Deep ZSY
Leon Lin (leonl@stanford.edu)

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 aimed to apply neural networks to a q-learning agent to ultimately be better at ZSY than humans. The inputs were simplified encodings for (s, a) pairs representing the game states and moves from simulated games and the output was an estimated Q*(s, a). I used three and four layer dense neural networks with ReLu and sigmoid activations and dropout regularization.

Related Work

The linear algorithm we built approached data purely sequentially: updating the model weights after each game. Inspired by Mnih et al.’s approach, I instead simulated between 10,000 and 100,000 games between each training session with experience replay.

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]

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

(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. To simplify the
representation of the history, I've summed over the
moves the opponent has made and the move the
agent has made in order to make the input
sequences all of the same length. 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.

Applying a discount factor in this finite horizon
problem doesn’t strictly speaking make sense from a
theoretical perspective, however during
experimentation it proved to be crucial for the
model’s performance. An alternative method I
thought of was to instead discount the costs so
that the model was less penalized for a poor
estimation of Q* when the actual value for Q* was
less well known. In practice, however, this
generally led to poorer performances and the idea
was scrapped after a few iterations.

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. The first round consisted of
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.

The First model

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. 50% dropout was applied
to each hidden layer. The first two used a ReLu
activation, the last was sigmoid for the predicted
Q*(s, a). The loss was the standard logistic loss
function.

The data was split into 98/2 for train and dev.
The graph shows the train and dev loss over 100
epochs in blue and orange, respectively. The
1.92M data points were trained with a mini-batch
size of 1024. The dev loss throughout was
calculated without dropout whereas the train loss
was only calculated without dropout at the very
end of training.

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.

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.

The ultimate metric is how well it would
perform against humans, but collecting large
amounts of human data was not possible. To test
the agent, I let it play 10,000 games against a
random agent (that took legal moves uniformly at
random) and a greedy agent (that took whatever
move it could to get rid of its lowest value cards).
I then created a combined loss metric that was the
product of the percentage losses to each static
agent.

Afterwards, the Deep Q agent played 100,000
games against itself, where with an exploration
probability of 0.1 it would draw a move weighted by
the estimate Q* value. After this agent was trained
for 100 epochs, it was tested against a few
humans, and the score below was how it fared
against the best human player over 200 games.

The table atop the next page shows the results
of the linear TD algorithm and the two iterations of
Deep Q against the static agents and against
humans.
<table>
<thead>
<tr>
<th></th>
<th>Vs Greedy</th>
<th>Vs Random</th>
<th>Combined Loss</th>
<th>Vs Humans</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD (linear)</td>
<td>81%</td>
<td>92%</td>
<td>152</td>
<td>30%</td>
</tr>
<tr>
<td>Deep Q</td>
<td>71.9%</td>
<td>96.6%</td>
<td>95.7</td>
<td>-</td>
</tr>
<tr>
<td>Deep Q, 2nd iteration</td>
<td>75.7%</td>
<td>97.2%</td>
<td>67.3</td>
<td>43.5%</td>
</tr>
</tbody>
</table>

**Hyperparameter Search/Next Models**

Given the limited scope of the project, I primarily searched over 3 hyperparameters: the reward discount, the cost discount, and the learning rate. As such, using the 100,000 games of the first Deep Q agent, I trained several dozen models over these parameters. Given time constraints, after the first few I decided to curtail the remainder after 10 epochs before the testing phase. Below were some of the most promising models:

<table>
<thead>
<tr>
<th>Name</th>
<th>R_Y</th>
<th>C_Y</th>
<th>α</th>
<th>Vs Greedy</th>
<th>Vs Random</th>
<th>Combined loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1e-3</td>
<td>42.0%</td>
<td>84.8%</td>
<td>881.6</td>
</tr>
<tr>
<td>11</td>
<td>0.9</td>
<td>1</td>
<td>1e-3</td>
<td>41.0%</td>
<td>76.7%</td>
<td>1374.0</td>
</tr>
<tr>
<td>12</td>
<td>0.95</td>
<td>1</td>
<td>1e-3</td>
<td>56.9%</td>
<td>92.5%</td>
<td>323.3</td>
</tr>
<tr>
<td>13</td>
<td>0.95</td>
<td>1</td>
<td>1e-4</td>
<td>62.9%</td>
<td>93.7%</td>
<td>236.0</td>
</tr>
<tr>
<td>14</td>
<td>0.95</td>
<td>0.99</td>
<td>1e-4</td>
<td>63.3%</td>
<td>94.7%</td>
<td>193.2</td>
</tr>
<tr>
<td>15</td>
<td>0.95</td>
<td>0.95</td>
<td>1e-4</td>
<td>60.4%</td>
<td>94.5%</td>
<td>218.5</td>
</tr>
<tr>
<td>16</td>
<td>0.95</td>
<td>0.9</td>
<td>1e-4</td>
<td>61.0%</td>
<td>94.1%</td>
<td>231.0</td>
</tr>
</tbody>
</table>

While Model 14 with the small cost discount did perform the best, most of the models with cost discount performed very poorly for unknown reasons. As such, I thought it more prudent to continue iterating with Model 13, the best performing model without cost discount.

