ShengJi+: Playing Tractor with Deep Reinforcement Learning

Jerry Shan

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-127
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-127.html

May 12, 2023

Acknowledgement

I would like to thank Professor Joseph Gonzalez and Sky Computing for supporting me throughout my graduate studies and my research project. I also want to thank Professor Sergey Levine for his amazing CS 285 course in fall 2022, without which I would not have discovered my interest in deep reinforcement learning.

ShengJi+: Playing Tractor with Deep Reinforcement Learning

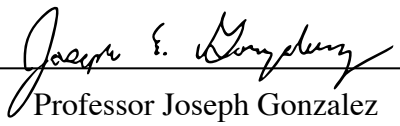
by Jiarui Shan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Joseph Gonzalez
Research Advisor

5/11/2023
(Date)

* * * * * * *

Professor Sergey Levine
Second Reader

5/11/23
(Date)

ShengJi+: Playing Tractor with Deep Reinforcement Learning

Abstract

ShengJi+: Playing Tractor with Deep Reinforcement Learning

by

Jiarui Shan

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

In recent years, humans have made significant progress in building AIs for perfect and imperfect information games. However, trick-taking poker games are still considered a challenge due to their complexity. Tractor (a.k.a. ShengJi) is a 4-player trick-taking card game played with 2 decks of cards that involves competition, collaboration, and state and action spaces that are much larger than the vast majority of card games. Currently, there is no existing AI system that can play Tractor. In this work, we present `ShengJi+`, an effective AI system for the Tractor game powered by deep reinforcement learning and Deep Monte Carlo. `ShengJi+` is trained using self-play for $\sim$1.2 million games and achieves 97.6% Level Rate over the random baseline agent. In addition to the main architecture, we also experiment with several training techniques for Tractor and discuss why they do or do not work based on the match statistics. Through case studies, we believe that `ShengJi+` exhibits intelligent and rational playing strategies that resemble human Tractor players. We open-source our code[1] to motivate future work on this topic and to introduce Tractor as a new benchmark for imperfect information multi-agent reinforcement learning, and to introduce `ShengJi+` as a strong baseline for future research into this game.

[1] https://github.com/themoon2000/shengji_plus.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the history of reinforcement learning, most works have focused on developing intelligent agents that could interact with and attain certain objectives in a perfect information setting, such as an AI chess player, or a robot that can run for a long distance. In these tasks, the agent has full knowledge about the environment, and is possible to always make the optimal move based on this information, or even learn the environment dynamics. In contrast, in imperfect information environments, the agent has access to observations which only contain part of the entire environment state, creating more challenges in learning the optimal policy.

In recent years, there has been considerable progress in using deep reinforcement learning to train AI players for imperfect information games like Japanese Mahjong (Li et al., 2020), Texas Hold'em (Brown and Sandholm, 2018), DouDiZhu (Zha et al., 2021; Zhao et al., 2022), GongZhu (Shi et al., 2021), Dota 2 (Berner et al., 2019), and so on. This is partly achieved by designing methods that account for and are robust against the inherent uncertainty of the environment as observed by the agent. Research into imperfect information games is important, as the uncertainties of game environments resemble uncertainties that an agent must face when interacting in the real world.

In this work, we study Tractor, a multi-agent and imperfect information poker game. there is no existing AI system that can play Tractor using deep reinforcement learning. The properties of Tractor, including its complicated game rules, dynamic trump, large state and action space, competition, collaboration, multiple separate but causally linked phases, on top of being an imperfect information game make it a difficult RL problem to tackle.

Chapter 2

Background of Tractor

Tractor is a 2 vs 2 trick-taking poker game. Specifically, Tractor belongs to the family of *point-trick* games. In these games, the goal of the players is to maximize the total points they earn. The standard Tractor is played with 2 decks of cards. Although playing with more or less decks of cards is possible, the rules are so different that they can be considered distinct games. Compared to DouDiZhu, another popular card game, Tractor has more game rules and more players, allowing for more complex collaboration and competition strategies. In addition to the required rules, there are also many optional rules that can be added to Tractor depending on what the players agreed upon. Some rules increase the state and action space dramatically.

In Tractor, the players sit in a circle and across their teammate, and each team collectively earns as many points as possible. In each round, one of the players is the dealer, and the dealer's team is called the *defenders*. The other team is the *attackers*, and the defenders' goal is to prevent the attackers from scoring 80 points or higher. There are 200 points in total, and they come from the cards that the players play. Each `5` is worth 5 points, each `10` is worth 10 points, and each King is worth 10 points. All other cards have no point value. In each trick, one player leads with a combination of patterns from the same suit, and the other players need to follow the pattern combination and play the same number of cards. There are three types of patterns:

- A single card (single).
- A pair of identical cards.
- Consecutive pairs in the same suit (called a *tractor*). We use the term n-**tractor** to denote a tractor consisting of n consecutive pairs.

The players' moves in a trick decide who wins the trick. At the end of a trick, all the points in the cards played go to the team that wins the trick, and the winner of the trick will lead the next one. The dealer leads the first trick. A round of Tractor ends when all cards have been played. If the defenders win (e.g. the attackers did not reach 80 points), they keep

defending and the dealer's teammate becomes the dealer in the next round; if the attackers win, they become the defenders and the player to the right of the current dealer becomes the next dealer.

A round of Tractor is played with a predetermined dominant rank between 2 to A, and during the round there are 4 distinct phases – declaration, kitty, bidding, and the trick phase – each requires reasoning over different state and action spaces. Each bidding phase is followed by one more kitty phase for the bidder.

- **Declaration phase**: the players take turns drawing cards from 2 shuffled decks. At any time, a player may declare the trump suit for this round by revealing 1) a card in the dominant rank, 2) a pair of identical cards in the dominant rank, 3) a pair of Black Jokers, or 4) a pair of Red Jokers. These declaration types are listed in order from small to big, and bigger declarations can override smaller ones. Each player draws 25 cards in total, leaving 8 on the table.

- **Kitty phase**: the dealer (in the first round, the last player to declare becomes the dealer, and in other rounds, the dealer is predetermined) takes the remaining 8 cards and discards 8 cards from his hand of 33 cards, The discarded cards, the *kitty*, will be revealed at the end of the round.

- **Bidding phase** (chao di): this is an optional component of Tractor. After the dealer places the kitty, other players take turns to decide if they want to override the dominant suit by revealing a pair of identical cards in the dominant rank or a pair of Jokers. The order of the bids is ♦, ♣, ♥, ♠, Black Jokers (XJ), Red Jokers (DJ). Players who bid takes the current kitty and places 8 cards back as the new kitty. Once a player has bidded, the dealer can also take part in bidding.

- **Main phase**: the dominant suit is settled and now the cards are divided into trumps and non-trumps. A card is a trump if it is a Joker, in the dominant rank, or in the dominant suit (if Jokers won the bid, there's no dominant suit). In each trick, the leading player plays a pattern, and the other players follow the pattern counterclockwise. A pattern consists of a suit (all trump cards are considered as the same suit) and a pattern structure (single, pair, tractor, or a combination of those). If a player cannot exactly match the pattern, they are required to match as much as they can by playing cards from the same suit. If they don't have enough cards in the suit, they can play anything as long as the total number of cards being played add up to the size of the pattern. If the pattern is non-trump and a player has no cards in that suit, they can *ruff* the pattern by matching it using trump cards. The player who has the biggest cards in the trick collects all the point cards that appeared in the trick for his team, and leads the next trick. If the attackers win the trick, they also collect all point cards in the kitty, multiplied by 2 raised to the size of the pattern (e.g. single card = 2, pair = 4, two consecutive pairs = $2^4 = 16$, up to 2^6). The attackers win if they collected at least 80 points in total, in which case, the attackers and defenders switch roles.

The details of pattern matching and ruffing are rather complicated and are explained in Appendix 2 along with examples for readers who are interested.

Winning a round is only part of a Tractor game. Each team also has a rank. The ranks start with `2`, and the winning team levels up. Each team's goal is to level up faster than their opponents and be the first team that levels up past rank `A`. The outcome of a round depends not only on which team won, but also on the number of points earned by the attackers. If the attackers earned at least 80 points, they become the defenders (with no rank gained), and for every 40 additional points earned, their team increases their rank by 1. On the other hand, if the attackers earned below 80 points, the defenders level up by 1 rank; if below 40 points, the defenders level up by 2 ranks; and if 0 points, the defenders level up by 3 ranks. Since winning by n ranks at once has the same effect as winning by 1 rank n times, the effective win rate for team A against B is the proportion of ranks won by team A among the total ranks won by the two teams. The goal of the players is to maximize the expected ranks their team wins in any round.

Tractor is a complicated game for AI systems. First, the state is large. Each player starts the round with a hand of 25 cards, which is more cards per player than in most card games, and cards with the same rank but different suits are considered distinct. In each round, there is a dominant rank that is the same as the rank of the dealer's team and a dominant suit that is settled during the round. The dominant rank and suit together determine the ranking of the cards, so the same hand has very different meanings depending on this information. Therefore, a good AI system needs to adapt its strategy depending on the dominant rank and suit.

The legal action space of Tractor is variable and difficult to enumerate directly. In the standard version of the game, the trick leader can play arbitrary combinations of patterns from the same suit. Due to computational limitations, however, we focus on a simplified version where the trick leader can only play **a combination of at most 3 patterns**, e.g. 1 single + 2 pairs. Even after this simplification, the legal action space of the game could still be very large, because if a player can't fully match a pattern, they can choose an arbitrary card to play for each card that they can't match. For example, if the pattern consists of 5 cards in a suit a player does not have, the player can choose any 5 cards to play, meaning there are $\binom{25}{5} = 53130$ possible actions to consider. Furthermore, unlike card games that enforce action ordering (e.g. in DouDiZhu, the next player's move must be bigger than the current player's move), a move in Tractor can be either bigger or smaller than the previous player's move, complicating the legal action space. Overall, Tractor is challenging for RL because many RL algorithms only work well on small and static action spaces, so additional action and state space reduction techniques need to be considered.

The variable action space also poses another problem, which is that some extremely high reward moves are rarely present in the legal moves, and when they do, the chance that the agent actually plays the move is close to zero, so it could take a long time for an RL agent to discover the move. For example, winning the last trick with a tractor (e.g. 3344) as an attacker will multiply the kitty points by 16. If there are at least 10 points in the kitty,

which is often the case, such move will add 160 points to the attackers' score, meaning that they win the round and level up at least 2 ranks in addition – an extremely good outcome. However, tractors rarely show up in a hand of 25 cards, and even if one does show up, unless the agent strategically saves those 4 cards till the end by planning ahead and winning the second last trick, it wouldn't discover the 16x kitty points opportunity.

