Playing Chinese Checkers with Reinforcement Learning

CS 229 Spring 2016 Project Final Report

Sijun He Wenjie Hu Hao Yin
[sijunhe, huwenjie, yinh]@stanford.edu

***Abstract*—We built an AI for Chinese checkers using reinforcement learning. The value of each board state is determined via minimaxation of a 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. The performance of our modified minimax strategy with tuned weights stands out among all the other strategies.**

I. Introduction

Chinese checkers is a game played on a hexagram-shaped board that can be played by two to six players individually or as a team. The objective is to be the first to move all ten pieces across the board into the opposite starting corners. As shown in Figure 1, the allowed moves include rolling and hopping. Rolling means simply moving one step in any direction to an adjacent empty space. Hopping stands for jumping over an adjacent piece into a vacant space. Multiple continuous hops are allowed in one move. A more detailed introduction of the Chinese checkers can be seen in Wikipedia.

The reinforcement learning is an area of machine learning typically formulated as Markov decision process (MDP). The model consists of states, actions, transitions, etc., which is suitable for decision making in board game like Chinese checkers. The objective of our project is to use reinforcement learning to build an AI agent for Chinese checkers, and to explore the effectiveness and efficiency of the AI.

To simplify the problem, our AI only solves the one vs one mode of Chinese checkers. Different from classic reinforcement learning where at each state the player solves a simple maximization problem, in our AI for Chinese checkers, it is a adversarial zero-sum game. Therefore, each player needs to consider not only his/her own strategy, but also the opponent's responding strategy. Therefore, a better way to depict such procedure is minimaxation, which is elaborated in Section 3.

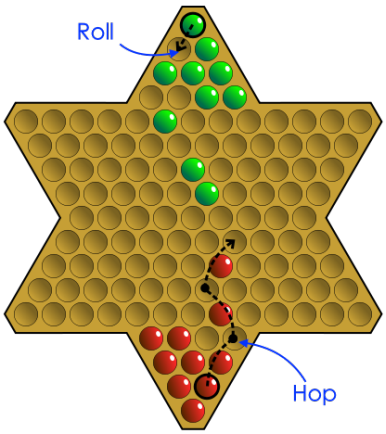

Fig. 1. Chinese checkers rules (downloaded from website)

The biggest challenge in applying reinforcement learning to our AI is how to learn the weights in function approximation. We modified the standard algorithm for function approximation so that it fits our minimax setting, and we adopted a diminishing learning rate to stabilize the update. Our simulation results showed that this learning procedure is effective, in that the difference between the weights

before and after an iteration is small. Simulation results also showede the robustness of our update, in that our learning procedure with different initializations will end up with very close weights.

We tested the performance of different strategies by playing against a random look-ahead greedy player. Simulation results showed that the minimax strategy with tuned weights significantly outperforms the minimax strategy with initial weights. Moreover, we further modified our strategy such that it divides the game into three stages and applies different strategies thereon. Simulation results showed that this modified strategy outperforms the basic minimax strategy.

The rest of this report is organized as the follows. We first talk about how we implement the board in Section 2. Then we introduce the basic methodology of our AI in Section 3, and point out the difficulties in implementation as well as our solution in Section 4. We cover our modified strategy in Section 5. Simulation results are shown in Section 6.

II. Board Representation

Figure 2 is the starting board where we worked on in minimax searching, weights tuning, and simulations. Each **o** stands for a vacant spot, **1** stands for a spot occupied by player 1's piece, and **2** stands for a spot occupied by player 2's piece.

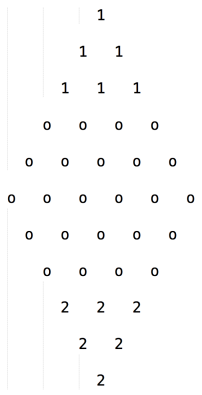

Fig. 2. Board representation

Note that this board is smaller than the original board of Chinese checkers. The state-space complexity of Chinese Checker 10^{23} is high [1], thus building and testing the AI for the full game board is computationally intensive. Thus we adopted a smaller board of 6 pieces for each player during development and simulation. Furthermore, the hexagram-shaped board was modified into a heuristic diamond-shaped board, which is reasonable for one vs one mode.