Model 13 was iterated as follows: 100k games were simulated, it trained over the simulated data for 30 epochs, it was tested for 10k games against random and greedy and another 100k games were simulated. Below is the performance over several of these cycles.

<table>
<thead>
<tr>
<th>Model 13 Iteration #</th>
<th>Vs Greedy</th>
<th>Vs Random</th>
<th>Combined loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>67.2%</td>
<td>94.4%</td>
<td>183.7</td>
</tr>
<tr>
<td>2</td>
<td>46.8%</td>
<td>89.4%</td>
<td>563.9</td>
</tr>
<tr>
<td>3</td>
<td>65.0%</td>
<td>95.0%</td>
<td>173.8</td>
</tr>
<tr>
<td>4</td>
<td>64.8%</td>
<td>95.1%</td>
<td>169.6</td>
</tr>
<tr>
<td>5</td>
<td>64.8%</td>
<td>94.7%</td>
<td>186.1</td>
</tr>
<tr>
<td>6</td>
<td>62.3%</td>
<td>94.2%</td>
<td>218.2</td>
</tr>
<tr>
<td>7</td>
<td>62.4%</td>
<td>94.1%</td>
<td>223.3</td>
</tr>
</tbody>
</table>

As none of these models had a combined loss score better than the second iteration of the original model, I did not move forward with human testing.

**Conclusion and Discussion**

Many of the algorithmic and hyperparameter choices in these models were made based on time and computational constraints. The reason that Model 13 was only trained for 30 epochs but simulated for 100k games was because I had access to two computing services: one with 4 vGPUs and one with 36 vCPUs. Simulations could be run in parallel very rapidly on the service with 36 vCPUs and so increasing the number of simulations was much less time intensive than increasing the training.

For Model 13, the train and dev costs continued to decrease with each iteration but after the fourth one the loss began to increase. This seems to imply that, although the estimations for Q* improved, the performance in game was worse. This could possibly mean that the values it was approximating were not good values to represent Q*. Ultimately, Model 13 did not do better than the original against the static agents and its combined loss score was, at its best, still worse than the linear algorithm.

I believe this was due to the lack of breadth and depth of the hyperparameter search. There were many more parameters to try such as the dropout percentage, more variations on the learning rate (I only tried 1e-3 and 1e-4), L2 regularization. Furthermore, several early models were 4-layer models, but those had issues with the losses exploding after 10-20 epochs possibly due to poor regularization; I did not have the chance to fully explore why these failed and instead proceeded with only 3 layer models.

Additionally, I did not implement an RNN over the history of moves, as I originally intended. Summing over the history looses the information of the specific patterns that the opponent had chosen and those patterns, I suspect, would have been very predictive of what cards the opponent had in its hand.

Finally, the human trials were done in a mobile app version of the game I developed for another class. Ideally, this app would have collected the human data so that the algorithms could train
against actual humans, but I also did not have the
time to implement this.

With only 200 games and a lot of variance
between games, it is difficult to say for certain how
well the algorithm actually performed. It did beat
some of the human players, at best scoring 66% 
victory, but human level performance is not
measured against the average but the best players.

If I had more time and resources, I would
conduct a more thorough hyperparameter search.
Then, I would keep multiple models through
several iterations to simulate those models against
each other instead of just against themselves, only
filtering some percentage each time. As a second
priority and depending on how much
computational resource I had available, I would try
to train an LSTM with the sequence of moves
instead of the summation over the history.

Ultimately, the best model with deep learning
did better than the best linear model. At the
moment, the mobile app version is available to
play against for Android and iOS if one downloads
the Expo app available in either app store uses this
link: exp.host/@leoni0000/zsy. In the future, a
tutorial will be added and perhaps human data can
be collected. At the moment, the game rules are
available in the appendix to this paper.

References:

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.,
Veness, J., Bellemare, M., Graves, A., Riedmiller, M.,
Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C.,
Sadik, A., Antonoglou, I., King, H., Kumaran, D.,
Human-level control through deep reinforcement
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 one 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, JJQKK, are valid patterns. 55566777788899 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 JJQQQ). 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.