Each round of the game involves hundreds of discrete actions played sequentially, making the correlation between the early actions of the players and the final outcome very weak. Also, sometimes bad moves only become costly if the player plays enough of them, so as a result, the outcome of a round is not always a strong indicator of how good the agent's individual actions were, and the model can only learn how good individual moves are when a huge number of games are played.

Chapter 3

Related Work

3.1 Policy Gradient Methods

Policy gradient methods are a family of deep reinforcement learning approaches that directly optimize a policy network by trying to maximize the expected cumulative reward. Some of the variants of these methods include REINFORCE (Williams, 1992), Actor-Critic (e.g. A3C (Mnih et al., 2016), SAC (Haarnoja et al., 2018)), PPO (Schulman et al., 2017), TRPO (Schulman et al., 2015).

While policy gradient methods are popular and known to work well for a wide variety of control tasks, they suffer from two critical weaknesses when being used against Tractor due to Tractor's immense and varying action space. In Tractor, the possible action space is insanely big (too big to even enumerate). Fortunately, at any point, the legal action space for a player is quite small relative to it. However, policy networks are only good at making a prediction within a known and fixed action space, so it is not efficient for tasks where the legal actions are sparse within the possible action space, as most of the computation is wasted on evaluating illegal actions.

Another related weakness of policy gradient methods is that they don't tend to generalize to previously unseen actions, especially for problems with large discrete action spaces (Dulac-Arnold et al., 2015). When a policy network makes a decision in a discrete action space, it considers each action separately and does not learn the relationship between actions from data. Due to the immense possible action space in Tractor, it is infeasible for a trained policy to observe all actions during training, so when it encounters an unseen action during test time, it has no idea how good it is because it has never learned to optimize that particular action during training.

3.2 Deep Q Learning

Deep Q-learning (Mnih et al., 2015) is another type of model-free RL algorithm that learns a function $Q : (\mathcal{S}, \mathcal{A}) \to \mathbb{R}$ that estimates the expected cumulative reward for taking action

$a \in \mathcal{A}$ at state $\mathcal{S}$ and acting optimally thereafter. The network is trained on the Bellman equation

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a').$$

The policy is never learned directly, but can be extracted from a Q function.

DQN is a plausible approach for Tractor because it can handle its large state space and dynamic legal action space.

A key advantage of DQN compared to PG is that it can potentially generalize to previously unseen actions. This is because in DQN, the action is part of the input to the model, so a neural network, being a smooth function approximator, could reason about a new action at some state from past training data where the action and state have been very similar.

There are two main limitations of DQN in Tractor. First, Tractor is a task with long trajectories, which means that Q-learning may take a long time to converge, as for each time step, the network updates itself based on its estimations for the next time step, so the quality of Q-value estimations for time step t would only be good if the network's estimations for time step $t + 1$ are good. Second, the Bellman update requires computing the max over states and actions, but the action space varies from state to state, so it's inconvenient to do the updates in batches. As a result, we expect DQN to be slow to train compared to other methods.

3.3 Neural Fictitious Self-Play

Neural Fictitious Self-Play (NFSP) is a deep reinforcement learning algorithm that aims to combine the idea of self-play with imperfect information. The agent learns a Q network and a strategy estimate network during training. The Q network is used to approximate the best response of the agent, like in Deep Q-Learning. The strategy estimate network's purpose is to model the average best response of the agent's opponent, and outputs a probability distribution over the opponent's actions, from which fictitious play samples are drawn during training. The authors showed that NFSP achieved state-of-the-art performance on the two-player game Leduc Hold'em.

Unfortunately, NFSP faces the same problem that policy gradient methods face, and that is the action space of Tractor is too large to either calculate a probability distribution over, or for the agent to have seen all actions during training. Additionally, the legal actions a player has in Tractor depends on their hand, which is private, so the other players don't even know this player's action space. Therefore, it's infeasible to estimate other players' strategies in Tractor.

3.4 Deep Monte Carlo

Deep Monte Carlo (Silver et al., 2016) is another algorithm in deep reinforcement learning that uses Monte Carlo simulation to estimate a value function, bypassing the need to su-

pervise the value function using past versions of itself (as in value iteration). In the original paper, the authors applied DMC to Go and used a value network and a policy network to perform Monte Carlo Tree Search. Instead of performing an exhaustive tree search (which is infeasible for Go), the value function can reduce the tree depth by approximating the values of the subtrees below a certain depth.

In DouZero (Zha et al., 2021), the authors used the same Monte Carlo idea to approximate a Q function instead. To optimize the Q function, they first use it as an argmax policy (in an epsilon-greedy way) to sample random episodes, then for each state-action pair (s, a), they update $Q(s, a)$ with the return averaged over all the samples concerning (s, a). They applied this technique to the card game DouDiZhu, and showed that it achieved state-of-the-art performance.

The main advantage of Deep Monte Carlo is that it can handle stochastic environments where the reward received by the agent is nondeterministic. This is true for Tractor because although the game is itself deterministic, the outcome of a round is nondeterministic from each player's own perspective, as they can't observe the full game. Deep Monte Carlo is unbiased in expectation as it directly approximates the true values (Sutton and Barto, 2018), despite having a high variance, which can be reduced by sampling lots of training data.

A limitation of Deep Monte Carlo is that it relies on the environment being episodic. That is, the agent's interaction with the environment must terminate after a finite number of moves. However, this is not a problem for episodic games like Go and Tractor.

3.5 Soft Actor Critic

Soft Actor Critic (Haarnoja et al., 2018) is a variant of Actor Critic that maximizes both the expected return and entropy at the same time. The benefit of maximizing entropy is that in many RL tasks, a good policy tends to be one where the agent could attain high returns while acting with randomness. In SAC, the actor learns a stochastic policy that maximizes a trade-off between the expected cumulative reward and the entropy of the policy. The entropy term encourages the actor from becoming too deterministic. By maximizing it, the policy is encouraged to take actions that are not just optimal, but also diverse, which can help it find better solutions and avoid getting stuck in local optima.

To control the degree of exploration the policy, SAC has a temperature parameter α that controls the relative importance of the expected cumulative reward and the entropy. A high temperature parameter encourages the policy to be more exploratory, while a low temperature parameter encourages the policy to be more deterministic. By adjusting the temperature parameter during training, the algorithm can find a good balance between exploration and exploitation.

A limitation of SAC is that the original implementation requires a policy network, which is not suited for Tractor. An alternative way to get the entropy could be to compute the

Q-values for each action in the legal action space, and then take a softmax and then treat it as a probability distribution.

Chapter 4

Method

4.1 RL Environment and Data Collection

To train an AI system on Tractor, it needs to have an environment to interact with. Since we have not found an existing implementation of Tractor for the purpose of Reinforcement Learning, we implement a new environment for the game from scratch and open-source it on GitHub for the benefit of future research.

As previously mentioned, Tractor is a 4 player imperfect information game consisting of sequential moves. The players sit in a circle, with their teammates sitting across them. To represent this setup formally, we identify each player with an *absolute position* being one of $\mathcal{P} = \{\texttt{NORTH}, \texttt{SOUTH}, \texttt{EAST}, \texttt{WEST}\}$, where `NORTH` and `SOUTH` are a team and `EAST` and `WEST` are a team. Players make move in the counterclockwise direction, following the order

$$\texttt{NORTH} \to \texttt{WEST} \to \texttt{SOUTH} \to \texttt{EAST} \to \texttt{NORTH}.$$

At any point during a round, each player observes a different and partial view of the complete game state S. From the players' perspectives, the absolute positions of the players are irrelevant when observing the game. Instead, what matters to them is the *relative positions* of themselves and the other 3 players, each being one of $\widetilde{\mathcal{P}} = \{\texttt{SELF}, \texttt{OPPOSITE}, \texttt{LEFT}, \texttt{RIGHT}\}$. Each absolute position establishes a different bijection from $\mathcal{P} \mapsto \widetilde{\mathcal{P}}$. For example, `NORTH` uses the following correspondences when deriving each observation from the game state:

$$\{\texttt{NORTH} \to \texttt{SELF}, \texttt{SOUTH} \to \texttt{OPPOSITE}, \texttt{EAST} \to \texttt{LEFT}, \texttt{WEST} \to \texttt{RIGHT}\}.$$

When deriving observations, we always describe positions relatively. For example, the position of the dealer is described as a relative position for each player.

During training, the `ShengJi+` plays against itself by moving around the table. On each player's turn, it pretends to be that player and makes the move $a = \text{argmax}_a Q(s, a)$ among the legal actions, with access to only the observation s that is available to that player at that time. The environment collects all moves made by all players for off-policy training, as well as all the observations drawn by the players right before those moves. The actions made in

the 4 phases of the game are stored separately because they are used to train the 4 different models. The actions made by each of the 4 player positions are also stored separately for the purpose of calculating rewards, but are combined when training each of the models. To speed up the data collection procedure, we come up with a training pipeline that uses parallel actors to run game simulations, using weights that are shared across these processes.