III. Methodology

We adopted the classical approach for game playing AI, game search tree, which best mimics the behavior of a human player while demonstrates super-human performance by taking advantage of the computing power. With each node representing a board state of the game, and each edge representing one possible move from one board state to a subsequent board state, the game search tree can emulate the thinking process of a human player. Chinese checkers is a two-player zero-sum game, thus an objective "value" is needed to evaluate the situation on the board. Player 1's goal is to maximize the "value", while Player 2 minimizes it. The logical approach is the minimax tree, which is a decision tree that minimizes the possible loss for a worst case scenario resulted from the opponent's optimal move.

Due to the large state-space complexity of Chinese checker, it is unrealistic to build a top down game search tree. Instead, a shallow k-depth minimax game tree that searches only the tip of the tree is built. At each node, the "value" is taken as the "minimax score" which is computed by the minimax algorithm of depth k. When the search has reached the bottom of the k-depth search tree, a.k.a. the leaves, the score is approximated by a raw score, which is a linear evaluation function based on the features of the board state. We exploited 6 features that are based on the positions of pieces on the board, which are described as the following:

- A_i: the squared sum of the distances to the destination corner for all pieces of player i;
- B_i: the squared sum of distances to the vertical central line for all pieces of player i;
- C_i: the sum of maximum vertical advance for all pieces of player i;

with $i = 1, 2$. Note that we used the square of the distances to penalize the trailing pieces, which would motivate each player to make the pieces cohesively and thus promote hopping. Besides, the 6 features are extracted from a larger amount of possible features, so that the overfitting of the model was avoided. The other features we have explored and found unnecessary included:

- the horizontal variance (how scattered) of pieces of each player;
- the vertical variance of pieces of each player;
- The maximum vertical advance for a single piece of each player;

The evaluation function is the form of

$$\widetilde{V} = w_1(A_2 - A_1) + w_2(B_2 - B_1) + w_3(C_1 - C_2)$$

where the weights $\boldsymbol{w} = (w_1, w_2, w_3)^T$ would be trained via function approximation, which would be described in the following section.

IV. Challenges and solutions

A. Weights tuning and function approximation

The performance of the AI is highly dependent on how well the weights in the evaluation function is tuned. The objective of weights tuning is to allow the evaluation function to consistently approximate the minimax value through the depth-k tree search. The challenge is to develop an algorithm to improve and stabilize the weights within certain iterations.

The following Algorithm 1 based on function approximation is our solution to perform weights tuning. An introduction of function approximation can be seen in [2]. The basic idea of the algorithm is to conduct value iteration after each game played by both AI players. A new weight is computed by performing least squares on the recorded feature vectors at each turn and their corresponding minimax scores. A diminishing learning rate α is imposed at each iteration, in order to stabilize the update. The iteration is repeated until the weights value are stabilized.

Algorithm 1 Weights Tuning

1: Initialize a weights $\boldsymbol{w}$;
2: **repeat**
3: Play one game with both player following Minimax-rule using weights $\boldsymbol{w}$, record the feature vectors at each turn in a matrix $\boldsymbol{\Phi}$ and the corresponding minimax scores in a vector $\tilde{\boldsymbol{v}}$;
4: $\boldsymbol{w}_{new} \leftarrow LeastSquare(\boldsymbol{\Phi}, \tilde{\boldsymbol{v}})$;
5: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(\boldsymbol{w}_{new} - \boldsymbol{w})$
6: **until** stabilized

B. Run-time complexity and alpha-beta pruning

Another challenge is the run-time complexity of the algorithm. Due to the nature of Chinese checkers, players usually have around 20 – 100 feasible moves at a typical turn. The worst-case number of board states evaluated in a minimax tree of search depth 4 is on the scale of 10^8. The large branching factor, combined with any non-trivial search depth, can easily result in impractical run-time for real-time game play.

