Efficient Learning in Chinese Checkers: Comparing Parameter Sharing in Multi-Agent Reinforcement Learning

Allen Gu and Noah Adhikari

University of California, Berkeley

Abstract

We show that multi-agent reinforcement learning (MARL) with full parameter sharing outperforms independent and partially shared architectures in the competitive perfect-information homogenous game of Chinese Checkers. To run our experiments, we develop a new MARL environment: variable-size, six-player Chinese Checkers. This custom environment was developed in PettingZoo and supports all traditional rules of the game including chaining jumps. This is, to the best of our knowledge, the first implementation of Chinese Checkers that remains faithful to the true game.

Chinese Checkers is difficult to learn due to its large branching factor and potentially infinite horizons. We borrow the concept of branching actions (submoves) from complex action spaces in other RL domains, where a submove may not end a player's turn immediately. This drastically reduces the dimensionality of the action space. Our observation space is inspired by AlphaGo with many binary game boards stacked in a 3D array to encode information.

We use proximal policy optimization (PPO) to train all of our agents. Our experiments investigate three variations of parameter sharing over our agent's architectures: independent, shared-encoder, and fully-shared. The independent approach involves training six entirely separate agents, each with their own policy. The shared-encoder architecture involves sharing the encoder network between all agents, but each agent having its own policy and value function heads. Our fully-shared variation involves sharing all parameters between all agents. All policies use action masking to avoid illegal moves.

Our experiments first analyze the performance of these three approaches on our Chinese Checkers environment for board size $N = 2$. We train policies through each of the three variations and evaluate them throughout the progression by simulating games against random opponents. Each random opponent samples a legal move uniformly at random. During evaluation, we introduce two metrics: win rate and game length measured in the number of turns to win.

We find that all three variations achieve a 100% win rate against the random opponents. The policy trained through full-parameter sharing achieves these results most efficiently, within the first 50,000 environment steps. The shared encoder model achieves a 100% win rate soon after while the independent approach lags far behind. Additionally, we look at the metric of game length–how many turns a policy takes to win. In this metric, as well, full parameter sharing trains more efficiently and converges with better performance.

For the trained policies, we briefly analyze game strategy and raise concerns over distributional shift in self-play, especially when using parameter sharing. Although our trained policy was still performant against random opponents, many possible states are unexplored through our training methods. Attempts to increase exploration through changing PPO's entropy coefficient led to inconclusive results and did not noticeably improve performance. We attribute the robustness of our trained policies to the fact that they fixate on their own pieces rather than the global board state.

In the head-to-head-matches, we find that full parameter sharing considerably outperforms the partially-shared and independent architectures at all stages of training.

We attribute the increase in performance to the fact that the fully shared model is able to take advantage of the homogenous environment to effectively learn six times as quickly compared to the independent approach. We do not expect the same performance gains in heterogenous environments, but similar methods in stochastic environments may prove effective.

The PettingZoo environment, training and evaluation logic, and analysis scripts can be found on [Github](#).

1 Introduction

Multi-agent reinforcement learning (MARL) has been a growing topic of interest in recent years[1]. Most prior work in this area has focused on two-player competitive, zero-sum games such as Go[2] or positive-sum cooperative games[3]. Several frameworks and tools have been built to support these environments[4,5].

The behavior of agents in competitive environments with more than two players is much less well studied[1]. Stochastic games such as poker are popular in this domain, but complete-information environments are not as common, and it is difficult to find out-of-the-box environments for competitive multiplayer complete-information environments. In this paper, we develop a custom PettingZoo environment for Chinese Checkers, a six-player competitive homogenous complete-information game, as well as study the performance of several MARL architectures on this environment.

Though reinforcement learning for Chinese Checkers has been explored before[6,7], our implementation is, to the best of our knowledge, the first to remain faithful to the true game.

2 Background

2.1 Chinese Checkers Rules[8]

Chinese Checkers is a game traditionally played with 2, 3, 4, or 6 players of differing colors on a star-shaped board as seen in Figure 1. Each player traditionally has 10 pegs, which are initially placed in the corner of the board closest to them. The goal of the game is to move all of one's pegs to the corner of the board opposite to their starting corner.