Algorithm 1 Actor Process of `ShengJi+`

Input: shared buffers B_D, B_K, B_C, and B_T for collecting samples for the 4 game phases; global Q networks Q_D, Q_K, Q_C, Q_T; hyperparameters
for $1, 2, \ldots$ **do**
  Initialize local buffers D_N, D_S, D_W, D_E to store moves for player North, South, West, and East respectively
  Initialize new Tractor round G with random rank $\in 2, \ldots, 14$ and random starting player $\in \{N, S, W, E, \emptyset\}$
  while round not ended **do**
    $Q \leftarrow$ one of Q_D, Q_K, Q_C, Q_T based on current phase
    $j \leftarrow$ one of N, S, W, E {Get player position}
    $\mathbf{s}_{t,j} \leftarrow$ partial observation at position j derived from current game state G
    $\mathbf{a}_{t,j} \leftarrow \begin{cases} \text{argmax}_{\mathbf{a}_t} Q(\mathbf{s}_{t,j}, \mathbf{a}_t) & \text{with prob } 1 - \epsilon \\ \text{random action} & \text{with prob } \epsilon \end{cases}$
    Run action $\mathbf{a}_{t,j}$ in environment, observe $r(\mathbf{a}_t, \mathbf{s}_t)$
    Add $(\mathbf{s}_{t,j}, \mathbf{a}_{t,j}, r_{t,j})$ to D_j
  end while
  for $j \in \{N, S, W, E\}$ **do**
    for $t = T, T-1, \ldots$ **do**
      if $\mathbf{a}_{t,j} \in D_j$ is an action in the main (trick) phase **then**
        $\tilde{r}_{t,j} = r_{t,j} + \gamma r_{t+1,j}$
      else
        $\tilde{r}_{t,j} = r_{t,j} + R_j$ {we don't discount actions in other phases}
      end if
      Add $(\mathbf{s}_{t,j}, \mathbf{a}_{t,j}, \tilde{r}_{t,j})$ to B_D, B_K, B_C, or B_T
    end for
  end for
end for

Algorithm 2 Learner Process of `ShengJi+`

Input: Global Q networks Q_D, Q_K, Q_C, and Q_T; shared buffers B_D, B_K, B_C, and B_T containing sampled episodes
for phase $p \in \{D, K, B, T\}$ **do**
  for each batch of data $(\mathbf{s}, \mathbf{a}, \mathbf{r})$ in B_p **do**
    Predict Q-values $\hat{\mathbf{r}} = Q_p(\mathbf{s}, \mathbf{a})$
    Update Q_p using MSE loss and RMSProp against target values $\mathbf{r}$
  end for
end for

4.2 State Representation

In each of the 4 phases of a round, a player has access to different amounts of information. All information is encoded into one-hot vectors. Below is the subset of information provided as input to each of the Q networks:

- **Declaration phase**: the observation includes the player's hand, the relative position of the dealer, the current dominant suit, the dominant rank, the position of the last player who declared (if any), and all trump cards revealed by the 3 other players (including declarations that are overridden). The actions are all possible declaration options, as well as the `DON'T DECLARE` action.
- **Kitty phase**: the observation is the same as in the declaration phase. The action is the ID of the chosen card, from 0 to 53. As mentioned previously, we simplify the task by choosing 1 card at a time for 8 times. The card choice is encoded as a 54-dimensional one-hot vector conditioned on the dominant rank and suit using the same ordering as shown in Figure 4.1, except with only one row.
- **Bidding phase**: the observation is the same as in the declaration phase, with additional information on the last person who made the bid. The action is all bidding options for the player, as well as the `DON'T BID` action.
- **Trick phase**: the observation includes everything from before. In addition, it contains the total number of bids occurred during the bidding phase, the points earned by the attackers, the points escaped by the defenders, the set of all unplayed cards, the moves played in the current trick, the kitty if the current player owns the kitty, the sequence of historical tricks, and the set of cards each player is publicly known to possess but have not yet played.

Instead of mapping cards to static one-hot vectors, we encode them dynamically conditioned on the dominant rank and the dominant suit, since they change the interpretation of the cards. The length of the flattened encoding is always the same (108), but the positions of the individual cards depend on the dominant rank and suit (see Figure 4.1). This dynamic

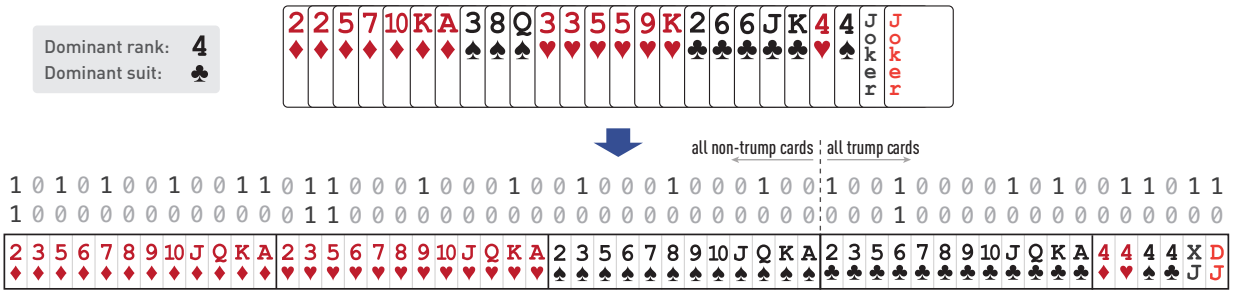


Figure 4.1: A card combination is encoded into a 54×2 one-hot matrix conditioned on the dominant rank and suit. The column represents the category and rank of the card, and the row represents card counts. Note that in this example, 3-3-5-5♥ forms a tractor, because the rank above and below the dominant rank are always treated as consecutive.

one-hot encoding also reduces the state space significantly by abstracting the dominant rank and suit. We use this method to encode card sets wherever it is applicable.

4.3 Reward Design

In Tractor, each player tries to maximize the total number of points their team earns, and it directly determines the outcome of the round. The total points a team earns in Tractor is the sum of the points it earns in each trick. Therefore, we use both the final outcome of the round and the points earned in each trick as reward signals. The final reward for a player is the number of levels their team attained in the round if they won, or the negative of the number of levels their opponents attained if their team lost. A player also earns a reward in trick t proportional to the number of points their team earned in trick t, and if his team lost points in the trick, the reward would be the negative of the number of points their opponents earned. For the first three phases of a round (declaration, kitty selection, bidding), no points are being earned yet, so the players' actions in these rounds only receive the final outcome as reward signal.

The discount rate γ influences the agent's preference between earning points now and planning to earn points in the future. A small discount also means that the agent prioritizes earning points now as opposed to later or leveling up. Although these two objectives are similar, they are not identical. For example, suppose that the attackers earned 90 points by the time that there are 3 remaining cards per player, and that one attacker has a big move that can either earn them 20 points right now, or possibly earning them 30 points or above in the last trick. In expectation, the first strategy has a higher reward, but in terms of maximizing the ranks attained, the agent should play strategy 2, because earning another 20 points is not enough to level up. In contrast, $90 + 30 = 120$ points are just enough to

level up, and it can only be achieved by the second strategy, so the expected number of ranks attained by strategy 2 is higher than that of strategy 1. Therefore, the relative reward weights for ranks attained and points scored in a trick directly affect the agent's strategy.

4.4 Architectures

In order to build a good AI system for Tractor, we consider three deep RL architectures for the main phase. The models for the pre-round phases are all supervised directly on the final reward without discount, because the trajectory lengths are mostly fixed, the players don't earn any points just yet (so no immediate reward for any actions taken in those phases), and the final reward signal is very far away. The main phase – with the largest state and action spaces among all phases – is where most of the strategic play happens, so we use this phase to run our experiments.

All 5 models use fully connected layers with ReLU activations that are optimized using RMSProp (learning rate = 0.0001) and MSE loss. The model for the trick phase uses an LSTM to encode the history of the last 15 tricks (each of which consists of 4 moves, one played by each player), and the output is concatenated with other state and action features. All agents use an epsilon-greedy exploration scheme with $\epsilon = 0.015$ for all phases. We use one GPU for training, and train all agents for 700000 games against themselves.

Deep Q Learning Agent

To train the DQN agent, we collect $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}, r_t)$ tuples for each action taken by the agent during self-play, and use them to supervise a Q-network $Q(\mathbf{s}, \mathbf{a})$ and value network $V(\mathbf{s})$. The value network is used to approximate

$$V(\mathbf{s}_t) \approx \mathbb{E}_{\mathbf{a}_t \sim \pi}[Q_\phi(\mathbf{s}_t, \mathbf{a}_t)],$$

which would otherwise be inefficient to compute using just the Q-network because the distribution of $\mathbf{a}_t$ is not fixed. Moreover, the action space is usually large, so computing $\max(\cdot)$ over actions is costly. We use the value network to supervise the Q network using a Mean Squared Error loss (d_t is whether time step t is terminal):

$$J(\phi) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \left[\frac{1}{2} \Big(r(\mathbf{s}_t, \mathbf{a}_t) + \gamma(1 - d_t) V_\theta(\mathbf{s}_{t+1}) - Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \Big)^2 \right]$$

Similarly, the value function optimizes the following loss:

$$L(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \in \mathcal{D}} \left[\frac{1}{2} \Big(Q_\phi(\mathbf{s}_t, \mathbf{a}_t) - V_\theta(\mathbf{s}_t) \Big)^2 \right]$$

Deep Monte Carlo Agent

To train the DMC agent, we supervise all 4 Q-functions using the discounted cumulative rewards. The t-th move for player j is recorded as a tuple $(\mathbf{s}_{t,j}, \mathbf{a}_{t,j}, r_{t,j})$, and at the end

of a round, the final reward R_j (which is different for the players) is incorporated into the reward for each player's actions with a discount factor γ, resulting in $(\mathbf{s}_{t,j}, \mathbf{a}_{t,j}, \tilde{r}_{t,j})$ where

$$\tilde{r}_{t,j} = r_{t,j} + \gamma^{T-t} R_j$$

for a round with T tricks in total. Then, we use the Mean Squared Error loss to fit $Q(\mathbf{s}_{t,j}, \mathbf{a}_{t,j})$ to $\tilde{r}_{t,j}$.

Maximum Entropy DQN Agent

To test out whether the maximum entropy framework is useful for Tractor, we implement a variation of the DQN agent for the main phase that adds an entropy term to the value function's training labels, like in Soft Actor-Critic, but without using a policy network. The agent still learns a Q-function which is used to extract its policy, and it is supervised using the softened value function (with the entropy term added):

$$V(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi}[Q(\mathbf{s}_t, \mathbf{a}_t) - \log \pi(\mathbf{a}_t \mid \mathbf{s}_t)]$$

In order to compute the action space entropy at state $\mathbf{s}_t$ without having a policy network to compute the action distribution from, we use the rewards $Q(\cdot, \mathbf{s}_t)$ assigned by the Q-network at $\mathbf{s}_t$, apply a Softmax, then treat it as a probability distribution and calculate the entropy from it. Finally, like in SAC, we use a temperature parameter α to weigh the entropy term against the reward term. It is initialized at 1 and gradually decreases to 0 by backpropagation. The objective function of α is

$$J(\alpha) = \mathbb{E}_{\mathbf{a}_t \sim \pi_t}[-\alpha \log \pi_t(\mathbf{a}_t \mid \mathbf{s}_t) - \alpha \bar{\mathcal{H}}]$$

where $\bar{\mathcal{H}}$ is the target entropy.

4.5 Learning Techniques

Oracle Guiding

Inspired by `Suphx` (Li et al., 2020), we implement a similar oracle guiding mechanism to drop out information about other players' private cards gradually. In Tractor, obviously each player cannot observe the hands of other players, but if they could, this information would be a great advantage. Similarly, if an AI system is trained with access to the hands of other players, it should in theory play better. However, it would be unfair for a model to always have access to this additional information, so we drop out the oracle inputs gradually using Bernoulli random variables parameterized by δ that decreases from 1 to 0 linearly over N games, to let the model gradually transition from being all-knowing to being ordinary by the end of the N-th game.