We address this problem by adopting alpha-beta pruning. A detailed description of this technique can be seen in [3]. Furthermore, in order to expedite the alpha-beta pruning, we enqueued all the feasible moves of the player in a priority queue where the priority is in the order of the weight-calculated score of the resulting board if the corresponding move is taken. The effectiveness of the alpha-beta pruning is shown below in Table I, where we listed the time used as well as the number of nodes visited for the starting board to compute the minimax value. It shows that the time used as well as the total number of nodes visited is significantly reduced.

	time (s)		nodes visited	
depths	w/o	w/ pruning	w/o	w/ pruning
2	4.07	0.85	196	27
3	84.39	7.31	4032	301
4	1763	30.68	82944	848

TABLE I
TIME USED AND NUMBER OF NODES VISITED IN COMPUTING THE MINIMAX VALUE

V. Further Development

In the strategy described above, we adopted minimax tree search at each turn of playing. Note that there is a waste of computing power in this strategy

at the beginning and end of game. At the beginning of game, the pieces of two players have not interact with each other, thus each player's move does not interfere the other's. Therefore, it is unnecessary to consider the opponent's strategy, thus the minimax strategy can be simplified as a pure maximizing strategy. This is also true at the endgame, when the pieces of two players are split up, thus each player only needs to consider how to end the game as soon as possible, without considering the opponent's strategy.

In the light of this knowledge, we modified our strategy above in the following way. In the start game, the player only consider his/her feasible moves and chooses the one that gives the maximal weights-calculated value. In the midgame when two players' pieces intersect, we search our optimal move via the minimax procedure. In the endgame, the player would take a move that achieves maximal vertical advance.

We will test the performance of this strategy and compare it with the basic strategy in Section 6.B(3).

VI. Results

A. Convergence of weight tuning

With Algorithm 1, we were able to tune and stabilize the weights. The criterion for stabilization is $\boldsymbol{w}_{new} - \boldsymbol{w}$. As shown in the plot, the difference of weights diminished as more training games were played. While the algorithm didn't necessarily converge, it did improve the effectiveness of the AI dramatically after weights were tuned, which will be shown in Section 6.B(1).

Furthermore, the algorithm is also robust to initializations. Two different unit-length initial weights were attempted, $[0.577, 0.577, 0.577]^T$ and $[0.990, 0.099, 0.099]^T$, and both of them converged to the similar values after weight tuning. The tuned weights from different search depth are shown below:

$$\boldsymbol{w}_{(\text{depth}=2)} = [0.911 \quad 0.140 \quad 0.388]$$

$$\boldsymbol{w}_{(\text{depth}=4)} = [0.902 \quad 0.004 \quad 0.431]$$

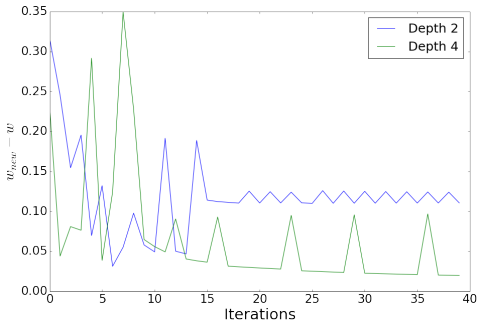

Fig. 3. Convergence of weight tuning

B. Benchmarking

We measured the performance of our algorithms by simulating 200 games against a benchmark strategy. The benchmark is a greedy random look-ahead algorithm that takes the move that gives the most combined vertical advance in 2 steps. Tiebreaker is preference to trailing pieces and further ties are broken by random selection. The result was measured by winning steps, which is the number of steps needed for the losing player to finish the game.

1) Effects of weights: Figure 4 demonstrates the game results of AI players with tuned weights and untuned weights. The untuned weights are the initial weights $[0.577, 0.577, 0.577]^T$. The results show that the AI with tuned weights performs significantly better than one with untuned weights, which is expected.

2) Effects of search depth: Figure 5 shows the game results of AI players with search depth 2 and 4. As expected, the AI with search depth 4 outperforms the same strategy with search depth 2. It is worth noting that the worst case runtime for depth 2 strategy is under 5 seconds, while the worst case runtime for depth 4 strategy is over 900 seconds.