2.1.1 Movement

The game is played in turns. On a player's turn, they must move one of their pegs to an adjacent empty space, or jump over an adjacent peg of any color to an empty space on the other side. A player may jump over multiple pegs in a single turn. A player can only move their own pegs. Only if a player has no possible moves may they pass their turn entirely.

2.1.2 Winning

The first player to move all of their pegs to the opposite corner wins the game. Though it is possible to continue playing after this point for ranking purposes, we focus on the variant where the game is terminated after the first player wins.

2.1.3 Spoiling

It is possible for a player to prevent another from winning by continously occupying one of the pegs in their goal corner, in a process known as "spoiling.". There are no official rules preventing spoiling[8], and rules prohibiting it tend to be quite complicated, so we permit it in our implementation.

2.2 Hexagonal Coordinate Systems

The geometry of Chinese Checkers lends itself nicely to a hexagonal grid, but it is nontrivial to represent hexagonal grids, and there are various ways to do so.[9]

One way involves exploiting a symmetry between cubes and hexagons, where each axis of the cube corresponds to the projection in 2D-space of one of the three axes of the hexagon. One can then describe the position of a hexagon in terms of three coordinates (q, r, s). Since the hexagonal grid lies in a plane, however, the coordinates are subject to the constraint $q + r + s = 0$. This is known as the cube coordinate system.

Because of the planar constraint, only two of the three qrs-coordinates are necessary. (q, r) is a common choice; this is known as the axial coordinate system.

2.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization[10] (PPO) is an on-policy method with an architecture similar to actor-critic algorithms. It addresses stability concerns in traditional actor-critic algorithms by limiting large policy updates. The algorithm accomplishes this through a clipped surrogate objective:

$$L = \mathbb{E}\left[\min\left(\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}\hat{A}_t,\ \text{clip}\left(\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]$$

In the objective, the advantage term incentivizes taking actions that are better than average. The ratio $\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$ is a measure of how far the new policy deviates from the previous one. By clipping and

minimizing with the non-clipped ratio, the objective also deters making large changes to the policy.

3 Methods

3.1 Chinese Checkers Environment

We developed a custom PettingZoo environment for Chinese Checkers to support MARL. The board has the same shape as the traditional game, but can be configured to play with different sizes N, where N is the side length of one of the corners of the board. For example, the traditional game has $N = 4$ and the $N = 2$ variation that we used for many experiments has 3 pegs per player.
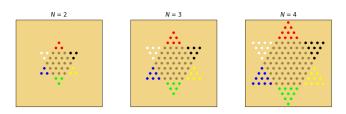

Figure 1: Game boards of size $N = 2, 3, 4$

3.1.1 Number of Players

For simplicity, we focus on the 6-player variant of Chinese Checkers as we wanted to investigate several algorithms in multiplayer self-play. However, our methods can be applied to other numbers of players.

3.1.2 Turn Limits

Unlike games such as chess, which impose turn limits to prevent infinite horizons[11], Chinese Checkers has no such rules. We impose a turn limit that ends the game with no winners after a large, configurable amount of turns (we found ≈ 200 to be a good choice for $N = 2$ and ≈ 1000 to be a good choice for $N = 4$).

3.1.3 Submoves

In Chinese Checkers, it may be difficult to enumerate all of an agent's possible moves, as jumps increase the number of possible moves significantly. Additionally, within a single move, it is possible to infinitely stall the game by jumping in a cycle.

To prevent this, we adopt the concept of branching actions from complex action spaces in other RL environments[12], where a move consists of one or more submoves that may not end the player's turn

immediately. In Chinese Checkers, a submove is a single jump, adjacent-space movement, or end-turn action. Jump submoves may not necessarily end a player's turn, but to prevent infinite turns, players cannot jump back to a space they previously occupied on that turn. Players can end their turn following a jump using the end-turn submove.