Reducing Action Space Through Sequential Moves in Sub-Action Spaces

In most RL problems with discrete action space, the standard practice involves training a network that, given the current observation $\mathbf{s}_t$, either computes a probability distribution $\pi(\cdot \mid \mathbf{s}_t)$ over the action space $\mathcal{A}$, or assigning a Q-value for each action. However, neither approach is efficient for Tractor because $|\mathcal{A}|$ could be too big to enumerate explicitly. For example, a player holding 25 cards from the same suit has an action space of more than a million moves, since playing any subset of those cards is a valid move. Supposing that $Q(\mathbf{a}_t, \mathbf{s}_t)$ takes 1 millisecond to compute, deciding on an action would take ≈ 17.7 minutes, which is too slow to be used in real-world settings. Another situation involving a large action space is that in each round, the dealer must choose 8 cards to discard from his hand of 33 cards, forming the kitty. This means there are up to $\binom{33}{8} \approx 1.39 \times 10^7$ actions to choose from, again too big to enumerate.

To work around this problem, we decompose actions that come from huge action spaces as a sequence of sub-actions from smaller action spaces whose union is equivalent to the original action. For the kitty task, we decompose it into the sub-action of selecting 1 card to discard each time and repeating this task 8 times sequentially, each time updating the player's hand after the move. For the first card, the sub-action space is now at most 33 (could be less as some cards could be pairs), and for each subsequent card, the action space gets even smaller. The total number of Q-value computations needed for the kitty phase is now bounded by $33 + 32 + 31 + \cdots + 26 = 236$ instead of 1.39×10^7.

Likewise, in the trick phase, instead of enumerating all legal moves the player could play, we decompose it into the task of choosing a pattern to play first, then deciding whether to add more patterns from the same suit. Doing so reduces the action space down to the number of individual patterns in the player's hand, which is small. As an example, suppose a player has {AKKQQ♥, 8♦}. To decide what to play, the player chooses from the following sub-action spaces:

1. {A♥, K♥, KK♥, Q♥, QQ♥, KKQQ♥, 8♦} → A♥.
2. {K♥, KK♥, Q♥, QQ♥, KKQQ♥, ∅} → KKQQ♥.
3. {∅} → ∅.

⇒ The final move is AKKQQ♥.

As shown, the pattern combination is finalized when the player chooses ∅. The Q-value of any decision from above is the Q-value of the union the card being chosen up to that point, treated as one move. In practice, this way of forming moves pattern by pattern is still slow, so we limit the maximum number of patterns per move to 3. So if the player gets to the fourth sub-action, they could only choose ∅ to finalize their pattern combination.

To investigate the value in training with pattern combination moves, we train two agents, one in single-pattern mode and one in multi-pattern mode, and then compare their performances.

Combo Move Penalty

In the real Tractor game, players often avoid playing pattern combinations unless they are certain that the move is valid (in which case, playing multiple patterns in one move is stronger than playing them individually) or they strategically use the combo move as a test to gain information about other players' cards, at the cost of potentially losing the current trick (since if any pattern in the combination is dominated by any other player, the current player is forced to play it). Among the two cases, the former occurs much more frequently, so we introduce a hyperparameter called the "combo-move penalty" that is subtracted from the player's reward for each of their combo moves that has failed. When the penalty is positive, the agent will be more careful when deciding to play a combo move because they will be penalized if they do not choose the move wisely.

4.6 Metrics and Game Modes

We use three metrics to compare the performance of two teams `A` and `B` in Tractor.

- **Win Rate**: The percentage of games won by team `A` divided by the total number of games played. Attackers win if they score at least 80 points, and otherwise the defenders win.
- **Leveling Rate**: The percentages of ranks leveled up by team `A` divided by the total number of ranks leveled up by either team. We focus on this metric as it more accurately describes the objective of Tractor than the raw Win Rate.
- **Average Attacking Point Difference (AAPD)**: The difference between the average points team `A` earns when being attackers against team `B` and the average points `B` earns when being attackers against `A`. While the LR can measure large performance differences between two models, the point difference can capture more subtle differences between two models' performances. We use this metric more when the models being compared have similar LR performance.

When simulating matches between two agents, we consider two match settings: **single-pattern mode** and **multi-pattern mode**. In single-pattern mode, the agents are not allowed to play pattern combinations (which also include single patterns). In multi-pattern mode (the regular Tractor game), the agents can play pattern combinations, with the standard rule that if there are any pattern components in the combination that fails to dominate,

then they are forced to play the smallest dominated pattern. The main purpose of comparing these two modes is to see whether training in multi-pattern mode helps to improve an agent's performance in single-pattern mode. The standard Tractor game is multi-pattern.

Chapter 5

Results

In this section, we report and compare the match statistics of various agent types and configurations to determine the best known architecture. The included level rate statistics have been repeatedly tested and are reproducible up to a $\pm 0.1\%$ error.

5.1 Discount Rate

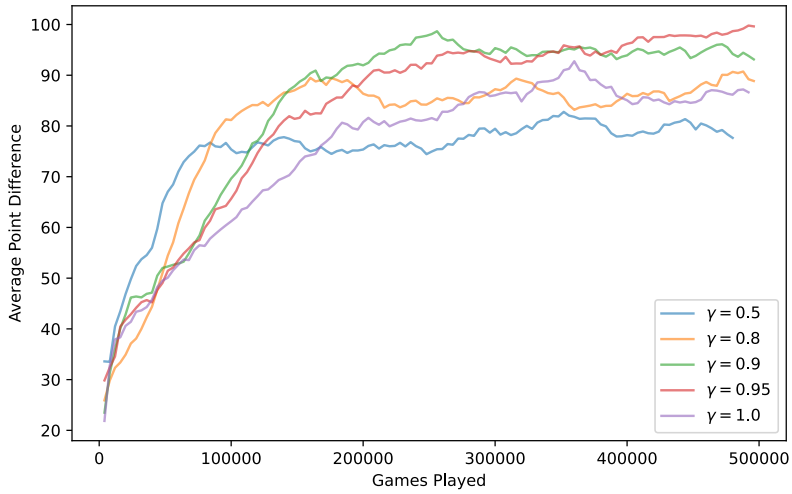


Figure 5.1: Average Attacking Point Difference learning curves for 5 different discount rates, while controlling other configurations.

Before running other experiments, we first run a preliminary test to understand the effect of the discount rate in Tractor. We run a single-pattern training process on $\gamma = \{0.5, 0.8, 0.9, 0.95, 1.0\}$, and find that the smaller the discount rate, the faster the model

learns initially, and the larger the discount rate, the longer the model learns and the more times it takes to reach its peak performance. In Figure 5.1, this pattern can be clearly seen from the AAPD curves. For $\gamma = 0.5$, the model learns very fast for the first 50000 games, but then appears to plateau at AAPD = 80. On the other hand, $\gamma = 0.95$ and $\gamma = 1.0$ appear to continue to learn even past 500000 games. This suggests that depending on the available time and computational resources, different discount rates should be used to reach the peak performance.

The pattern between the learning curves and discount rate is a reasonable one, and it explains why $\gamma = 0.95$ appear to be the best discount rate among the 5 tested choices. A small discount rate means that the model focuses on maximizing the points it earns in the next few tricks. However, a locally optimal strategy in Tractor is almost never globally optimal, because earning and protecting the kitty points necessitates long-term planning. For example, suppose there are 10 points in the kitty. Playing a large pair early might lead to 15 points gained in the current trick, but saving the pair until the end and leading the last trick using it could earn $10 \times 4 = 40$ points, a higher total return, and thus a better strategy. However, a $\gamma = 0.5$ model discounts future rewards so much that it does not see the 40 points, whereas a $\gamma = 0.95$ model would value 40 points opportunity more than earning 10 points now even if it's 20 tricks away. That is, $\gamma^{20} \times 40 \approx 14.34 > 10$.

According to this reasoning, $\gamma = 1$ should have the best performance, but empirically this has not been the case. One explanation is that a discount rate strictly less than 1 encourages the model to prioritize points it can earn now as opposed to later, because the outcomes of future tricks are uncertain. Moreover, the closer the discount factor is to 1, the higher the range and variance of target Q-values the Q network needs to fit, and hence the model could struggle to converge. Thorugh this preliminary test, we decide to use 0.95 as the discount rate for all of our other experiments.

5.2 Comparison of DMC and DQN Methods

To find out which deep reinforcement learning approach is most effective for Tractor, we set up two agents with identical configurations except one uses Deep Q-Learning for training the main network and the other uses Deep Monte Carlo for training the main network. We match them both against the Random baseline and with each other for a total of 10000 games each. The result suggests that DMC is significantly better.

There are many possible explanations as to why this is the case. One is that DMC is naturally suited for tasks with high uncertainty, like Tractor. DQN supervises the agent based on the model's prediction of what happens next in the game and adds the current step reward to the maximum expected reward among all actions that can be taken at the next state. Although rewards are indeed dense in Tractor (since points can be earned in each trick), they are a result of the combined actions of the other players, which are unknown to the current player ahead of time. The same move could have led to a high reward this time, but a large penalty the next time. Due to information imperfect nature of Tractor, an

	DMC	**DQN**	**Random**
DMC	-	69.5% 59.7%	**96.7% 97.4%**
DQN	30.5% 40.3%	-	93.9% 94.2%
Random	3.3% 2.6%	6.1% 5.8%	-

Table 5.1: Leveling Rates of DMC, DQN, and Random in single and multi-pattern mode respectively.

