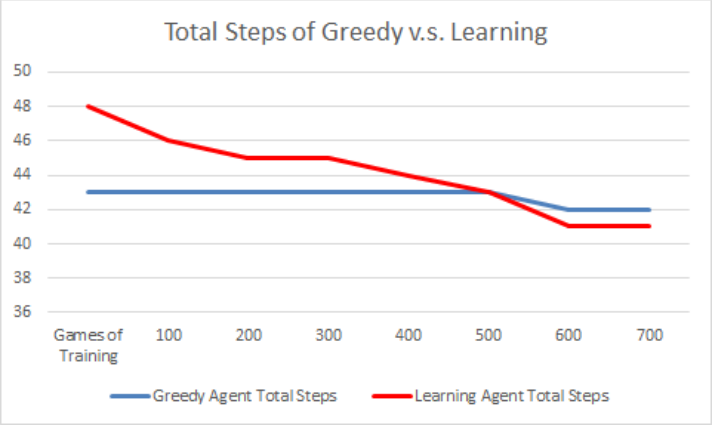

Figure 8: Total steps in greedy-play

To better illustrate the situation, for the testing sake we abort the selection rule in the algorithm but instead designate the algorithm to choose the action with the highest value when playing the greedy player. While the exploration/exploitation ratio will be sharply reduced in this way but the performance of our AI will increase definitely. The results are shown in Figure 9 and Figure 10 indicating our AI's better performance over the greedy one after 500 matches.

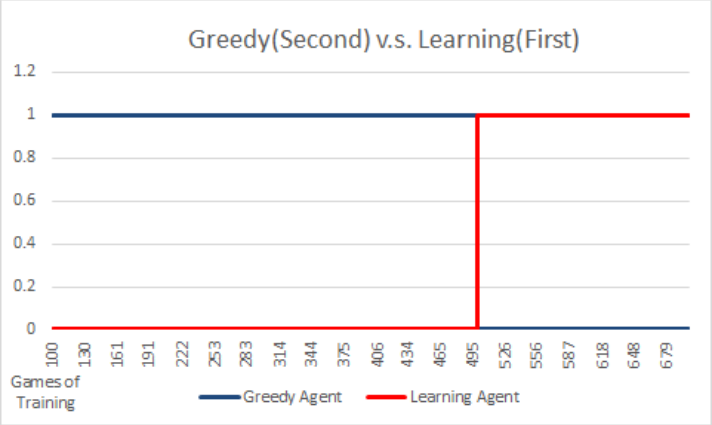

Figure 9: Learning agent first

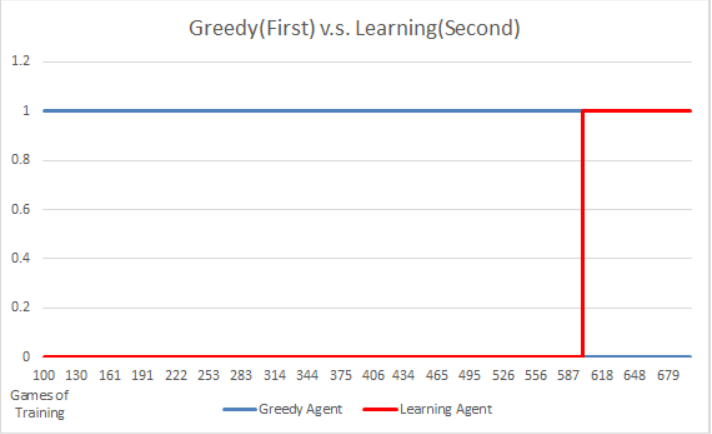

Figure 10: Greedy agent first

As the last part of our experiment, the AI is tested by playing with the skilled player - human beings to see how it behaves when encountering the player with unknown strategy. As shown in Figure 11 and Figure 12, the AI did not behave as well as playing with the greedy player probably because our training set is not large enough to contain the intricate moves which the skilled player may take.

