Final Report for ZSY Playing

Group Members: Wei-ting Hsu, Leon Lin, Hua Feng
SUNet ID: hsuwt(006117538), leonl(006053207), fengh15(06043870)

Introduction

We aimed to train an agent through Reinforcement Learning to beat a human at the card game ZSY. The inputs are the game states at each particular turn and the outputs are the optimal move given the input game state. Below are the abbreviated game rules:

1. A game consists of a number of rounds and each round has a number of turns. Both players are dealt 18 random cards (their 'hand's) at the start. On the first round, one player is randomly selected to have the first turn. That player plays a pattern which can be one of the following: 'single' (a single card, e.g. 5), 'double' (two cards of the same rank, e.g. 99), 'triple' (three cards of the same rank, e.g. 777), 'bomb' (four cards of the same rank, e.g. 4444), or a 'chain' (consecutively ranked cards in which each 'link' is at least a double, e.g. 6677788 but not 66788 because there are fewer than 2 7s or 6688 because the ranks are not consecutive).

2. At each following turn, a player can play cards that match the pattern and have higher rank (e.g. 555 can be followed by 777 but not 88 or 444; 55666 can be followed by 77888 but not 77788 or 33444), play a bomb, or pass. Players alternate turns until one player passes, and the round terminates. The last player to play some cards in the round begins the next round and sets a new pattern. The game terminates when one player runs out of cards and wins the game.

To model this game, we built a game class in python that deals cards, keeps track of the running card counts, and sends game state data to the appropriate agent class that return a move according to that agent's policy. This was a joint project with CS 221, but in that class we approached a different version of the game in which both players could see the other's cards. In that case, the game is theoretically deterministic from the perspective of the agents, and we tried to design a minimax agent rather than a learning agent. The only shared portion was the game class we wrote, modified to send full game state data (i.e. hands of both the agent and its opponent) to the agents for 221 rather than partial game state data (i.e. the hand of the agent and the number cards the opponent has left) for 229.

We tried two RL algorithms: Q-learning and TD-learning. In the first, we tried to find the values for states coupled with actions (the move taken), and in the second we tried to find the values for just the states.

Related work

The classic technique for turn-based game is a game tree search based on different assumptions of opponent's strategy[1]. Perfect information of the game state is required to construct the game tree to predict opponent's strategy and to evaluate the value of successor states. But in ZSY, the players don't know their opponent's hand since the cards are randomly dealt to both players and the remaining cards are unknown. This makes the information incomplete and a game tree impossible.

To deal with the incomplete information problem, various strategies have been proposed. The most straightforward method is to enumerate all the possibilities and take the action maximizes agent's game value[2]. But, this only applies to games with small possible state space due to the cost of computation. One possible way to address this problem is by making the game open-handed [3], which reduces the number of possible states at the cost of changing the game policy; this would also affect the behavior of the opponent. Other researchers propose only investigating a statistically significant number of possible states based on player's prior knowledge [4], but this could lead to suboptimal strategy due to strategy fusion and non-locality [2].

TD Gammon [5] for backgammon and Deep Blue for chess were the inspirations for this project. ZSY is a fundamentally different challenge than both games, however, since the exact game state is unknown to the players. In chess, the state is known and the transitions are deterministic; in backgammon, the states are known and the transitions are probabilistic with known probabilities; in ZSY, the states are partially known, and the transitions from one partially known state to the next are probabilistic with unknown probabilities.

Dataset

All the data was generated by the game class we wrote, which can be found here. For our two approaches, we extracted different features.

Algorithms and Features

Q-Learning

With partially known states and unknown transitions, it is still possible to use Q-learning with what is known about the state paired with the possible actions. As the game is a finite horizon problem, we set □ = ***1***. Making this the Q-function:

$$Q^*(s,a) = R(s,a) + \underset{s' \sim P_{sa}}{E}[\max_{a'} Q^*(s',a')]$$

Ideally, s (state) is both player's hands and the current pattern at play, a (action) is a move by the player, and R(s,a) (reward) is 0 for all states in the middle of the game, +10000 if the agent wins, -10000 if the agent losses. Q*(s,a) is the optimal Q-value for being in state s and take action a. s' is the state taking a at state s leads to, which is under some transition probability distribution .

