CS230

Training A Tractor Agent with Deep Reinforcement Learning

Junming Wang
Department of Computer Science
Stanford University
`junmingw@stanford.edu`

Abstract

Board games present a unique challenge for AI agents. Tractor, a popular playing cards game, is challenging because it requires players to make decisions based uncertainty due to not knowing other players' cards at hand. We model the problem as a Markov Decision Process with unknown transition functions and trained one AI agent playing this game reasonably well using deep reinforcement learning. Our results show that the trained agent performs very well against both random agents and agents using our designed strategy.

1 Introduction

Board games present a unique challenge for AI agents.*Tractor*, a popular playing cards game, is challenging because it requires players to make decisions with uncertainty due to not knowing other players' cards at hand. We build a Tractor simulator, model the problem as a Markov Decision Process with unknown transition functions, and train an agent that plays this game reasonably well using Deep Q Learning approach. Our motivation of using Deep Q Learning is due to the extremely large state space, so we use a neural network to approximate state values.

1.1 Rules of Tractor

Tractor is a four-player playing cards game. The players are divided into two teams: players sitting opposite to each other form a team. We denote the four players as players 0, 1, 2, and 3, and the teams as teams 0 and 1. In other words, Player 0 and Player 2 are on Team 1, and Player 1 and Player 3 are on Team 2. Each team has a *level* that starts at 2 before the game starts.

The objective is to win *rounds* to increase the level of one's team past 14. In each round, one team plays the role of *declarer* while the other team plays the role of *opponent*. The opponent team aims at winning cards, some of which have points, to accumulate points, and the declarer team will try to prevent the opponent team from accumulating points. The number of points, at the end of a round, will determine whether and by how many one team's level will increase, as follows:

- $points = 0$: the declarer team's level increases by 3.
- $5 \leq points \leq 15$: the declarer team's level increases by 2.
- $20 \leq points \leq 35$: the declarer team's level increases by 1.
- $40 \leq points \leq 55$: no level change.
- $60 \leq points \leq 75$: the opponent team's level increases by 1.

- $80 \leq points \leq 95$: the opponent team's level increases by 2.
- $points \geq 100$: the opponent team's level increases by 3.

Additionally, the opponent team will become the declarer in the next round if points is greater or equal to 40.

One deck of playing cards is used for this game. There are cards with points: "5" cards each has 5 points, and "10" and "King" cards each has 10 points, while the remaining cards have no points. During the game, there is also a *trump* suit which is determined in the dealing phase by one of the players putting down a card whose value equals to the current level of the declarer team. The suit of the card put down by this player will be the trump suit for the current round. A card in the trump suit is stronger than any card in any other suit. Both jokers are trump cards.

A procedural rundown of one round of Tractor is the following:

- Dealing phase. In a predetermined order (e.g. counterclockwise), players draw cards one by one from shuffled card pile until there are only 6 cards left in the middle (so each player will have 12 cards in her hands). In this phase, any player can put down a cards whose value equals to the level of the declarer team to decide the trump suit for this round.
- Declarer's reorganizing phase. One of the declarers take the last 6 cards in the middle, and then select 6 cards from her hands to put back in the middle, facing down. This card pile in the middle is called the *kitty*. Any points in the kitty will be doubly added to the current points if the opponent team wins them.
- Playing phase. The declarer who set the kitty plays first. One *trick* consists of 4 cards, one played from each player. The card the first player plays creates a constraint on the players playing after her: they can only play a card in the same suit. Only if they run out of same-suit cards can they play a card from different suit. After all 4 cards are played, this trick concludes, and the player who played the strongest card wins this trick, and points of cards played (if there are any) will be added to the points for this round if the winner of this trick is on the opponent team; none will be added otherwise.

Such rounds are repeated until one of the teams' level increases past 14. More detailed explanation of rules can be found here[1].

In order to reduce complexity of the model and we decide to simplify the game rules by skipping kitty assignment in the simulator.

2 Related Work

To the best of our knowledge, there is no any previous work that has tried to learn the optimal policy to play Tractor. However, there are attempts in training AI for other board games. For example, there is a game called *Werewolf*, the objective of which is to discern if players are lying. Gîrlea et al. (2014) solves this problem by tracking beliefs and intentions. Nakamura et al. (2016) uses a psychological model based on multiple perspectives to construct a Human-like agent to play the game.

Deep reinforcement learning has achieved expert-level play in Chess, Go, etc (Heinrich & Silver, 2016). AlphaGo (Silver et al., 2016) was able to defeat the world's best human players of Go. Additionally, OpenAI (Mnih et al., 2013) has trained agents to play Atari games and surpassed a human expert.