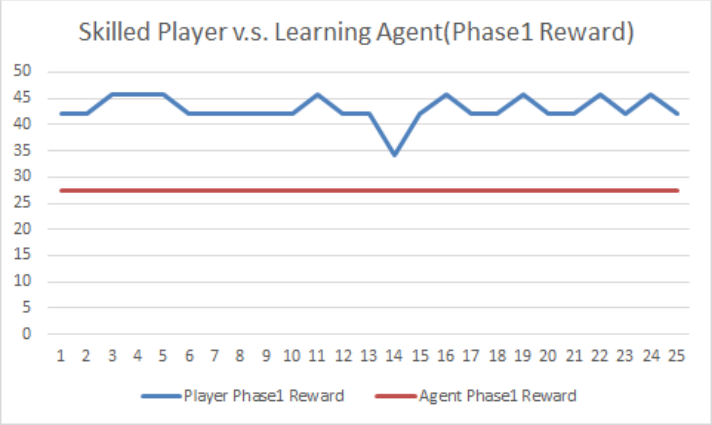

Figure 11: The 1^{st} phase reward in human-play

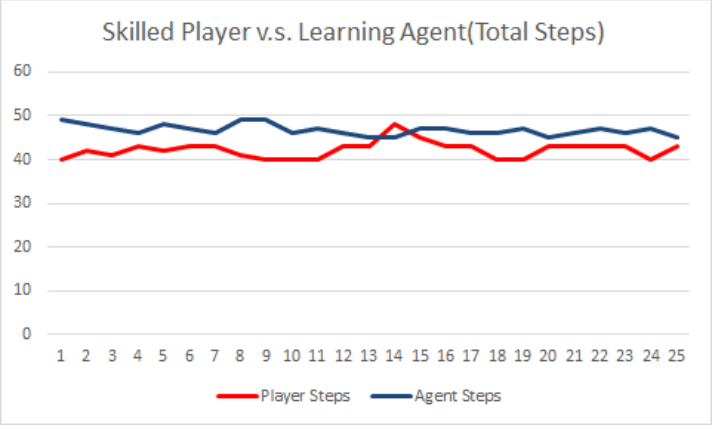

Figure 12: Total steps in human-play

6 CONCLUSION

As we see in our experiments, SARSA(λ) combined with greedy selection rule behaves quite well in playing against both greedy player and self-playing. However, we are not sure it is strong enough to play against more intelligent players, even human beings. Given the fact that the algorithm seems improve quickly up to a certain point and then stabilize we believe there are some rules easily to be pick up from the raw board representation while the learning process will stagnate if no further information is provided. We will discuss the solutions to step further in the following section.

7 FUTURE DIRECTION

There are four future directions. The first one is to improve our learning algorithm because so far we have only tried Q(λ) and SARSA(λ) but left the model-based learn-

ing and Monte Carlo method behind. We wish to implement those two and test their performance. Moreover, future work could be using the TD-learning with different parameter λ to compensate for the overfitting or overbias problem.

Second, the more complex neural network will be deployed in our solution to relax the model-storing pressure and in the future work, we will be devoted to study the neural network with one hidden layer but more intermediate neurons. The third direction lies on the selection mechanism. As we observe, the greedy one seems perform good enough for exploration which we believe could be improved by more advanced method.

On the other hand, we will explore multiple-player case in the future which appears widely different from the two-player case. The reward function as well as learning rate will be altered. Plus, more advanced and efficient learning algorithms are supposed to be apply.

8 WORK DIVISION

This project was completed by two people: Li, Haocheng and Chen Ken. Li mainly covered the theoretical study of the background and proposed proper algorithms as well as the learning methodology. Together with Chen Li took the on-line courses from Udacity and Coursera in machine learning from which he learned how to establish the learning framework for the game. He also worked with Chen to assign the reward functions. Besides, Li researched in each model-free learning algorithm from previous study so as to compare their performance under the context of this game. The model-based algorithm was left to next term for deep study. Chen worked on the practice and programmed each details in Python. He embedded the game basic rule and mechanism into the profile and design the delicate user interface. Besides, he set up two models, one for training and backup and another for playing against human players. For the training purpose Chen additionally constructed three rivalries described in the previous section. He was responsible for exploring python supporting library as well as training data by the method presented above.

The detailed work in each week completed by each people is shown in Appendix A.1.

9 ACKNOWLEDGEMENTS

We are grateful to professor Lau Wing-Cheong for his guide in finding appropriate learning algorithm suitable for Chinese Checkers. Also we're appreciated the help from Sijun He, Wenjie Hu and Hao Yin from Stanford University for their previous results of applying reinforcement learning in Chinese Checkers.

References

[1] S. He, "Playing chinese checkers with reinforcement learning," *CS 229 Spring 2016 Project Final Report*, 2016.

[2] A. Ng, "Cs229 lecture notes," *Stanford css229: Machine Learning*, vol. Part XIII, 2010.

[3] I. C. Dubel, L. L. Bsc, I. J. Brandsema, and S. S. Bsc, "Reinforcement learning project: Ai checkers player," *Utrecht University*, 2006.