If both player's hands are known to an agent, then s' is deterministic, but it isn't so we must use a partial state for s instead, as described in the introduction. As only a partial state is known, and the number of different states enormous, the best option was to extract what features we could from the state and use function approximation:

$$Q(s,a) = \theta^T \phi(s,a)$$

where theta is our learned weights, and the optimal policy is:

$$\pi^*(s) = \max_a Q(s,a)$$

Weights are learnt by gradient descent over the objective function:

$$J(\theta) = \frac{1}{2} \sum_{((s,a),(s',a'))} (\theta^T \phi(s,a) - R(s,a) - Q(s',a'))^2$$

The weights are updated with each new training pair that was produced:

$$\theta := \theta - \eta[\theta^T \phi(s,a) - R(s,a) - Q(s',a')]\phi(s,a)$$

The problem of partially known states meant that we could not evaluate the Q for the opponent's turn, as the agent cannot know what the opponent's possible moves are. So, we had to only evaluate Q at the agent's turn. We didn't use any feature involves opponent's hand other than the number of cards in the hand.

At the beginning of the agent's turn, we extracted many features, including: the difference between agent's and opponent's card counts, the number of doubles, triples, and bombs in agent's hand, whether it is before(marked as 1) or after half game (marked as 0) combined with number of cards with high/low value, etc. Each feature is coupled with the action to take. (e.g at a certain round of full deck game, if agent has 366JK in ts hand, opponent has four cards, the move under evaluation is playing a 3, then the corresponding features would be ('f1',1,(3,[1])):1 (card count difference), ('f2',[1,0,0],(3,[1])):1 (for 1 double, no triples, no bombs), ('f3',0,2,(3,[1])):1 (later half of the game, 2 high value cards(JK)) and ('f4',0,3,(3,[1])):1 (later half of the game, 3 low value cards), where 3,[1] indicates the action of playing a 3.

The Q-learning agent was not very successful, and we tried to address its flaws with the next algorithm.

TD-Learning

We believe that the main problem with Q-learning was that the feature space was too high dimensional, so our next attempt was TD-learning: value iteration without considering the action, just the state. That makes the optimal value function:

$$V^*(s) = R(s) + \max_a \underset{s' \sim P_{sa}}{E}[V^*(s')]$$

And, with features for value approximation, we have:

$$V(s) = \theta^T \phi(s)$$

However, with partially known states, we can't know the transition probabilities. This is a problem for TD that wasn't a problem for Q because without the action, evaluating the optimal policy depended on evaluating the next states and not just the current one:

$$\pi^*(s) = \max_a \underset{s' \sim P_{sa}}{E} V(s')$$

To solve this, we invented a hybrid Q/TD model that allowed us to make the transitions deterministic. The transition from the agent's turn to the opponent's turn is deterministic because the agent knows what card it plays, but the transition from the opponent's turn back to the agent's is unknown because the agent doesn't know what actions the opponent has. However, the agent also doesn't decide the opponent's action, so it doesn't need to evaluate optimal values for the second transition. So, instead of the states being only when it's the agent's turn, we have the states as being both when it's the agent's turn and when it's the opponent's turn, and we can reduce the optimal policy function to:

$$\pi^*(s) = \max_a V(s')$$

where s is the game state at the agent's turn and s' is the game state at the opponent's turn. But, this raises another problem: the game state at the agent's turn and at the opponent's are fundamentally different in a way that affects every feature, but that no feature captures. So, we convoluted all the features over whether they were extracted during the agent's turn or the opponent's turn. This is, if the agent has 3 7's at the beginning of its turn, it would have a feature that is (1, 'triple', 7), but if it has 3 7's at the beginning of the opponent's turn, it would have a feature of (2, 'triple', 7) with the '1' or '2' being the convolution. Ultimately, this hybrid model reduced the feature dimension from num_features*num_actions in Q to num_features*2 in the hybrid.

Here are the features we extracted, pre-convolution:

- Number of cards in the agent's hand (e.g. ('Alen'):7 if the agent has 7 cards)
- Number of cards in the opponent's hand
- For each single, double, triple, or bomb, a binary feature of the type paired with the rank (e.g. ('triple', 7):1 if the agent has three sevens)
- The count of the chains with a particular length (e.g. ('FullP', 6):1 if the agent has 445566, ('FullP', 5):2 if the agent has 77788 and JJQQQ)
- The count of the sub-chains of a chain with a particular length (e.g. ('SubP', 4):2 if the player has 445566 because there are sub-chains of 4455 and 5566)

Some of these features have vastly different ranges than others, so we rescaled them to make them be between zero and one most of the time. The values for the features for the number of cards were divided by 18, the maximum number of cards an agent can have, and the values for sub-chains were divided by 3, which we determined through experimentation to produce the best results. The values for full chains were left unchanged because even though they can be greater than one, specific pattern lengths are rare enough that this did not damage the results; the remaining features are already binary.