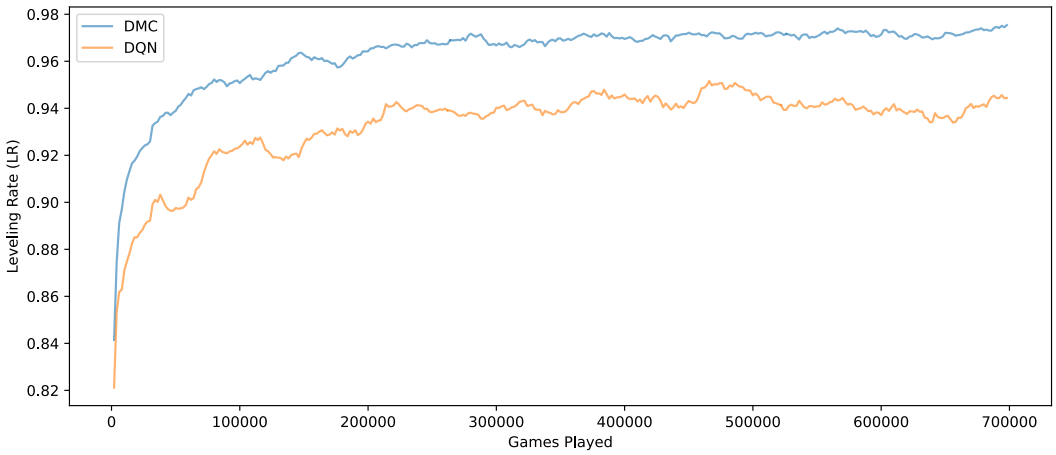


Figure 5.2: Deep Monte Carlo vs Deep Q Learning in terms of level rate.

agent's action in the current trick is very weakly correlated with what reward they will get in the next trick, so supervising the Q network using a past version of itself is less effective than supervising on the actual cumulative reward. Furthermore, DQN suffers from overestimation of Q-values, especially over long episodes. On the other hand, DMC is suited for situations with high uncertainty because it is "averaging" over cumulative rewards and synthesizing trajectories that are similar, thus being unbiased.

Another way to think about this difference is that Tractor is a game where the value variance of observations during most of the game is almost as big as the value variance of observations in the beginning (sometimes you can't predict who will win until the very end). However, the Bellman equations behind DQN basically assume that the agent has access to more information one step into the future, which can be used to reinforce its understanding of the current state. This might be true for control tasks like learning to run (knowing the next state helps the agent evaluate whether the action taken is good), but for highly stochastic games like Tractor, there is not much to be gained by seeing the next state because it's going to be different each time. The DQN method accumulates the estimation errors between time steps, so when the agent predicts Q values for observations that are early on in the game,

they could be far away from reality. The DMC agent seems better in this sense, as it learns moves that are statistically optimal and makes no assumptions about future states.

5.3 Dynamic vs. Static Card Encoding

As shown in 4.1, we propose a dynamic card encoding method for Tractor that maps a set of cards to a length-108 binary tensor in a way that depends on the dominant rank and suit. More specifically, the suits are ordered such that the dominant suit is always encoded last. The motivation of this encoding scheme is to help the model better generalize its understanding of a hand to different dominant ranks and suits. In the dynamic encoding mode, all observations related to sets of cards, such as the player's current hand, the kitty (if the player placed the kitty), historical cards being played, cards played by players in the current trick, and the cards revealed cards by each player, use this encoding scheme.

To determine its effectiveness, we compare it with a static baseline card encoding scheme that always encodes cards into a fixed 108 bits-long binary vector according to the canonical suit order ♦, ♣, ♥, ♠, followed by Black and Color Jokers. We match each model with the random baseline for 10000 games, then matched them against each other for another 10000, in both single-pattern and multi-pattern modes. Results are shown in table 5.7.

	Dynamic Encoding	**Static Encoding**	**Random**
Dynamic Encoding	-	51.6% 47.9%	**96.7% 97.4%**
Static Encoding	48.4% 52.1%	-	96.4% 97.1%
Random	3.3% 2.6%	3.6% 2.9%	-

Table 5.2: Leveling rates of dynamic vs static encoding schemes in single-pattern and multi-pattern modes.

The LR statistics suggest that the performance difference between the two encoding schemes is very small, which implies that at least to some extent, deep neural networks are naturally able to reason about a hand of cards conditioned on the dynamic trump and suit. This is a meaningful observation because unlike many card games, in Tractor, the same card can have different meanings depending on the game context. The finding that neural networks do not require special handling in the encoding of the state/observation suggests that any complete binary representation could be a good starting point for encoding the states of such games.

The average attacking points difference reflects more subtle differences between the two card encoding schemes, and we can see from figure 5.3 that after training for about 250000 games, the dynamic encoding agent consistently establishes a larger point difference when played against the random model than the static encoding agent does. Although point difference does not necessarily translate into higher leveling rate, it does demonstrate an

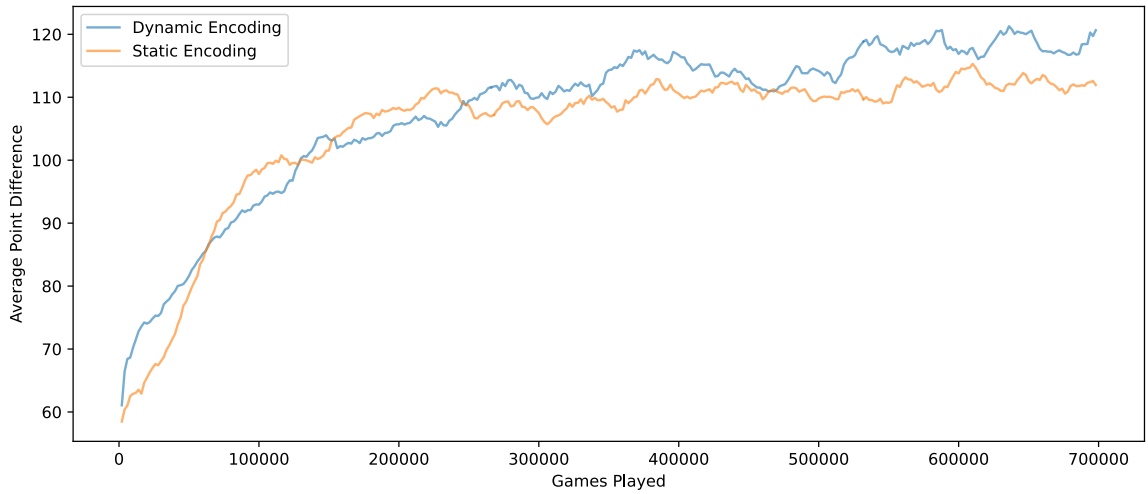


Figure 5.3: Average Attacking Points Difference of dynamic and static card encoding.

understanding of point acquisition strategies, which is at the core of the game tactics of Tractor.

5.4 Maximum Entropy Signal

To determine if introducing an maximum entropy signal could encourage the Tractor agent to explore more diverse policies, we run a modified DQN agent that uses a softened value network to supervise the Q network. However, through our preliminary experiments, we find that adding the entropy signal does not lead to a statistically significant improvement over the original DQN agent. The main reason why this could be the case is that the environment of Tractor is very different from typical RL environments. Maximum entropy encourages the agent to enter states where there are many possible and equally likely actions to choose from. However, in Tractor, a player often has more ambiguous choices when being dominated by another player, in which case, he does not have a strong preference between his choices of actions because none of them could dominate the other player(s). On the other hand, a player is likely to be in a good situation when there are not too many actions to choose from. This is because a player can only dominate another player if he can match the pattern that the other player played, and if he has a match, then the rules of the game dictates that he must play one of the matches. Therefore, maximizing the entropy in Tractor could potentially lead to the agent preferring situations where it is dominated, which is a poor strategy for Tractor.

5.5 Oracle Guiding

Oracle guiding is a general technique for imperfect information games introduced by (Li et al., 2020) for Japanese Mahjong. To determine if it is effective for Tractor, we implement two agents with and without oracle guiding (keeping other configurations identical), and compare their performance against each other as well as against the random baseline. For the agent with oracle guiding, we choose $N = 50000$.

	Without Oracle	**With Oracle**	**Random**
Without Oracle	-	65.5% 65.8%	**97.2% 96.7%**
With Oracle	34.5% 34.2%	-	93.4% 93.3%
Random	2.8% 3.3%	6.6% 6.7%	-

Table 5.3: Leveling rates with and without oracle guiding.

Contrary to expectation, we find that oracle guiding adversely affects the performance of `ShengJi+`. The reason this result could initially be a bit surprising is that the information of the game state available to the oracle-guided model is always greater than or equal to that of the unguided model. Specifically, the oracle-guided model has strictly more information for the first N games, and has the same level of information as the unguided one for the remaining training session.

One possible explanation is that the type of moves that are optimal to play given oracle access to other players' cards in Tractor belongs to a different distribution than those which are good strategies under normal circumstances. One example are moves that have both high risks and high rewards. Without knowing other players' cards, one cannot assess the risk of such a move so it would not be wise to play it straightaway. However, with oracle access, the agent could tell whether there is any actual risk in this high reward move, and if not, it would be optimal to play it instead of any low-risk low-reward move. A specific example could be leading a trick with a `10` in a non-trump suit. In real Tractor games, players almost never play a point card to lead a trick except if all bigger cards have been played, because otherwise this move is betting on the teammate having the biggest remaining card(s) in the suit among the 3 players. If neither `A` (the biggest card) has been played yet, for instance, then `10` could only be optimal if the teammate has the pair of `A`, which is a low probability event. But, if this is indeed true and the current player knows this information, then playing `10` is a good move, as it would be a no-risk but high-reward move.

In short, we deduce that the oracle-guided agent may have initially started out learning strategies that cannot be sustained once it loses the oracle access. This may have forced the agent to have to adapt its strategies after the first N games have passed, taking more time to learn in total than the unguided agent. Additionally, oracle access increases the state representation by 3×108 to encode other players' hands, which is a burden in terms of network size and computational efficiency.

5.6 Knowledge Transfer Between Game Modes

As described in Section 4.6, we consider two game modes when training `ShengJi+`: single-pattern (SP) mode and multi-pattern (MP) mode. MP mode can be considered as a superset of SP mode because both the state and action space of SP mode are a subset of those of MP mode. To compare training in the MP game environment with training in the SP environment, we train a baseline model that only plays with itself in SP mode, again for 700000 games. Then we match the baseline model against an agent trained on MP mode but otherwise has identical configurations.

	MP Agent	**SP Agent**	**Random**
MP Agent	-	51.4% 52.0%	**96.7% 97.4%**
SP Agent	48.6% 48.0%	-	96.5% 96.6%
Random	3.3% 2.6%	3.5% 3.4%	-

Table 5.4: Leveling rates of agent trained on single-pattern mode vs on multi-pattern mode.

The match statistics suggest that the agent trained in MP mode is more competitive than the SP agent in *both* game modes. This is unexpected because the SP agent is trained in the SP game environment, whereas the MP agent is trained on a slightly different environment, so one would naturally expect the SP agent to be better when playing in its own environment. However, the MP agent outperforms it even though it is not allowed to make multi-pattern moves (which presumably are a part of the strategies it learned during training) in SP mode.