3 Approach

3.1 MDP Modeling

3.1.1 States

A state is what Training Agent observes at each point of time in the game. It contains the following information: current points, both team's levels, which team is the current declarer, Training Agent's

[1]https://www.pagat.com/kt5/100.html

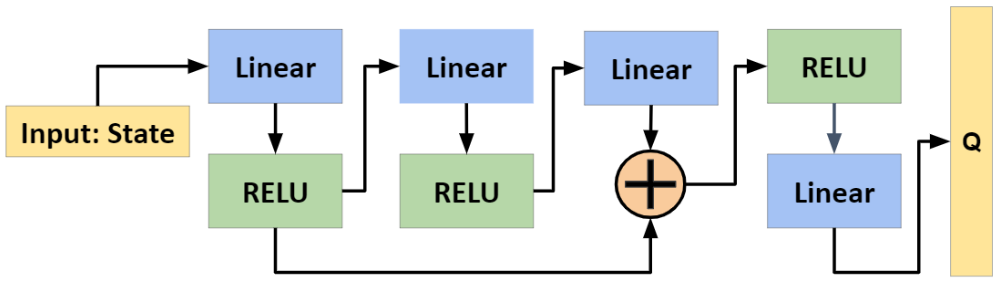

Figure 1: Our Network Architecture: fully-connected network with four hidden layers (followed by ReLU activations) and a skip connection (the '+' symbol denotes element wise addition).

cards at hand, the current cards played on board, and all past cards that have been played. Note that since we only need to update the Q values of the state in our algorithm where it is Training Agent's turn, the "whose turn it is" information is omitted from the state.

Formally, a state can be represented in a tuple with 6 elements. An example of state can be:

$$\begin{aligned} s = [[40], [2, 2], 0, [&\text{Spaid Jack}, \text{Club Queen}], \\ &[], [\text{Spaid Ace}, \text{Heart Ace}, \dots] \end{aligned}$$

This expression represents the state where the current points is 40, both teams are at level 2, currently Training Agent has "Spaid Jack" and "Club Queen" in hands, there is no cards played for the current trick (e.g. Training Agent is the first to play), and all previously played cards are included such as "Spaid Ace", "Heart Ace", etc.

3.1.2 Reward

The reward definition is: if the training agent is on the declarer team, the opponent team gaining points will cause a negative reward equal to the points gained; if the training agent is on the opponent team, their team gaining points will cause a positive reward equal to the points gained. In addition, winning the game gives reward.

3.2 Deep Reinforcement Learning

Now we describe in this section our main approach. We follow the DQN framework as in (Mnih et al., 2013), where a neural network is trained to estimate Q-values, assisted by a target network and a replay buffer.

3.2.1 Network Design

Our DQN pipeline is formulated as follows: let $\hat{q}(s, a)$ denote our Q-network parameterized by weights $\mathbf{w}$, which takes as input the state representation s and outputs the Q-values as a vector for each action, and $\hat{q}^-(s, a)$ denote the target network with target weights $\mathbf{w}^-$. Our goal is to minimize:

$$\mathcal{L} = \mathop{\mathbb{E}}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}^- (s', a') - \hat{q}(s, a) \right)^2 \right] \tag{1}$$

Input Our Q-network takes a vector representation of states as input. To encode the states into numerical vectors, we convert each sub-part (current cards, previous played cards, etc) separately and concatenate them all together. Specifically, all cards, we use one-hot representation for the suit (Heart, Club, Spaid, Diamond, Joker) and a scalar representation for the value; and for scalar values (such as current points), we directly convert them into 1d single scalars. Then, we concatenate all converted parts together, and as a result, we get an input vector $s \in \mathbb{R}^{4+6*(12+4+48)}$, or $\mathbb{R}^{388}$.

The converted input vector of the state from the previous subsection is:

$$
\begin{aligned}
s_{input} = [&40, 2, 2, 0,\\
&0, 1, 0, 0, 0, 11, 0, 0, 1, 0, 0, 12, \underbrace{0, \ldots, 0}_{\text{10 * 6 zeros}}\\
&\underbrace{0, \ldots, 0}_{\text{4 * 6 zeros}}\\
&0, 1, 0, 0, 0, 14, 1, 0, 0, 0, 0, 14, \underbrace{1, 0, 0, \ldots, 0, 1, 4}_{\text{all other cards' one-hot}}]
\end{aligned}
$$

In this example, the card Spaid Jack is represented as $[0, 1, 0, 0, 0, 11]$, where the first 5 indices represent the Spaid suit, while 11 represent the card value.

Output The output of our Q-network is a vector representing the estimated Q-values for each possible action. Note that for now, the action set consists of playing any single card, so $|A| = N$.

Network Architecture For our main training model architecture, we employ a fully-connected network with 4 linear hidden layers of size 1024 (followed by ReLU activations), with a residual connection (He et al., 2016) added from the activation of the first hidden layer to the logits of the third hidden layer. For the input layer, we project the input state vector into a 1024-dimensional vector, and for the output layer, we project the 1024-dimensional vector into an output vector of size $\mathbb{R}^{|A|}$. Detailed architecture is shown in Figure 1.

4 Simulator and Dataset

Our data are generated in the process of Deep Q Learning. We use Python to build the game engine and AI players to simulate the game environment. This simulator's purpose is to provide data for the neural networks by generating reward and next state.