We chose to only extract the length of a chain and not the specific rank and pattern of it because, while most random hands contain some chains, there are so many possible chains that the probability of having any particular one is vanishingly small and may not be visited enough times in exploration for their features to have optimized weights.

An important strategy of the game is to try to play full length chains instead of their sub-chains. Playing a full length chain both gets rid of more cards and is less likely to be countered. But, if the agent holds on to a full chain too long, it might not even get the chance to play it if the opponent already won. This was the intuition behind having features for sub-chains as well as full-chains: we wanted the agent to be able to tell when it is better to break up a chain and when it is better to pass and keep it.

To address other problems with Q, we made some other changes in TD. In Q, we updated the weights after each transition; but, since rewards only come at the end of the game, much of the initial learning had kept 0 weight for most features; in TD, we store the extracted features in a history for each agent, and only update the weights after a game terminates, walking backwards from the last features. Since the number of different games is enormous, we may have reduced the step size too rapidly, so we changed from a step size of $1/\sqrt{}$ to $5/\sqrt{\ + \ \mathit{100}}$ to make the gradient drop more gradual. Also, we changed the reward to +1 for winning and -1 for losing.

Additional Methods

Greedy: We implemented a baseline, 'greedy', agent that always takes the legal move with the lowest card rank without considering other factors. We used this as our preliminary test for the learning agents because testing against humans takes too much time, and, though this baseline has never beaten a human, it does win 79.5% of the time against an agent that just plays randomly.

TDNS: We noticed that the different instances of TD algorithms trained with the same, small, number of games can have wildly different win rates against the greedy agent. This may be due to local optima. To counter this, we created TDNS (short for 'TD Natural Selection'); we initialized 50 TD agents, trained each for some number of games, tested them against the greedy agent, and killed the bottom 40; then we trained the remaining 10, tested them again, and removed some more; this process was repeated for 3 rounds, resulting in one algorithm that performed 2% better than the non-selected TD agent against the greedy agent.

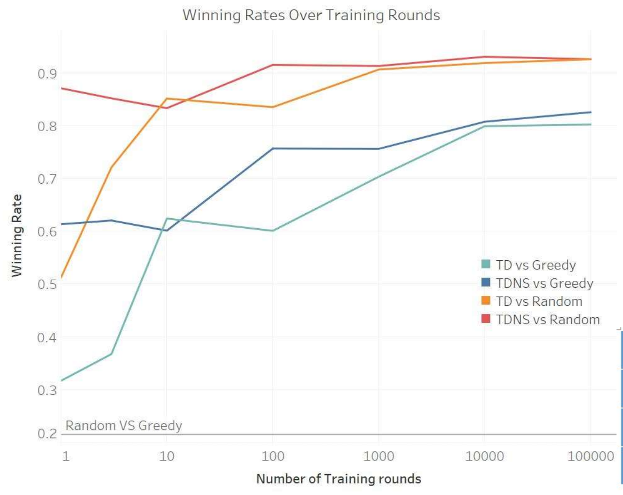
More complex features: The features we used in our algorithm don't capture some more subtle strategies in the game. For example, if an opponent has already passed on beating a double 7, then that opponent is probably less likely to be able to beat a double 9. Also, it might be possible to infer from what cards have already been played what cards might remain in the opponent's hand as there are finite cards in the deck drawn without replacement. We attempted to add features for both of these, one that kept track of which patterns the opponent has passed on and one that approximated the probability that the opponent has the cards to beat a pattern, but in testing neither additional feature improved performance over the greedy agent, so we removed both in the final algorithm.

Experiments, Results, and discussion

The Q-learning agent, our first attempt, did not perform very well. It won no games against the greedy agent and no games against humans. Only when we reduced the deck from 54 to 36 cards did it beat the baseline. This was probably because of the feature dimension, but could have also been due to other problems mentioned above. Once the TD agent began to win against greedy, we doubled down on improving the TD agent. We graphed the learning curve for the TD and TDNS agents below, and we included a table for the wins and losses of the best algorithm against humans.