3.1.4 Board Representation

The game board uses cube coordinates, with the center of the board being $(q, r, s) = (0, 0, 0)$.

Note that due to the star-shaped nature of the board, it is difficult to represent the board as a perfect hexagon. To circumvent this, we simply treat the board as a hexagon of diagonal length $(4N + 1)$, which corresponds to a circumscibed hexagon about the star. Extraneous spaces are then masked out to prevent movement out-of-bounds.

For symmetry, we rotate the board so that the current player's home triangle is at the top of the board when making moves.

The 3D representation of the board is quite sparse, so we flatten the board with the axial coordinate constraint $q+r+s = 0$ when communicating with agents.

3.1.5 Observation Space

The observation space is a flattened representation of 8 different layers of the board, where each layer provides unique information about the game state. Each layer is as a two-dimensional $(4N + 1, 4N + 1)$ grid where row and column indices (i, j) map to axial coordinates (q, r) of the board space. To maintain rotational symmetry, the board is always rotated when creating the observation so that the current player's home triangle is at the top of the board. The layers represent the following binary information for each space on the board:

Layer	Description
0	Current player's pegs
1-5	Other players' pegs, in clockwise order starting from the current player
6	Where the current player has jumped from this turn
7	Where the current player jumped during the last submove, if the last submove was a jump

After flattening, the observation space becomes a one-dimensional $(4N + 1) \times (4N + 1) \times 8$ binary vector.

3.1.6 Action Space

Actions in the Chinese Checkers environment are represented by a tuple $(q, r, \texttt{direction}, \texttt{is\_jump})$:

(q, r)	Position of the peg being moved in axial coordinates
$\texttt{direction}$	Direction of movement from 0 to 6 in angle increments of $\frac{\pi}{3}$, where 0 is due-right, 1 is up-right, etc.
$\texttt{is\_jump}$	Whether the action is a jump

If a previous submove was a jump or if no legal moves are available, a player may also end their turn immediately without a move (known as the end-turn action). Thus, we represent the action space as a discrete space of size $(4N + 1) \times (4N + 1) \times 6 \times 2 + 1$.

3.1.7 Action Masking

For each agent, the observation, in addition to the observation space, also includes an attribute $\texttt{action\_mask}$ which is a binary vector of size $(4N + 1) \times (4N + 1) \times 6 \times 2 + 1$ that indicates which actions are legal for the current agent during the current submove. This is used to mask out illegal actions when sampling from the action space.

3.1.8 Rewards

Various reward schemes were tried:

1. *Sparse:* $+5N$ for winning, $-N$ for losing, 0 otherwise

2. Sparse reward, plus a small *goal bonus* for entering pegs in the goal $(+0.1)$ and penalty for removing them from the goal (-0.1)

3. Sparse reward, plus a tiny *movement bonus* for moving forward $(+0.001)$ and a penalty for moving backward (-0.001)

4. *Positive-sum:* $+5N$ for winning, 0 otherwise, plus goal and movement bonuses and penalties

We found that agents with sparse rewards had difficulty learning, so we ultimately opted for a positive sum, dense reward scheme (4) due to our goals involving self-play.

3.1.9 Environment Stepping

The pseudocode for a single environment step within our game is summarized below. Each step corresponds to moving a single peg within the game or ending a turn. Consecutive jumps are handled across multiple environment steps.

Algorithm 1: Environment stepping in Chinese Checkers

  function step (a)
    Input: a: Integer action in
        $[0, (4N+1) \times (4N+1) \times 6 \times 2 + 1)$
    Convert integer action to a submove
    Rotate the board to the current player
      and perform the submove
    Update rewards for agents
    if *submove is not a jump* **then**
      Advance to the next player
    end
  end

3.2 Multi-Agent Learning

There are many approaches to configuring models in multi-agent settings. In this section, we describe both our model algorithm and the variations we tested on the Chinese Checkers environment.