4.1 Training Iteration

In each iteration, the simulator feeds the current state as input to the neural network, and obtains the best action according to the Q-value estimations by the neural network. Then the simulator takes the action and generates next state and reward. This (s, a, ns, r) (state, action, next state, reward) data pair is then stored to the replay memory $\mathcal{D}$. Subsequently, a minibatch of (s, a, ns, r) is sampled from the replay memory and is used to update the neural network.

4.2 Opponent Strategies

For all players who are not the training agent, we introduce two different strategies they use.

- **Random Strategy**: A player using random strategy determines card to play uniformly randomly from playable cards at hand.
- **Designed Strategy**: This strategy is a random strategy with a few rules added upon. A player under this strategy will play higher-valued cards if there are cards with points in the current trick, and will play lower-valued cards if there is no card with points in the current trick.

We tested the effectiveness of our designed strategy against random players: the team using our designed strategy achieves 75% win rate in 1000 games.

5 Experiment Results

We built simulator and learning algorithms as described in previous sections, and we conducted experiments with the same default Tractor environment (4 players, 1 deck, 2 teams) for 5 million iterations under two different settings: training the agent against random strategy players and our

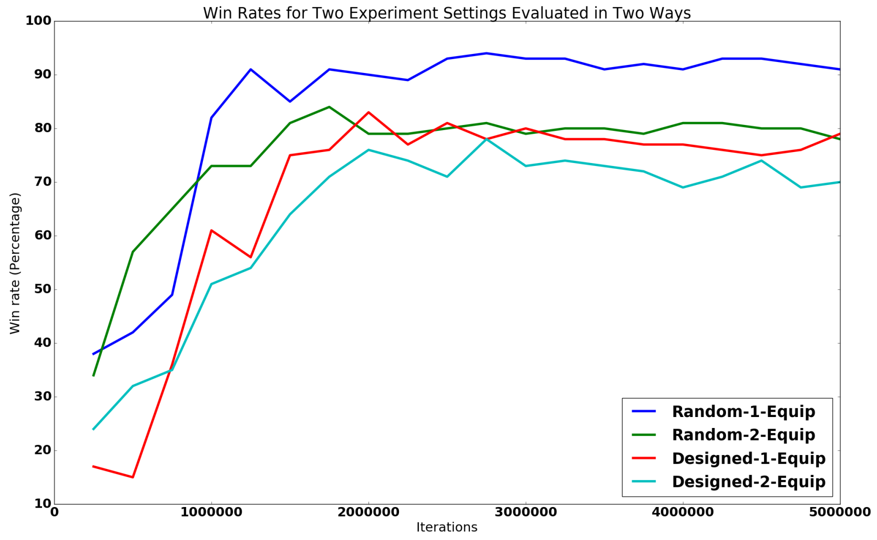

Figure 2: Experiment results under 2 different experiment settings (against random and designed), evaluated in two different ways (equipping only Player 0 and both Player 0 and Player 2 with learned policy). X-axis is the iteration number, and Y-axis is the winning rates (percentage).

designed strategy players. The weights of the neural networks are saved every 250,000 iterations and used to evaluate the performance by equipping either Player 0 or both Player 0 and Player 2 with learned policy and simulating 1000 games under both settings. The winning rates calculated are shown in Figure 2.

The result shows that all four models beat the baseline ($48\% - 52\%$ win rate), with the best-achieving 91% win rate against random agents. Our learning algorithm is capable of learning good policies against both random and strategic agents.

Two important observations can be made from Figure 2. First, learning a good strategy against random agents is easier than learning a good strategy against strategic agents. This result aligns with our expectation. Second, equipping both players on the same team with our learned policy in both experiment settings performs better than only equipping one player with our learned policy while leaving the other using the same strategy as the opponent. This observation shows that our learned policy makes it easier to win.

6 Conclusion and Future work

In this project, we took advantage of Deep Q-learning that combines stochastic mini-batch updates with experience replay memory to ease the training of deep networks to solve a popular game with high dimension observation space. Future works include exploring two-decker version of Tractor, in which players are not limited to play 1 card at a time, which will tremendously increase the action space. In addition, building a interactive interfaces and having humans play against trained AI can be an interesting future direction too.

7 Contributions

Junming contributed to most work related to this project. Credits are given to CS234 Teaching Team for part of the deep Q network code comes from Homework 2 in CS234. Github directory for source code of this project: https://github.com/tomjmwang/CS230-Final-Project.

References

Gîrlea, C. L., Amir, E., and Girju, R. Tracking beliefs and intentions in the werewolf game. In *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Heinrich, J. and Silver, D. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Nakamura, N., Inaba, M., Takahashi, K., Toriumi, F., Osawa, H., Katagami, D., and Shinoda, K. Constructing a human-like agent for the werewolf game using a psychological model based multiple perspectives. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–8. IEEE, 2016.

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