Battle Royal Exploration for Card Game ZSY

Leon Lin (leonl@stanford.edu)

The Game

争上游 (ZhengShangYou, or “Competition Upstream”) is a Chinese card game that is part strategy, part luck. Each player is dealt ~18 cards, get rid of cards by matching patterns, and win by getting rid of all their cards first. I aimed to train an RL agent to have an above 50% win rate against humans in a 2-player version of this game.

3 Main Challenges

- *Stochastic + large discrete state space*: With 2 players each dealt 18 cards, there are ~151 trillion initial states
- *Partially Observed*: A player only see their own cards.
- *Train vs Test*: It cannot be trained against humans for the data required, but it will be tested against them

I have previously attempted this but only achieved about a 40% win rate against a human player (me).

The Data

Balancing the need to preserve the complexity of the game with the need to save memory, all ‘hands’ of cards are represented by stacks of one-hot encodings of how many there are of each card (suit doesn’t matter in ZSY). 5x15 arrays represent the player’s hand, action, cards it has played, and cards the opponent has played. These 4 are concatenated into a 5x60 array to represent state-action pairs.

A hand of cards

```
>>> h
array([[2, 1, 2, 0, 0, 0, 2, 2, 2, 1, 2, 2, 1, 1, 0]], dtype=int8)
```

1x15 array representing the counts of each value of card

```
>>> zsy.dc.handToExpanded(h)
array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0],
       [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

5x15 array with each column being a one-hot encoding of whether there are 0, 1, 2, 3, or 4 of a card

Simulated games are stored in a replay buffer inspired by Mnih et al. (2013). Many different models are initialized together, battle each other to contribute to a single, collective replay buffer for data efficiency, exploration, and hyperparameter search.

Github Repo

PC build of game

Mac build of game

Round Robin to Battle Royale

Battle Royale Algorithm

Init buffer with 100,000 random games
Init A with 24 agents, 12 with duplicate models
for e in NumEpochs **do**
$\epsilon = \frac{1}{e+2} * \sqrt{\frac{|A_0|}{|A|}}$
Kill_Threshold $= 0.5 * (1 - .75^{\,e * \frac{|A|}{|A_0|}})$

TRAIN
for i in range(500) **do**
sample = random 4096 points from buffer
for Agent ag in A **do**
Train Agent ag sample

SIMULATE
for i in range($|A|$) **do**
for j in range(i, $|A|$) **do**
Simulate $50000/\frac{|A|*|A|+1}{2}$ games between A_i and A_j
Add games to buffer

TEST
for Agent ag in A **do**
Update ag’s ”VS” winrate
Test Agent ag for 200 games against the 3 static agents
Update Agent ag’s winrate against the 3 static agents

KILL
if ag’s VS winrate < Kill_Threshold **then**
Remove ag from A

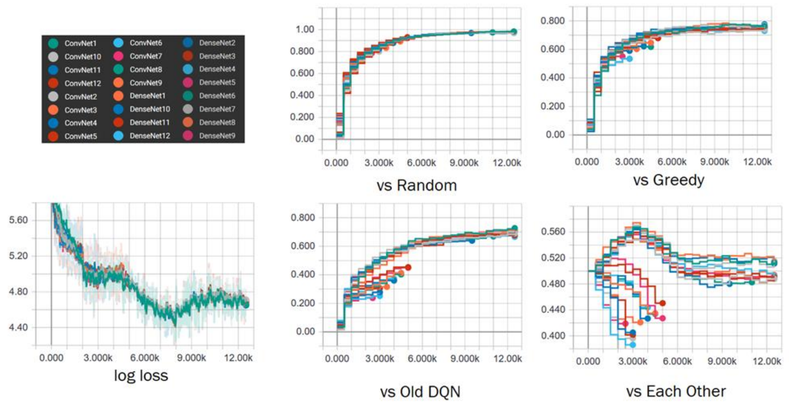
The PvP nature of the game allows for different models to be tested against each other, and gradually selected/reduced as training continues.

Two initial experiments were run to determine the parameters for the final algorithm:

1) To determine the size of the buffer, 5 agents were trained on a large amount of random games, and tested at various intervals to see when winrates stopped increasing. This set the buffer size to max out at 250k and start with 100k

2) To figure out when to kill models and how much to explore, a shorter version of the algorithm with 12 agents and no removal. From this, I decided the exploration and kill threshold formulae.

Then, the models from the second experiment were doubled (each one with 2 instances), and the Battle Royale was run.

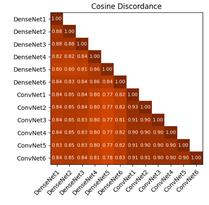


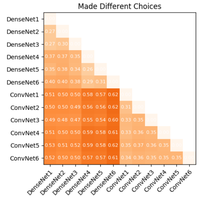
Each state-action pair is assigned a value of 1 for winning games and 0 for losing games. DenseNet Agents flatten the 5x60 array and pass it through 3-4 fully connected layers while ConvNet Agents first pass it through 2-4 layers of 3x3, 1x3, or 1x1 valid convolutions before 2 fully connected layers. Activations are all ReLu, LeakyReLu, or Sigmoid (for the output layer). See the configs.csv files on the Github repo for full definitions of each agent. Testing was done against a Random agent, a Greedy agent, and the agent from the previous project.

ConvNet agents performed significantly better than DenseNet agents.By the end 10 of the 24 initial agents remained, all of which were CNNs.

Discordance and Aggregation

The ConvNet Agents all had similar performances despite different network structures. Did they learn the same policy? Using the models from the Round Robin experiment, one final round robin was run: all 12 models evaluated each state and action to find the average cosine similarities for their Q values across and percentage of times they chose different actions





Despite being trained on the same data, their policies seemed quite different. This suggested that an aggregation model might improve performance. 9 kinds were tested: taking the average, minimum, or maximum Q value from the top 3, 6, or 9 models. The aggregation agents did significantly better than any individual agent.

Agent vs	Random	Greedy	Old DQN
Min6Combo	97.8%	77.7%	75.4%
ConvNet2	96.6%	72.6%	62.8%
DenseNet9	96.2%	61.7%	24.4%

Human Tests

I build the game in Unity for PC and Mac and got these test results back from human players.

The data is too small to conclusively say that human level performance was achieved, but the initial results are promising: on average, no human beat the agent.

Player	Ratio	Winrate	σ
1	62:66	48.4%	4.4%
2	27:32	45.8%	6.5%
3	16:24	40.0%	7.5%
4	12:20	37.5%	8.6%
5	20:36	35.7%	6.4%

Next Steps

With the game built for human players, it is possible to collect real human player data and to better visualize different game states where the agent chose badly. Additionally, different models such as RNNs can be attempted and gradient normalization may be helpful for longer term training.