[4] M. Lai, "Giraffe: Using deep reinforcement learning to play chess," *arXiv preprint arXiv:1509.01549*, 2015.

[5] M. Block, M. Bader, E. Tapia, M. Ramírez, K. Gunnarsson, E. Cuevas, D. Zaldivar, and R. Rojas, "Using reinforcement learning in chess engines," *Research in Computing Science*, vol. 35, pp. 31–40, 2008.

[6] M. H. Winands, Y. Björnsson, and J.-T. Saito, "Monte-carlo tree search solver," in *International Conference on Computers and Games*, pp. 25–36, Springer, 2008.

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

[8] A. Gosavi, "Neural networks and reinforcement learning," *Department of Engineering Management and Systems Engineering*, vol. MO 65409, 2014.

[9] T. Hester, M. Lopes, and P. Stone, "Learning exploration strategies in model-based reinforcement learning," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pp. 1069–1076, International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[10] P. D. P. D. Farias, "Explicit explore or exploit (e3) algorithm," *2.997 Decision-Making in Large-Scale Systems*, 2004.

[11] M. Grześ and D. Kudenko, "Pac-mdp learning with knowledge-based admissible models," in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pp. 349–358, International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[12] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural computation*, vol. 2, no. 4, pp. 490–501, 1990.

A Appendices

A.1 Weekly Log

- *Week 1 ∼ 3*
  - Theoretical work
    - Finished the on-line courses in supervised learning in Coursera including gradient descent, artificial neural network and its integration with convolution and Bayesian network.

    * Finished the on-line courses in unsupervised learning in Udacity about supervised learning including supported vector machine, PAC and techniques of fixing overfitting/overbias ratio.
    * learned the basic reinforcement learning algorithm and their simple applications.
    * Twigged Chinese checkers rules and common used tricks, and conceived the general solution to apply machine learning onto it. (deciding which learning method we primarily want to apply). Up to this point, we were inclined to use model-free learning algorithms.
    * Realized the requirement and expectation of game agent against different level of players which need to be dealt with in different ways.
  - Practical work
    * Completed the function code of that on-line project with Python.
    * Designed the chess board and user interface of the Chinese checkers in Python
    * Implemented the piece move functions and embedded the game rules, such that we can play jumping board game manually

- *Week 4 ∼ 6*
  - Theoretical work
    * Searched for the existing work for board game and studied their methodology. In this case, we mainly focused on how different techniques combined to generate an ensemble learning scheme.
    * Studied the existing deep learning schemes for other chess game like artificial neural networks + Monte Carlo tree search implemented in AlphaGo(Go), alpha-beta pruning + temporal differences learning implemented in chess ([4]), SARSA(λ) learning in [3] and Monte Carlo tree search solver in [6].
    * Based on the learning pattern (SARSA(λ) for example) which was chosen for our first experimental AI we constructed an evaluation function and Markov decision tree, and plus, carried on necessary math calculation to design an algorithm fitting the learning method including design the in-loop iteration equation and practical method to tune weights.
  - Practical work
    * Implemented our algorithm in Python and set up the database to store states from three different phases.
    * Implemented the self-play mechanism.
    * Implemented the auto-play mechanism with random probability space

- *Week 7 ∼ 8*
  - Theoretical work
    * Optimized the algorithm by improving the evaluation function or adding smart widgets in programming like the value threshold mentioned in main body. Noted that the value of function was trained by thousands

of trajectories and each each trajectory followed the trained policy. We can altered the weight for each attribute to avoid overfitting and got better results.

- Practical work
  - Implemented the auto-play mechanism with random probability space
  - Implemented Q learning data structure probability space

- *Week 9 $\sim$ 10*
  - Theoretical work
    - Used the artificial neural network to implement the game agent with determining the input characteristics and output attributes.
    - Studied the algorithm of using gradient descent to improve the neural network in [12].
  - Practical work
    - Implemented our algorithm in Python and set up the database to store states from three different phases.
    - Implemented SARSA(λ) algorithm.

- *Week 11 $\sim$ 12*
  - Theoretical work
    - Evaluated the performance of our agent by playing against with players adopting ϵ-greedy algorithm.
    - Evaluated the performance of our agent by playing against with itself(self-playing).
    - Examinated the computer resource consumption by our agent and optimized our algorithm to reduce it as possible.
    - Compared the outcome with other existing works [6]
    - Summarized the results and limitation of our methods ass well as decided our future plan.
  - Practical work
    - Collected data from the training and stored them into database.
    - trained the learning rate, weight and discount.
    - Programmed each opponents in details and tested our algorithm's performance by playing against them.