We believe that the likely cause of this phenomenon is that by being allowed to make multi-pattern moves during training, the MP agent acquires additional knowledge about the game that could generalize to single-pattern mode. A fact supporting this hypothesis is that a pattern combination move is only valid to play if all of the individual pattern components are dominating the other players, so a good MP agent would need to have an understanding of single pattern moves first before reasoning about how these patterns can be combined. Such information synthesis is also made possible by the smoothness of the neural Q networks. If the agent plays the combination `A`♥`K`♥ and receives a positive reward, then the Q-values for the individual moves `A`♥ and `K`♥ may also be increased as a result of this training example. Conversely, if the agent knows that `A`♥ and `K`♥ are high value moves, then its valuation of the combo move `A`♥`K`♥ should also be high. This information sharing mechanism would not exist if the agent instead learned a policy network that computes distributions over actions, because then `A`♥, `K`♥ and `A`♥`K`♥ would treated as three distinct actions.

If knowledge transfer does happen from MP mode to SP mode, then it potentially means that training an RL model on a well-chosen augmented environment could lead to improvements in the original environment. For the Tractor case, this augmented environment happens to be defined already as part of the game rules, and the augmented legal action space of MP mode is just the powerset of the legal action space of SP mode for each suit. It would

be interesting to investigate how such augmented environments should be defined for other games and whether training on those environments helps the agent to improve in the original environments.

5.7 Combo Move Penalty

A common understanding of Tractor among human players is that combo moves are high-risk high-reward and should be played with caution. To incorporate this knowledge into `ShengJi+`, we use the combo move penalty p to adjust how risk-averse the agent is at playing combo moves. We train two agents, one without the penalty and one with $p = 0.15$. When the agent plays a failed combo move, they receive negative reward for any points lost in the current trick as a result of it, as well as a $-p$ reward to penalize the mistake further.

	$p = 0$	$p = 0.15$	**Random**
$p = 0$	-	48.3% 51.3%	96.7% 97.4%
$p = 0.15$	51.7% 48.7%	-	97.2% 96.7%
Random	3.3% 2.6%	2.8% 3.3%	-

Table 5.5: Leveling rates with combo move penalty of 0 and 0.15.

The match statistics suggest that a higher combo move penalty increases the model's performance in single-pattern mode but decreases its performance in multi-pattern mode. This is not surprising given that the $p = 0.15$ agent is discouraged from playing combo moves relative to the $p = 0$ agent, so it spent more time playing and learning about single-pattern moves than combo moves. On the other hand, the $p = 0$ agent could freely explore combo moves without cost, so it may have developed more intuition about those moves during training. It is possible that there exists a midpoint value $0 < p < 0.15$ that retains the benefit of both agents, but this has not been tested in our experiments.

5.8 Best Configuration

From our experiments, we define `ShengJi+` to be the configuration that maximizes the leveling rate in multi-pattern mode (because this is the standard Tractor game). It uses Deep Monte Carlo for all 4 phase models, dynamic card encodings, no oracle guiding, no entropy signal, and no combo move penalty. The agent uses epsilon greedy with $\epsilon = 0.015$ and uses a discount rate of $\gamma = 0.95$ for the main phase. We train this agent for 1.2 million games and it achieves a leveling rate of 97.6% and an average attacking point difference of 124.74 against the random baseline in the standard (multi-pattern) mode of Tractor.

Chapter 6

Conclusion

This work presents an AI system for the 2v2 trick-taking card game Tractor, which has not yet been studied in the literature either as a game or as a reinforcement learning problem. Inspired by `DouZero` (Zha et al., 2021), `ShengJi+` uses Deep Monte Carlo and we show that it is an overall efficient solution for Tractor when dealing with its large and variable state and action spaces. Through matches against the random baseline, we obtain a leveling rate of 97.6%. We hope that the techniques we introduce, such as dynamic card encoding and sub-action reduction, could serve as a baseline for future research into this game and provide insight into how better AI systems for Tractor and other related card games can be designed. We open-source the code[1] for both `ShengJi+` and the Tractor game environment which we implemented, including instructions on how to download and play with the pre-trained model weights.

Through observing `ShengJi+` playing against itself and against humans in interactive mode, we believe that the current system already demonstrates a lot of intelligent behavior that is similar to how human players reason about the game in the simplified single-pattern mode. Some of our case studies and detailed analysis of `ShengJi+`'s tactics can be found in the appendix. For multi-pattern Tractor, however, we feel that there is still a lot of room for improvement, because the agent still frequently plays pattern combinations that fail. The strategies of Tractor are complicated, and it could take a lot more games for a self-play system to truly master the full game. Some future research directions include ways to supervise the models on expert data, better reward design, better network architecture, better ways to reduce/abstract the state and action space, better ways to train the agent to learn pattern combinations, etc.

[1] https://github.com/themoon2000/shengji_plus

Bibliography

C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

N. Brown and T. Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

J. Li, S. Koyamada, Q. Ye, G. Liu, C. Wang, R. Yang, L. Zhao, T. Qin, T.-Y. Liu, and H.-W. Hon. Suphx: Mastering mahjong with deep reinforcement learning. *arXiv preprint arXiv:2003.13590*, 2020.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

N. Shi, R. Li, and S. Youran. Scrofazero: Mastering trick-taking poker game gongzhu by deep reinforcement learning. *arXiv preprint arXiv:2102.07495*, 2021.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.

D. Zha, J. Xie, W. Ma, S. Zhang, X. Lian, X. Hu, and J. Liu. Douzero: Mastering doudizhu with self-play deep reinforcement learning. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 12333–12344. PMLR, 18–24 Jul 2021. URL http://proceedings.mlr.press/v139/zha21a.html.

Y. Zhao, J. Zhao, X. Hu, W. Zhou, and H. Li. Douzero+: Improving doudizhu ai by opponent modeling and coach-guided learning. *arXiv preprint arXiv:2204.02558*, 2022.

Chapter 7

Appendix

7.1 Case Studies

Tractor is a complicated game, and even human experts may not always play the optimal moves because of hidden information. To qualitatively evaluate the policy that `ShengJi+` uses, we conduct case studies for each of the 4 game phases of Tractor to see if the choices that the model makes are similar to what human players would make. The kitty and bidding phase are combined into one section as the strategies for the kitty phase depends on the bidding phase. For each phase, we provide an analysis of the moves we see `ShengJi+` make. We provide the seeds for the game rounds we analyze, so that they can be reproduced. The checkpoint used for this analysis is 1144000; Q-values may differ for other checkpoints.

Declaration Phase

In the declaration phase, players are dealt 25 cards in counterclockwise order from 2 decks, leaving 8 on the table, which are reserved for the dealer. As cards are dealt, players who possess cards in the dominant rank can reveal their card(s) to declare the trump suit of the current round. Using a pair of dominant rank cards can override an existing 1-card declaration, and pairs of Jokers can override all (Red Jokers are bigger than Black Jokers). Since it is advantageous to have a large number of trump cards, a good strategy is to reveal the suit which the player is stronger in, if the player happens to have the dominant rank card of that suit. But it's not always easy to tell which suits are strong until most cards have been dealt, so a player who declares early faces the risk of their declared suit not being strong anymore after receiving all 25 cards. But if the player declares late, then some other player could declare an unfavorable trump suit before he takes action. If there is a bidding phase, then overriding declarations are not as common, since bidding can achieve the same result of overriding the trump suit, but in addition, the bidding can swap cards with the kitty, hence obtaining a stronger hand too.

To examine how `ShengJi+` reasons about declaration, we run a random round (random seed = 1) with `ShengJi+` playing with itself, and observe how it makes declarations. In this

round, the dominant rank is 4, and the dealer is predetermined to be North. Upon receiving the 5th card, West becomes the first player to gain the ability to make a declaration of 4♦ with his current hand:

QT4♦; T8♠

Declaring ♦ would mean that QT4♦ become 3 trumps, and any diamond card he receives also becomes a trump. `ShengJi+` computes the following Q-values:

Action	Q-value	Probability
DON'T DECLARE	0.9426	0.5302
DECLARE 4♦	0.8215	0.4698

Based on these values, we can tell that `ShengJi+` is optimistic about the outcome of the game, as it predicts positive Q-values for both of West's actions. This is logical, because 4♦ is a dominant rank card, which is the biggest card except Jokers in this round. West has 1 such card among the 5 cards he drew, and the other 4 cards are relatively big in rank, so overall, West's current hand is above average.

Another observation we can make from this is that West prefers not declaring right away at this point. Most human players would do the same, because 5 cards give too little information on how much advantage 4♦ will end up having by the time all 25 cards are drawn. Specifically, it's highly uncertain whether diamond will end up being a strong suit for West, despite 2 are already drawn (4s are always trump in this round, so declaring ♦ brings 2 additional trumps to West). Therefore, we consider West's move to be rational in this case.

Let us now examine a case where a player makes a declaration and evaluate the rationality of the move. Later in this same round (seed = 1), East makes a declaration of 4♣. When he makes this move, his current hand is:

93♦; QJ84♣; J42♥; J642♠; DJ

Based on East's current hand, we observe that he has the option to declare 4♣, 4♥, or 4♠, in addition to not declaring. `ShengJi+` predicts the following Q-values for East:

Action	Q-value	Probability
DECLARE 4♣	1.1649	0.2642
DON'T DECLARE	1.1482	0.2598
DECLARE 4♥	1.1342	0.2562
DECLARE 4♠	0.9816	0.2199

In this case, 4♣ is indeed the best declaration option for East, because East has 4 total club cards, making it one of his strongest suits so far. Also, relative to spade, the average rank of his club cards are bigger, so it makes sense that East prefers to declare ♣. The

ranking of this move above `DON'T DECLARE` is also understandable, because the majority of the cards have already been drawn, so from East's perspective, waiting longer to decide what suit to declare would create the risk of allowing some other player to declare a suit that's not favorable for him. Nevertheless, `DON'T DECLARE` is ranked 2nd because it could also be a good strategy in this particular scenario. East should be rather indifferent between clubs and spades, and drawing a card in either suit helps break the tie, and his suit counts are not terribly uneven, so he could afford to wait to get one or two more cards before making a more informed decision. The only irrational Q-value assignment is 4♠, which should be higher than 4♥ because East currently has one more spade card. However, since only the argmax action is chosen, those bottom Q-values have no effect on the move carried out by East.

In both the not declaring and declaring case, we see that overall, `ShengJi+` is being logical and consistent, and produces Q-values that are similar to what humans would think in the same situation. We also observe that the action's Q-values may not be entirely sorted by the actions' optimality from a human player's perspective, because an agent only receives feedback on the actions it actually took.

