ZSY

CS 230 Project Milestone

Leon Lin (leonl)

Github repo: https://github.com/leonl0000/ZSY

Introduction

争上游 (ZhengShangYou, or"Competition Upstream") is a Chinese card game that is part strategy, part luck. Each player is dealt about 18 cards that they must get rid of to win, and they get rid of cards by matching patterns. Game rules are in the appendix.

Like chess and go, there are patterns to be seen that, if accurately modeled, convey a large advantage in game play. The state space is enormous—possibly on the order of the factorial of the number of cards in play. As far as I could tell, this particular game has not been studied before for automation. Unlike chess and go, the initial states are random and a very small portion of them can even be unwinnable.

A year and a quarter ago, I was in a 3-person CS 229 team that tackled ZSY. We used a TD-learning algorithm with hand-picked features that played many games against a random and a greedy agent we designed. While it did beat those purely manually designed agents, it could only beat humans about 30% of the time. There was no neural network involved and the hand-picked features didn't work that well.

In this project, I aim to apply neural networks to a q-learning agent to ultimately be better at ZSY than humans.

Dataset

The first challenge is to represent the game in a way that captures its complexity without being unworkably memory intensive. During gameplay, it is useful to represent hands as counts of cards of each value because they can be simply added to or subtracted from to represent taking a move. However, this obscures the fact that having two of a kind is fundamentally not just twice having a single: having a pair allows for different kinds of patterns to be formed.

Thus, I've made a dual-representation. During gameplay, the hands, the moves, and the history of moves are all represented by counts. During learning, they are represented by a stack of one-hot encodings of how many there are of each card.

A hand of cards

```
>>> h
array([[2, 1, 2, 0, 0, 0, 2, 2, 2, 1, 2, 1, 1, 0]])
```

(1,15) array representing the counts of each value of card

```
>>> zsy.dc.handToExpanded(h)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]], dtype=int6
```

(5,15) array with each column being a one-hot encoding of whether there are 0, 1, 2, 3, or 4 of a card.

Each player starts with 18 cards and keeps playing until one of the players runs out. The representations above work for both a hand and a move. Thus, I can represent the entire game as a sequence of (5, 15) move-arrays and reconstruct every aspect of the game from there.

For Q-learning, I let (s, a) be represented by the current history, the cards in a player's hand, and the move that player makes. At the moment, I haven't figured out how to deal with the fact that the history is of variable size, growing as the game continues. So, I've simplified it by summing over the moves that a player has taken and the moves the opponent has taken. Thus, the final representation of (s, a) is 4 of the (5, 15) arrays: the cards played by the agent, the cards played by the opponent, the cards in a agent's hand, and the move the agent takes.

The reward is 1 or 0 at the end if the player won or lost. Defining the intermediate values is tougher: Q* should be the expectation of taking the best move, but at the beginning neither how to calculate the expectation nor what the best move is, is known—to know, after all, would be to have already solved the problem.

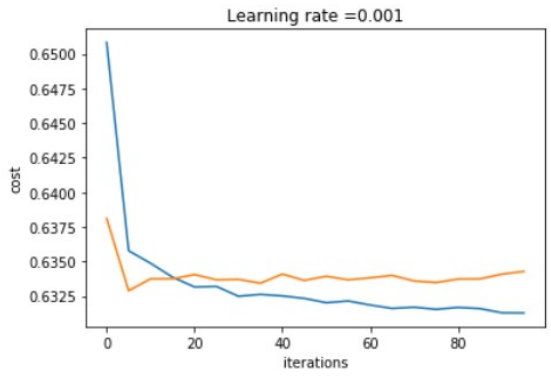
For now, I've simply used the end value (win/lose) as the value for every step in the game with a discount factor of 0.9 per move[1]. I expect that when the model is very good, the discount factor can be changed to 1 as this is a finite horizon problem.

Ultimately, this means that I have x, y pairs where x is the 4 (5, 15) arrays representing (s, a) and y is the approximation of Q*(s, a) by discounted reward.

I simulated my first dataset by letting 2 random agents play 100,000 games against each other. Each game lasted an average of 19.2 back-and-forth moves for a total of 1.92M data points.

Approach

I flattened and concatenated the 4 (5, 15) arrays for x into a (300, 1) array. I then fed this into a 3-layer fully connected network with 200, 40, and 1 units in the layers. The first two used a ReLu activation, the last was sigmoid for the predicted Q*(s, a).




```
Parameters have been trained!
Train cost (No dropout): 0.62401
Test cost (No dropout): 0.634024
```

The data was split into 96/2/2 for train, dev, and test. After just 5 epochs the dev loss started to steadily climb so I added dropout with keep_prob=0.5 to each layer. This graph to the left is the logistic loss over 100 epochs.