3.2.1 Model Architecture

All agents learn through Proximal Policy Optimization (PPO) with a clipped surrogate objective[10]. In our architecture, the actor and critic share an MLP encoder but have differing policy and value function heads.

Encoder Network: The encoder network takes in a vector the size of the observation space as input. Each encoder contains a hidden layer of size 64 and output layer of size 64, with ReLU activation functions.

Policy Head: The policy head is a single fully connected network of shape $(64, (4N + 1) \times (4N + 1) \times 6 \times 2 + 1)$ that takes the encoder output as input and outputs action logits.

Value Function Head: The value function head is a single fully connected network of shape $(64, 1)$ that takes the encoder output as input and outputs a scalar estimating the value of the state.

3.2.2 Variations on Multi-Agent Setup

We compare three different policy configurations for the multiple agents: fully-independent, shared-

encoder, and fully-shared variations.

- *Fully-Independent:* Agents learn separately with their own single-agent PPO policy.

- *Shared-Encoder:* All agents share the same encoder, but have unique policy and value function heads.

- *Fully-Shared:* Agents share all parameters, effectively training a single policy.

3.2.3 Training Procedure

We trained all three policy configurations using self-play. As the policies are updated, the agents become stronger and thus have stronger opposition, so this algorithm is reminiscent of curriculum learning[13]. The training algorithm is summarized below:

Algorithm 2: Agent self-play training procedure

function train (n, policy)

    Input: n: Number of training iterations
              policy: Policy configuration

    Initialize 6 policies following the given configuration, sharing parameters if necessary

    repeat n **times**

        Sample a batch of experiences by playing against the other policies

        Update the models using PPO

    end

end

3.2.4 Evaluation Procedures

We evaluate the performance of trained agents via several metrics.

Random agent evaluation: The evaluated agent plays against a field of random agents. The number of wins and the number of turns taken to win are both recorded.

Head-to-head matches: Each of the three architectures contributes two agents in a match of six players. The positions of the agents are randomized, and the agents play several games against each other. The number of wins per player and the number of turns taken to win are both recorded.

Algorithm 3: Evaluation procedure

function evaluate (n, policies)

    Input : n: Number of games
              policies: Six configurations

    Output: *wins*: Wins per policy
               lengths: Game lengths

    repeat n **times**

        Play a game with the given policies

        Record whether the game was a win for each policy into *wins*

        Record the number of turns taken for the game to terminate into *lengths*

    end

    return *wins, lengths*

end

---

4 Experiments

4.1 Comparison of Multi-Agent Configurations

First, we compare the different multi-agent setups: fully-independent, shared-encoder, and fully-shared. Each algorithm was trained over 100 iterations of 4000 timesteps each, with evaluation after every timestep. During evaluation, one trained policy would compete for 30 rounds against five random policies until one player wins or 150 turns have been played.

From the results in Figure 2, we find that as more parameters are shared between the agents, the more efficient our training becomes. Win rate against random opponents is a better metric of model performance compared to average rewards because although longer games are less optimal, they tend to receive larger rewards.

Full parameter sharing reached over a 90% winrate in less than 50,000 environment steps of training while the fully-independent algorithm took almost 200,000 environment steps. Note that the winrate of an algorithm excludes games that were truncated due to the turn limit.

Additionally, the game durations during evaluation were shorter for parameter-sharing methods. The policy trained through a fully-shared approach was able to win in fewer turns, where one turn consists of all moves made by a single player before allowing the next player to act. From Figure 2, full parameter sharing converges to a policy that wins in nearly 20 moves, faster than the other two approaches.