Kitty and Bidding Phase

In the bidding phase, players who possess pairs of identical dominant rank cards or Jokers can take the current kitty and put back 8 cards of their choice. The two main purposes of doing the bid are to acquire a stronger hand and to manipulate the kitty points. The kitty is revealed at the end of the round, and if the attackers happen to win the last trick, they receive a certain multiple of the total points in the kitty, depending on how they won the trick. Therefore, a player would place the kitty differently depending on their role and their perceived chances of winning the last trick. If the bidder is a defender, then he would be cautious at putting point cards in the kitty, because those points face the risk of being earned by his opponents with at least a 2x multiplier, and if he is an attacker, then he would usually place more points in the kitty, especially if he thinks his hand is strong, because from his perspective, every point he puts down now can be eventually earned back by 2x or more. An attacker bidder's strategy could get even more tactical. If he thinks that he has little chance of winning the last trick, he could put fewer points in the kitty, but trick the defenders into thinking otherwise and saving (or wasting) their best cards until the end to protect a pointless kitty while the attackers take advantage and earn points early in the round.

To see how `ShengJi+` makes decisions regarding bidding and kitty selection, we run it on a random round (seed 14) against itself. In this round, the dominant rank is 3, the dealer is assigned to South, and the dominant suit is currently ♣ as declared by East. South finishes placing the kitty and now its East's turn to decide if he wants to bid. East's hand is

KQ98♦; JT92♣; K9972♥; AQT99842♠; 33♠ 3♣ DJ

As we can see, East can either choose not to bid, or bid using 33♠. `ShengJi+` makes the following Q-value estimates:

Action	Q-value	Probability
`BID USING` 33♠	2.1819	0.804
`DON'T BID`	0.7704	0.196

Obviously, `ShengJi+` has a strong preference for bidding. In fact, any reasonable human player would make the same decision, because East's spade suit is much stronger than his club suit, so he definitely wants spade to become the trump suit. In addition, spade is the largest suit among the 4, so bidding using it ensures that no one else can bid using another suit after him. This is a massive advantage for East, because he does not need to worry that some other suit will become the trump suit.

East makes the bid and grabs the kitty placed by South, resulting in a temporary hand of 33 cards, from which he must pick 8 to discard.

In the table below, we show the Q-value and probability of the top 5 and bottom 5 actions `ShengJi+` infers for East's "augmented" hand. The Q-value of a card can be considered as the expected number of ranks that `ShengJi+` thinks can be won if the card is discarded into the kitty:

Action	Q-value	Probability
10♣	2.1065	0.075
10♣	1.951	0.064
K♦	1.8693	0.059
8♦	1.6967	0.045
K♥	1.525	0.0421
	...	
4♠	1.0418	0.0259
3♣	1.0271	0.0256
2♠	0.9958	0.0248
DJ	0.9684	0.0241
3♠	0.9589	0.0239

These Q-values suggest that `ShengJi+` indeed has a basic understanding of the bidding phase in a way that's similar to what human players think. In particular, in this situation, `ShengJi+` understands its role as an attacker and places high probability to point cards (10, K). It also knows that the most valuable cards to a player are the Jokers and the dominant rank cards, followed by other trump cards. Although there exists a more optimal ordering of the cards (e.g. the 2♠ should receive a bigger Q-value than 4♠), `ShengJi+` does get the rough ordering right, which is impressive given that it learned the ranks of the cards completely through training. When the round is played out, East's team won by 4 levels, totaling 220 points, which is a huge win. Among those points, $45 \times 4 = 180$ points are

extracted from the kitty, which is purposely placed by East. East is able to multiply kitty points by 4x because he manages to dominate the final trick with his pair of 3♠.

However, `ShengJi+` does not seem to exhibit the strategy of getting rid of a suit completely or almost completely except Aces. Such a strategy is common among human players, because having no cards from a suit gives a player advantage when other players play the suit (the player can ruff the trick). It is possible that with more training, `ShengJi+` could acquire this strategy when placing the kitty, but we also suspect that because we let `ShengJi+` choose one card to discard at a time, it is unable to plan ahead and choose the optimal 8-card subset. Or perhaps getting a suit completely is a very unlikely trajectory to occur naturally through exploration that the agent never even learned that this strategy is good. We hope that future research into Tractor can address this limitation of `ShengJi+`.

Finally, we acknowledge that our current state space representation for the kitty phase ignores information about which cards came from the original kitty. In actual Tractor, as the bidder takes the kitty and puts back 8 cards, he can use the original kitty to infer the hand and suit strengths of the last kitty owner. For example, if the kitty does not contain any clubs, then the player can infer that the last kitty owner still has clubs in his hand. On the other hand, if the original kitty contains a wide range of clubs, then the player can infer that the kitty owner probably got rid of all his clubs, meaning that it would be dangerous to play clubs during the trick phase. We encourage other researchers to explore ways to incorporate this additional information into the state space of the kitty phase.

Main (Trick) Phase

In the trick phase, the goal of the attackers is to earn as many points as possible and the goal of the defenders is to prevent the other team from earning points (called escaping). In each trick, each player has to play cards of an equal number, and the winner takes all points for his team. Therefore, one common collaboration strategy is that one player plays a large move (something guaranteed or very likely to be unbeatable), and his teammate plays point cards, so that they earn the points. In the following example, we simulate a random round of `ShengJi+` playing against itself (seed = 6), and observe the moves of the players. The dominant rank is 14, the dealer is West, and the dominant suit is none. That is, only Aces and Jokers are trumps in this round.

Here are the initial cards of the players, after North is done with placing the kitty:

Player	Hand
North	AAQQ953♦; AQT442♣; AQ97433♥; AJJT♠; DJ
West	T986532♦; KKQT♣; QT976♥; AK9976433♠
South	KJ864♦; AJ986532♣; AJ882♥; KQ654♠; XJ, DJ
East	KT42♦; 9877653♣; KKJT554♥; QT8872♠; XJ

Below is the information for the first 10 tricks:

Trick #	Trick Leader	Player 1	Player 2	Player 3	Player 4
1	North	QQ♦	62♦	64♦	42♦
2	North	JJ♠	33♠	54♠	88♠
3	North	33♥	76♥	88♥	KK♥
4	East	55♥	74♥	T9♥	J2♥
5	East	K♦	3♦	10♦	8♦
6	East	77♣	44♣	KK♣	32♣
7	West	A♠	XJ	XJ	A♣
8	South	A♣	10♦	A♠	7♠
9	South	K♠	2♠	10♠	6♠
10	South	6♠	10♠	A♦	4♠

The trick phase involves reasoning over the most complicated state and action spaces, and from the moves `ShengJi+` choose, we can see that although some moves are not optimal (underlined), there is logic behind its decisions. Below, we comment on the moves that `ShengJi` makes in each trick, and discuss the possible reasoning behind those moves:

T1. North begins the first trick with a pair of Q♦. This is a very good move based on his hand because this is the largest non-trump pattern he can play. Since Aces are trumps in this round, Kings are the largest non-trump cards, but North has none, so the best he can play is this move. The other players follow the move with two single diamonds. West and East play optimally because they are not giving away any points in this trick. South's decision to not play the K♦ is also justifiable because K♦ is the largest diamond card in this round, so more value could be derived from this card by saving it until later.

T2. North continues to play optimally by playing the next largest non-trump pair he has, which is JJ♠. West has a choice of playing 99♠ or 33♠, but since neither pair could beat JJ♠, he chooses the smaller pair, 33♠. East has no option but to play 88♠. South has two point cards, K♠ and 5♠, and chooses to play 5♠, which is definitely a good choice. However, South chooses not to play K♠. This is justifiable for the same reason as South's last move.

T3. North leads the trick with 33♥. This is actually the optimal move in this situation. Although North also has a pair of 4♣, 33♥ is statistically the better choice here because North only has 2 non-trump club cards (Q♣, 10♣) above rank 4, but 4 non-trump heart cards (Q♥, 9♥, 7♥, 4♥) above rank 3, so 33♥ has a slightly smaller chance to be dominated than 44♣.

It turns out that two other players beat North. Among them, East happens to have the larger pair (KK♥) so his team gains 20 points. South has no choice but to play 88♥. West also played optimally here by not playing any point cards, as he is the

second player in this trick and it would be irrational for him to bet that East has the largest heart pair among all 4 players.

T4. East continues to play a pair of hearts (55♥). This is now a low-risk high-reward move for East because all players have already played 2 heart cards (a pair of they have one), so the chance that someone has a second heart pair is small. North reacts optimally by playing the two smallest pointless cards (74♥). West also plays the point card 10♥ under the reasonable assumption that South does not have a second pair of hearts. Indeed South does, and he plays J2♥, the last two non-trump hearts he has.

T5. East plays K♦, the largest non-trump diamond card. This is an excellent move, and is in fact the single most optimal move East should play at this time, because this is the only move that is guaranteed to dominate the trick, guaranteed to earn 10 points for his team, and an opportunity for his teammate to play a point card. North and South respond by playing their smallest pointless cards. West plays optimally by playing 10♦, adding another 10 points to his team in this trick.

T6. East plays 77♣, the only pair remaining in his hands. This is the optimal move for him, because this is the only pattern he could play that has a reasonable chance of dominating the trick. North has to play 44♣. West happens to have KK♣ and dominates the trick while earning his team 20 points. South has no club pairs, so he plays his two smallest pointless cards, again an optimal move.

T7. West plays a trump card, A♠. This is also West's only trump card. As a reminder, the only trump cards in this round are Aces and Jokers. West's move is a good one because his only trump card is one of the smallest trump cards, so unless he plays it first, his ♠ will be useless. Playing it first means that only Jokers can dominate it. But there are only 4 Jokers in total, so seeing the result of this trick can help West gauge the relative trump strengths of the other players and potentially eliminate a few Jokers.

South responds optimally by playing a Black Joker. This is better than playing an Ace because doing that wastes the Ace (it would be dominated by West's Ace). This is also better than playing the Red Joker because the Red Joker is a bigger card that should be saved until later (e.g. to protect kitty points) and not wasted on a trick that doesn't have any point cards involved. Additionally, playing the Black Joker means that if someone dominates it using a Red Joker, then South has the only remaining Red Joker, which can be a great advantage. Therefore, South plays the Black Joker to dominate West at the minimal cost.

East has no choice but to play a Black Joker. Doing so reveals to the other players that East has no Ace (because East would want to play the smallest single trump card). However, from West's perspective, East could still have one or more Red Jokers.

North responds optimally by playing an Ace. He has no incentive to waste a Red Joker to dominate his teammate and for no points.