The training loss (blue) slowly but steadily decreased over the 100 epochs but the dev loss just flattened out after about 20 epochs.

It is difficult to tell how much over-fitting there is because Bayes' error can't be known without the real Q*(s, a) values.

[1] More specifically, I have 1 and -1 as the rewards for winning or losing. This way, when the discount is applied, the values are shrunk towards 0 as the midpoint between winning and losing. If the rewards were 1 and 0, then applying the discount would bias all rewards towards losing (closer to 0). After applying the discount, I then add 1 and divide by 2 to get the value between 0 and 1 for the sigmoid function to predict.

I then built an agent that played based on these trained parameters: at each turn, it sums over the histories and concatenates onto it every possible move it can make and the hand that results from it to create (s, a) pairs. It feeds these into model and chooses its move based on the best result.

Table: Learning Agent win rates

vs.	Random	Greedy	Human
Deep Q	96.6%	71.86%	--
TD (linear)	92%	81%	~30%

Played against the random agent, the deep Q agent won 96.6% of 10k games. Played against the greedy agent, the deep-Q agent won 71.86% of 10k games. The non-deep learning TD agent from the previous project had a 92% win rate against random and 81% win rate against greedy. The TD agent had a 30% win rate against humans, but I haven't run any human trials yet for the q agent, nor have I (yet) re-written the TD agent to work with the new game simulator.

Analysis/Next Steps

Immediately, it looks like the deep Q agent learned very well how to beat the random agent, scoring better than the TD agent in that regard. This, I suppose, demonstrates how neural networks fit data much better than a linear model. It didn't do as well against greedy, but given that this was just the first model I'm quite surprised it beat the greedy algorithm at all. Writing the simulator took longer than I expected, but now I should be able to focus on the model itself.

The next step is to iterate the model: let the agent play against itself to generate new data and train on that data. There are many hyper-parameters to tune before this, though. Besides the network architecture hyper-parameters like learning rate, keep_prob, and layer sizes, there is also the discount for Q(s, a) and the exploration_prob[2] for the q agent.

If the network architecture remains the same between model iterations, I can compare the losses between them to get a sense of the improvement and use the previous model's parameters as a sort-of pre-training for it. As noted before, it's hard to get a sense of what the minimum loss is because the discount factor is only a proxy for the fact that the value of Q*(s, a) is based on expectations that are difficult to model. It may very well be that for the data of two random agents playing against each other, a 0.634 dev loss is close to the best any model could get.

Tangential to that, I need to re-write the TD agent for this simulator to play it against the deep Q agent and I need to test the deep Q agent against humans.

Additionally, the primary motivation of the (5, 15) representation of a hand was to use a CNN over it to find patterns. With the time remaining, implementing a CNN will be second priority to optimizing the current, fully connected model. I also think that an RNN might be useful for dealing with the sequence of moves in a game because summing over them loses some of the complexity. However, the RNN module is next week so I don't yet have a good sense of how it could be used.

[2] With chance of exploration_prob, the deep Q agent will sample from its possible moves weighted by their scores rather than just take the move with the highest score.

Appendix

Rules for simplified ZSY:

Two players are dealt 18 cards randomly from a deck of 54 cards (13 per value, 2 jokers). The goal is to get rid of all the cards in ones hand. A coin is flipped to determine who starts the first round.

That player that starts a round has these options to play:

- **Single**: 1 card
- **Double**: 2 cards of the same number
- **Triple**: 3 cards of the same number
- **Bomb**: 4 cards of the same number
- **Chain**: a series of consecutively-valued cards, for which each 'link' has at least two of that number. For example, 33444, JJQQKK, are valid patterns. 5556677778899 is, but 44566 is not because there's only one five and 7799 is not because it's not consecutive.

The next player must play cards that match the pattern exactly, but are higher. For example, if 777 was played, the next player could follow with 888, 999, QQQ, or so on. If 55666 was played, he could follow with 77888 or JJQQQ (but not JJJQQ). Alternatively, the player can play a "Bomb" over any pattern, and those can only be beaten by higher bombs. Or, the player could pass.

The order of card values is shifted slightly from typical, with 2 being the highest non-joker card. The order for ZSY from low to high is (suits don't matter):

3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A, 2, Black Joker, Red Joker

When every player has passed, the last player to play some cards wins the round, and gets to start the next round, setting the new pattern. As soon as a player runs out of cards, that player wins the game.