Primarily, we looked at the TD agent's win rate against the random and greedy agents and against humans. After an agent was trained each of the following: 1, 3, 10, 100, 1000, 10000, and 100000 games, it was tested against both the random and greedy agents for 100000 games to produce an accurate win rate. For the TDNS agent, 5% of the training games were on the first round, 25% were on the second round, and 70% were on the third, except for the 1, 3, and 10 game training data points: for 1 game of training, there was only one round in which all 50 agents were trained for 1 game, and the bottom 49 were killed; for 3 games of training, each round had only one game; and, for the 10 games of training, round 1 had 1 training game, round 2 had 2, and round 3 had 7. For human trials, we each played 20 games against the TDNS agent that was trained over 100000 games.

Secondarily, we looked at the moves it took too see if they make sense strategically from our perspectives, and we looked at if the weights for the particular features made sense. Though these metrics are subjective, they were helpful early on in determining if the agents were learning in a sensible way.



player\score	TDNS vs Human Player
Player1	8:12
Player2	8:12
Player3	6:14

While the TDNS agent does not have a >50% win rate against humans, we have observed it make good moves that neither the greedy agent nor the Q agent had made, such as holding onto a long pattern for later use instead of breaking it to play immediately. It is difficult to say if all the weights are sensible—if we knew that ahead of time, we wouldn't need to do reinforcement learning—because the weight of a particular pattern doesn't necessarily correspond directly to how good it is to play it, as playing a pattern changes other features such as the number of cards the agent's hand and other patterns around that pattern; however, the weights for single, double, and triple cards do increase monotonically from the lowest rank to the highest rank, which seems right.

One problem we noticed is that the TDNS agent (and the TD agent) would often begin a game with a high-rank card or pattern. This is contrary to the usual game strategy of starting with low-ranked cards to get rid of them first, as they are harder to get rid of in the late stage of the game. This does make sense from the perspective of the features, as none of them capture temporal elements of the game, i.e. that it treats every turn and every round the same. We could not come up with any solutions to this, except perhaps to convolute the number of cards in a player's hand over every other feature, but that would again dramatically increase the dimension of the feature space.

Our original goal was to have an agent that could beat a human most of the time, but with unknown states, perhaps a computer agent for this game is fundamentally limited to a 50% win rate against an expert human player, as only so much information about the game can be extracted.

Conclusion/Future Work

Ultimately, we did fail to make an agent that can beat humans most of the time, but we have constructed a learning agent that beats any other agent we designed with manually specified policies and that could, at least, beat humans a significant percentage of the time.

We have many additional ideas that we have yet to experiment on, such as other additional, more complex features, but our best ideas for additional features did not improve the algorithm's win rate. Perhaps this is because those complex features should interact with the other features in nonlinear ways that would have been better captured by a convolution neural network. If we were to continue this project, feeding the features through a CNN would be the next avenue we would explore.

One of our ideas was to train the algorithm against the greedy agent instead of against itself; while this did lead to it being able to beat the greedy algorithm with much fewer training games, after 100,000 games the win rate still leveled off at about 80%, the same as when it trained against itself; and, it probably would have overfit to a strategy to just beat the greedy agent that would make it less effective against humans. There was the possibility that TDNS also overfit to beating the greedy agent, but we figured that with only three rounds it would not have been too severe. Ideally, TDNS would have operated by testing against humans, but just as it would be best to train against humans, this would have been far too time intensive. The next step, after trying a CNN, would be to build an online interface where anyone could play against this program so that we could collect data from a significant number of human games.

References
[1] Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., & Edwards, D. D. (2003). *Artificial intelligence: a modern approach* (Vol. 2). Upper Saddle River: Prentice hall.
[2] Frank, I. (1998). Search in Games with Incomplete Information. In *Search and Planning Under Incomplete Information* (pp. 89-123). Springer London.
[3] Berlekamp, E. R. (1963). Program for double-dummy Bridge problems—a new strategy for mechanical game playing. *Journal of the ACM (JACM)*, *10*(3), 357-364.
[4] Levy, D. N. (1989). The million pound bridge program. *Heuristic Programming in Artificial Intelligence, The First Computer Olympiad, pages 95 -103. Ellis Horwood*.
[5] Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, *38*(3), 58-68.