T8. South continues to play a trump card. This is a good move because South has an above-average number of trump cards (the average is 3 in this round), so some players will run out of trump cards before he does. Playing another trump card forces all other players to play one as well, and by doing so, South can learn the strengths of the other players' trump cards. Indeed, both West and East run out of trump cards in this trick. East plays a 10♦. This is not irrational, because from East's perspective, both Red Jokers have not been played so his teammate West could potentially have a Red Joker and earn this point card. However, West has none. As soon as this trick is over, it's clear to all players that North and South possess all of the 6 remaining trump cards.

T9. South plays a K♠. Again, this is the largest spade card, so it's an excellent move that both dominate the trick and escapes 10 points for his team. East and West both react optimally by playing pointless spade cards. North plays a 10♠, which helps his team escape 10 points but this also happens to be North's last spade card.

T10. South plays another spade card 6♠. This is not unreasonable because at this point, South has played all good patterns in his hand, so he has no choice but to play a small single card (note that he does not want to play a trump card because only him and his teammate have trump cards, so playing a trump card will only consume their team's trump cards with no damage to their opponents). East seizes the opportunity to play a point card. This is a logical move because East knows that currently there is still one K♠ out there. South just played one in the last trick, so South can't possible have the other K♠. North also probably doesn't have it, because North is the dealer and would have played it at the beginning of the round if he had the K♠. So, he reasonably deduces that his teammate West must have the K♠. If this is true, then he expects that by playing the 10♠, his team will earn 20 points in this trick. However, North in fact emptied out his spades, so he ruffs the trick using a single Ace, which is the optimal move. By ruffing this trick, North secures the 10 points from being earned by the attackers. West, having observed the ruff move, plays a pointless spade card instead, again the optimal choice.

Based on our evaluation of `ShengJi+`'s behavior in this round and other rounds, we believe that `ShengJi+` is able to consistently make rational and (mostly) optimal moves in the main phase of Tractor. Occasionally, we may observe a move where we think there exists an alternative move that is provably better (e.g. in trick 7 from the example above, West could have played K♠ first before playing A♠), but even in those cases, the decisions of `ShengJi+` are interpretable. We very rarely see `ShengJi+` making unexplainable and severe mistakes while playing.

7.2 Pattern Matching and Ruffing Rules of Tractor

The rules of Tractor in the trick phase are complicated to understand and even more complicated to implement, and in this section we provide a few examples to illustrate the rules of pattern matching. In each trick, a player will lead the trick by playing a combination of patterns from a suit. If all players accept the patterns, they go in order and play cards that match the pattern to the best of their ability. By "best of their ability" we mean that the players try to play an equal number of cards that match each pattern from the longest (i.e. tractor) to the shortest (i.e. single). If at some point they can no longer perfectly match the pattern, they can choose anything to play for the remaining cards.

We will illustrate a few cases by considering only two players. The rules for 4 players are the same.

Player	Hand (non-trump)	Hand (trump)
#1	AQQJJTT9♦; 862♣; AAK♠	KJ9733♥ 5♣ 55♥ XJ XJ
#2	AKK86644♦; AQQ443♣; KJ73♠	ATT72♥ 5♠ DJ

- **Single Pattern 1: ♠ Single**

  Suppose that player 1 moves first, and he plays K♠. This pattern requires player 2 to respond with any single ♠ card. The possible actions player 2 can take are {K♠, J♠, 7♠, 3♠}.

- **Single Pattern 2: Trump Pair**

  Suppose player 1 plays 33♥. This is a pair of trump cards, so player 2 must choose a trump pair to play if he has one. Since player 2 only has a pair of 10♥, his action space is {TT♥}. Playing this action beats player 1.

- **Single Pattern 3: ♣ Pair**

  Suppose player 2 moves first and plays QQ♣. Player 1 cannot perfectly match the pattern because he doesn't have a pair of clubs, but can match the suit because he has at least 2 clubs, so he can freely choose any two to play. The possible actions are {62♣, 82♣, and 86♣}.

- **Single Pattern 3: ♦ 2-Tractor**

  Suppose player 2 moves first and plays 6644♦. This is a tractor because the dominant rank is 5, so 4 and 6 are consecutive. Player 1 must respond by playing a heart tractor of length 2. In fact, player 1 has a tractor of length 3 (QQJJTT), which gives him two choices to play: {QQJJ♦, JJTT♦}. Either action beats player 2.

- **Single Pattern 4: ♦ 3-Tractor**

Suppose player 1 moves first and plays QQJJTT♦. This is a 3-tractor, and player 2 must try to match the pattern as best as he can. Of course, he doesn't have any 3-tractors, but he has a 2-tractor, so he has to play it. That matches 4 out of the 6 cards in the pattern. For the 2 remaining cards, player 2 must play a pair if diamonds if he has one. He does and it's a pair of K♦. So player 2's action space is {KK6644♦}. It does not beat player 1 as it's not a perfect pattern match.

- **Single Pattern 5: Trump 2-Tractor**

  Suppose player 1 plays 55♥ XJ XJ. This is a trump 2-tractor. Player 2 needs to respond by playing a trump tractor. Since he doesn't have any, he instead needs to play 2 trump pairs. But he only has one, so he must play TT♥, and in addition, he can play any two trumps. The possible actions are {TT72♥, ATT2♥, ATT7♥, ATT♥ 5♠, TT7♥ 5♠, TT2♥ 5♠, ATT♥ DJ, TT7♥ DJ, TT2♥ DJ, TT♥ 5♠ DJ}. As you can see, the (legal) action space can easily go up when the move consists of many cards.

- **Combo Pattern 1: ♠ Pair + Single**

  For the next few examples, we will consider the case where the trick leader plays pattern combinations. Suppose player 1 plays AAK♠. Player 2 must respond by playing a pair of spades and a single spade. However, player 2 doesn't have any pairs, so he instead needs to play any 3 spades. His action space is {J73♠, K73♠, KJ7♠}. None of them beat player 1's move.

- **Combo Pattern 2: ♣ 2 Pairs + Single**

  Suppose player 2 plays AQQ44♣. Player 1 has no club pairs and has less than 5 clubs, so he must play all of them. In addition, he needs to choose any 2 cards to play. The action space is {862♣ AQ♦, 862♣ AJ♦, . . . , 862♣ XJ XJ}. Since player 1 is unable to perfectly match the pattern, he cannot beat player 2 with any move. But among his choices, some are better than others.

- **Combo Pattern 3: ♦ 2-Tractor + Single**

  In general, a pattern combination is only allowed if none of the other players can beat any of the patterns components. Suppose player 2 plays A6644♦. This is a combination of a 2-tractor and a single. However, player 1 could reject player 2's move in this case because he possesses a larger 2-tractor. So player 2 is punished for failing to play his move and must play the pattern that is beatable. In this case, he has to take back A♦ and play 6644♦. Same as before, player 2 chooses an action among {QQJJ♦, JJTT♦}. Player 1 is not required to reveal any card to reject player 2's move, but player 2's unplayed A♦ is now public information to all other players (this is a disadvantage for player 2). Therefore, human players usually only play a pattern combination if he knows that the pattern is unbeatable.

- **Combo Pattern 4: ♦ 3-Tractor + Single**

Suppose player 1 plays AQQJJTT♦. Player 2 has no perfect match, but he has 3 pairs of diamonds, so he must play them. In addition, he needs to play a single. The only choices are A♦ and 8♦. So player 2' legal action space is {AKK6644♦, KK86644♦}. Neither action beats player 1's move.

Next, we will introduce the concept of ruffing. When a player following a pattern doesn't have any cards in the pattern's suit, he can play any set of cards whose length is the same as the pattern. But for the player to beat the pattern, he must play a set of trump cards that perfectly match the pattern. Such moves are called *ruffs*.

To give a few examples of ruff moves, we will use the following initial hands:

Player	**Hand (non-trump)**	**Hand (trump)**
#1	AQQJJTT9♣; AAK♠	KJ33♥ 5♣ 55♥ XJ XJ
#2	AJT86644♦; AKK4♣	ATT722♥ 5♠ DJ

- **Ruff Example 1: ♠ Single**

  Suppose player 1 plays K♠. Player 2 doesn't have any spades (note the 5♠ is not considered a spade because 5 is the dominant rank), so all single cards are valid moves (i.e. his legal action space is {A♦, J♦, . . . , DJ}). Among those moves, every single trump card can ruff the K♠. For example, player 2 can play 2♥, which beats player 1 and gives player 2 the chance to lead the next trick. Deciding what cards to ruff a trick depends on the position of the player in the trick, and if the player is not the last to go, he needs to consider the possibility of the next player ruffing using something even bigger.

  When multiple players ruff a combination of singles, the player with the largest single card wins. For example, if the trick leader plays AK♠ and the dominant card is 5♥, then 2♥ DJ beats XJ XJ even though they both ruff the trick.

- **Ruff Example 2: ♠ Pair + Single**

  Suppose player 1 plays AAK♠. Player 2 doesn't have any spades (again, 5♠ doesn't count), so he can play any set of 3 cards. However, if player 2 wants to ruff player 1's move, he must play TT♥ and one more trump card, because ruffing requires playing a set of trumps that perfectly match the target pattern, in this case, a pair and a single. So among player 2's legal actions, the ones that can beat player 1's move are {ATT♥, TT7♥, TT2♥, TT♥ 5♠, TT♥ DJ}.

  In general, when multiple players ruff a trick containing any number of pairs and singles, the player with the single largest pair wins. For example, if the dominant suit is ♥ and the trick leader plays two pairs and a single like AAK99♣, then 88322♥ beats 7766♥ DJ even though they both ruff the trick.

- **Ruff Example 3: ♦ 2-Tractor + 2 Singles**

  Suppose player 2 plays AJ6644♦. Player 1 doesn't have any diamonds, so he can respond by playing any 6 cards. However, if player 1 wants to ruff the trick, he needs to play 6 trump cards, including a 2-tractor and 2 singles. Player 1's only trump 2-tractor is 55♥ XJ XJ. The valid ruff moves are thus {KJ♥ 55♥ XJ XJ, K3♥ 55♥ XJ XJ, K♥ 5♣ 55♥ XJ XJ, . . . , 3♥ 5♣ 55♥ XJ XJ}.

  When multiple players can ruff a pattern combination containing a tractor, they compare their trump tractors and the one with the largest tractor wins, regardless of the sizes of the other cards (the "kicker" cards) they play.

To build a game environment for Tractor, the algorithm must first be able to identify the patterns that are present in the trick leader's move, then check if each pattern component is unbeatable based on the other players' hands. If the check is successful, the leading move is executed and the environment then needs to generate the list of possible response actions the other players can make. The other players choose from those actions, and finally the environment needs to decide who wins in the trick. We implement the Tractor environment and include it in our code.