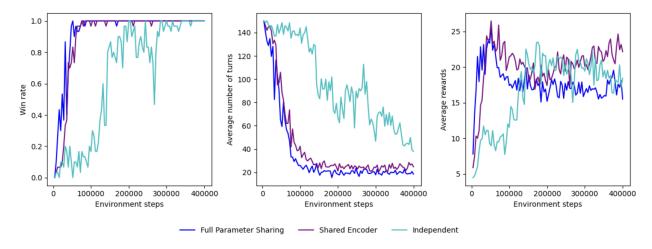A naive strategy for Chinese Checkers could be

Figure 2: Evaluation occurred at repeated intervals throughout training for fully-independent, shared-encoder, and fully-shared multi-agent configurations. **Left:** Win rate of policy against five random policies. **Middle:** Average game length (turns made by policy). **Right:** Average rewards.

to move each piece from the home triangle to the target triangle one by one, without jumps. For a board of size $N = 2$, this strategy performs 20 moves. The optimal strategy including jumps would improve this by a couple moves. Interestingly, even when facing opponents that may both harm (e.g. block the target triangle) or help (e.g. increase the potential for jumps), our model is near this hypothetical optimum.
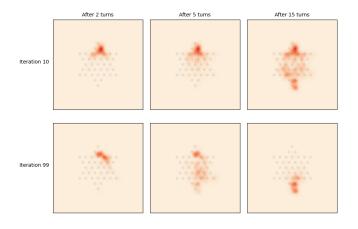
4.2 Analyzing Policy Strategies

Figure 3: Heatmaps for the policy trained through full parameter sharing. These plots show the frequency of different peg locations on the board after some amount of turns have been made. The trained policy plays as red with pegs starting at the top triangle.

Although all trained models eventually reach a 100% win-rate against random players, policies tended to move the pieces to the target zone in an independent fashion: moving one piece first, then the next, and so on. To visualize these patterns, we run training checkpoints 100 times against five random policies. During each game, the board states were collected at different numbers of turns. These board states were used to generate heat maps in Figure 3 showing the frequency of pegs throughout the board.

This behavior seems to be learned early on in training, as shown by Figure 3. By the 10th iteration, turn 15 shows a high probability that some pegs are still in the home triangle while others have already made it to the target triangle. A similar pattern can still be seen in the 99th training iteration, though at an earlier number of turns. In turn 5, many pegs are still left in the home triangle while others are far down the board.

4.3 Head-to-Head-to-Head Matches

For head-to-head matches, we trained each of the three architectures for 100 iterations of 4000 timesteps each. After each iteration, we evaluated the performance of each policy by having it play 20 games against the other two policy configurations, with a turn limit of 200. Note that truncated games with no winner are not treated as wins for any player, hence the winrates may not sum to 1. The results are shown in Figure 4.

The fully-shared policy outperforms the other two policies when they are pitted against one another. Over time, the policies also complete games more quickly, indicating more efficient strategies.

We hypothesize that the fully-shared policy is able to take advantage of the homogenous environment to effectively learn six times as quickly com-
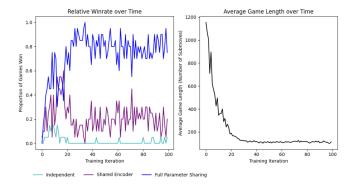
Figure 4: Win rates and game length of all three architectures piloting two randomized players per game throughout training.

Figure 5: Effect of varying the entropy coefficient c on average game lengths against random opponents.

pared to the independent approach. We expect full parameter sharing to still perform well in stochastic homogenous environments, but heterogeneous environments will probably not see the same performance gains.

disproportionately fixate on their own pieces rather than the global board state in trying to get to the goal so they are more robust to distributional shift in evaluation.

4.4 Exploration

Distributional shift between training and evaluation is a concern for multi-agent self play. The distribution of observed states in self-play is not representative of all potential opponent policies. In the context of Chinese checkers, as a self-play agent improves, the board state will tend to follow certain patterns dependent on its learned policy. As a result, out-of-distribution states (such as those from a random policy) may harm agent performance during evaluation.

We hypothesized that encouraging exploration when parameter-shared MARL algorithms are trained in self-play would improve performance in the random agent evaluation. In PPO algorithms, exploration can be encouraged by an entropy coefficient c. This entropy term is added to the clipped surrogate objective. However, testing this claim through a hyperparameter sweep on c yielded inconclusive results.

$$L = \mathbb{E}[L^{CLIP} + cS(\pi_\theta(s_t))]$$

PPO agents with full parameter sharing for $c = 0.0, 0.005, 0.01$ were trained for 50 training iterations of 4000 environment steps each and evaluated against random opponents. The average evaluation game length curves for the different hyperparameter values are shown in Figure 5.