3) Effects of modified strategy: Figure 6 compares the game results of the basic minimax strategy and the modified strategy. The modified minimax strategy improves the performance tremendously for depth 2, while there's no significant improvement for depth 4. The reason is that the endgame algo-

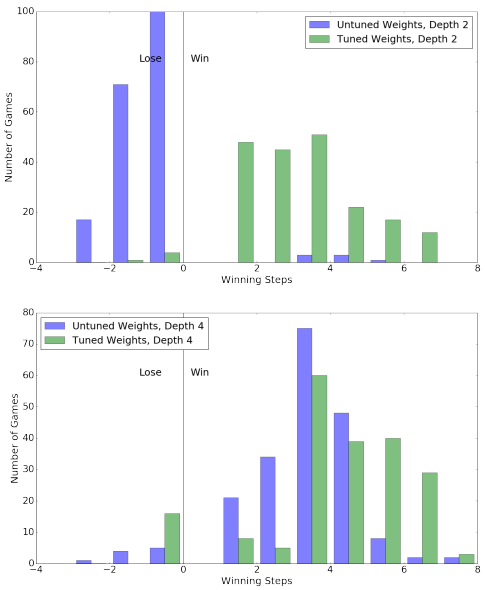

Fig. 4. Effects of weights

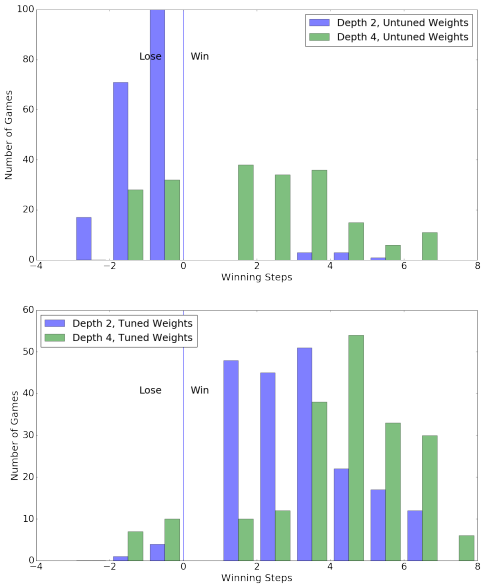

Fig. 5. Effects of search depth

rithm in the modified strategy is comparable to the minimax algorithm with a search depth 4. Though there's no improvement in terms of winning steps, the modified strategy is orders-of-magnitude faster in terms of time complexity, reducing the runtime for an average of 400 seconds to less than 10 seconds.

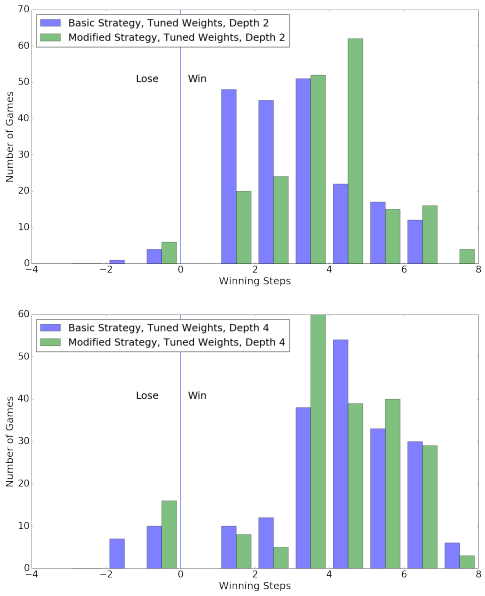

Fig. 6. Effects of modified strategy

REFERENCES

[1] Bell, George I. 2009. The shortest game of chinese checkers and related problems. *Integers* **9**(1) 17–39.

[2] Liang, Percy. 2015a. Lecture 8: MDPs II. http://web.stanford.edu/class/cs221/lectures/mdp2.pdf.

[3] Liang, Percy. 2015b. Lecture 9: Games I. http://web.stanford.edu/class/cs221/lectures/games1.pdf.