Due to the variance in game lengths, it was difficult to draw meaningful conclusions about the effect of exploration on parameter sharing approaches to MARL. We posit that the agents may
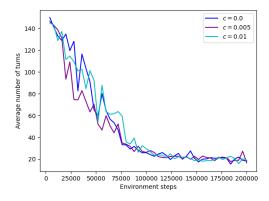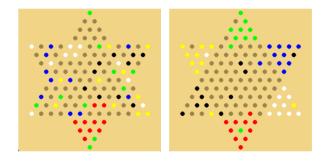
4.5 Extension to Larger Boards

Figure 6: End game states for boards of size $N = 4$. A full parameter sharing policy trained for 100 iterations plays as red against (**left**) random opponents and (**right**) itself.

For larger boards $N > 2$, the spoiling mechanic in Chinese Checkers becomes more prominent which prevented good results to compare the different multi-agent schemes. Although we tried experiments on larger boards, pegs would often become stuck or spoil other agents, rendering metrics like win rate less useful because most games end without a winner. Figure 6 shows end game states for $N = 4$.

5 Conclusion

We have introduced a new competitive environment for multi-agent reinforcement learning. On

this environment, we showed that even in multi-agent competitive settings, parameter sharing over multiple agents can improve training efficiency and performance. We found that PPO agents trained with full parameter sharing performed the best when using game length as an evaluation metric. A shared encoder training approach also performed better than the fully independent setup. We attribute these performance gains to the homogenous environment, for which parameter sharing is particularly well-suited.

5.1 Group Contributions

We collaboratively developed the PettingZoo Chinese checkers environment and MARL training and evaluation logic during synchronous work sessions. We also worked together on all sections of the report. **Noah Adhikari** extended the evaluation framework to compare the trained agents in head-to-head gameplay. **Allen Gu** analyzed the performance of the independent, shared-encoder, and fully-shared methods in play against random opponents and also coded the heatmap plotting to analyze the policies.

References

[1] S. Gronauer and K. Diepold, "Multi-agent deep reinforcement learning: a survey," *Artif Intell Rev*, vol. 55, pp. 895–943, 2022.

[2] D. Silver, A. Huang, and C. Maddison *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016.

[3] A. Oroojlooy and D. Hajinezhad, "A review of cooperative multi-agent deep reinforcement learning," *Appl Intell*, vol. 53, pp. 13677–13722, 2023. https://doi.org/10.1007/s10489-022-04105-y.

[4] J. K. Terry, B. Black, A. Hari, L. S. Santos, C. Dieffendahl, N. L. Williams, Y. Lokesh, C. Horsch, and P. Ravi, "Pettingzoo: Gym for multi-agent reinforcement learning," *CoRR*, vol. abs/2009.14471, 2020.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[6] Z. Liu, M. Zhou, W. Cao, Q. Qu, H. W. F. Yeung, and V. Y. Y. Chung, "Towards understanding chinese checkers with heuristics, monte carlo tree search, and deep reinforcement learning," *CoRR*, vol. abs/1903.01747, 2019.

[7] S. He, W. Hu, and H. Yin, "Playing chinese checkers with reinforcement learning," 2016.

[8] M. T. Games, "The rules of chinese checkers," 2023. https://www.mastersofgames.com/rules/chinese-checkers-rules.htm.

[9] R. B. Games, "Hexagonal grids," 2021. https://www.redblobgames.com/grids/hexagons/.

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[11] FIDE, "Laws of chess," p. 13, 2008.

[12] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," *CoRR*, vol. abs/1711.08946, 2017.

[13] Y. Du, P. Abbeel, and A. Grover, "It takes four to tango: Multiagent selfplay for automatic curriculum generation," *CoRR*, 2022.