Reinforcement Learning Applied to a Game of Deceit

Theory and Reinforcement Learning

Hana Lee
leehana@stanford.edu

December 15, 2017

Figure 1: Skull and flower tiles from the game of Skull.

1 Introduction

Skull is a simple game of deception played by 3-6 players. Each player receives four tiles. Three of these tiles depict flowers, with the fourth depicting a skull. At the beginning of a round, all players simultaneously choose one of their tiles and places it face-down on the table. Then play proceeds clockwise, with each player taking one of two actions: Add or Bet. If a player chooses Add, they place another tile face-down on top of their stack. If they Bet, they choose a number and from then onward, each player has a choice of two actions: Raise or Pass. If a player Raises, their bet (higher than the previous bet) replaces the previous bet. If a player Passes, they are out of the round.

Once all players but one have Passed, the player who made the last bet must turn over a number of tiles equal to their bet, starting with their own stack. If they turn over only flowers, they win 1 point. If they turn over a skull, they permanently lose one of their four discs (losing all four means that a player has lost the game). The first player to win 2 points wins the game.

2 Motivation

One of Skull's main game mechanics is bluffing: misleading other players in an attempt to trick them into flipping over your skull tile. In my experience, playing Skull without a willingness to lie most often results in a loss. While reinforcement learning has been applied with varying degrees of success to other board games[1] and there has been a great deal of research on adversarial machine learning agents, including deceptive ones[2], I am primarily interested in the application of machine learning to deception mechanics in games. I plan to train a reinforcement learning agent to bluff effectively in a game of Skull.

My primary goal is to evaluate the usefulness of reinforcement learning as a technique used to teach artificial agents how to lie and get away with it. I personally enjoy many games of deception including Skull, Mafia, Resistance, Secret Hitler, and more. One defining feature of these games is that they are inherently social; much of their value comes from the thrill of lying to other people, and the challenge of trying to unravel other people's lies.

While teaching artificial intelligences to lie has been an unintended outcome of some machine learning experiments (Facebook's recent attempt to teach bots the art of negotation, for example[3]), it is very much the intended outcome of my experiment. A significant barrier to the games of deception that I enjoy is the number of human players required; Mafia, for example, is not a game that can be played with one or even a handful of friends. Training a game-playing agent to lie effectively could not only make it possible to play a game of Mafia without a dozen friends on hand, but it could also open avenues to more realistic simulation of these games, leading to new strategies for winning.

3 Method

While Skull is a simple game, its mechanics are too complex to easily assign rewards for reinforcement learning. For the purposes of training an agent, I simplified the game to a 2-player variant that takes place over a single round. If the player who makes the last bet successfully turns over enough flower tiles, they win the game; otherwise, they lose. For simplicity, bets can only be raised in increments of one.

The next step after simplifying the game was to formulate the new, simpler version as a Markov decision process (S, A, $\{P_{sa}\}$, γ, R). S is the set of all possible states in the game, where a state holds the following information:

- Player 1's stack (the tiles face-down on the table)
- Player 2's stack (the tiles face-down on the table)
- The current bet
- Whether the game is over

A is the set of all possible actions that can be taken. These include:

- Add Skull (add a skull tile from the deck to the stack)
- Add Flower (add a flower tile from the deck to the stack)
- Bet (place a bet greater than the current bet and less than or equal to the number of tiles in all stacks)
- Pass (decline to raise the bet; this triggers the end of the game)

$\{P_{sa}\}$ are the state transition probabilities. I hardcoded these according to my knowledge of the game; the intention was for Player 2 (the reinforcement learning agent's opponent) to behave approximately as I would in a real game.

γ is the discount factor; for preliminary experiments, I set the discount factor to 1 (no discount).

$R : S \times A \times S \to \mathtt{R}$ is the reward function. Because the intention is to train the reinforcement learning agent to lie, I assigned a reward to each successful bluff. Bluffs are defined as follows: Player 1 (our reinforcement learning agent) takes a Bet or Raise action that it knows it cannot accomplish. The agent receives a small reward if Player 2 "takes the bait," i.e. takes the Raise action instead of "calling" Player 1 on the bluff by Passing and allowing them to flip over a skull. I also assigned a positive reward to game victory and a negative reward to defeat.

4 MDP

For the baseline learning algorithm, I modeled the problem as a regular MDP with full state knowledge available to the reinforcement learning agent (although the state probabilities I hardcoded still took into account that the "human" player has limited information).

After formulating Skull as an MDP and generating a state space, I implemented the policy iteration algorithm described in the CS229 "Reinforcement Learning and Control" notes for MDPs with finite, discrete state spaces. I made a few changes to the algorithm due to the nature of the problem; for actions that are not possible given the current game state, I "cascaded" to the next possible action. I also assigned rewards based on the tuple of current state, action, and next state, instead of only current state and action.

The state space contained 750 states. The policy iteration algorithm converged after about 7 iterations on average. I calculated the fraction of the time that the optimal policy found by the algorithm would dictate a bluff; that is, the reinforcement learning agent would decide that the best possible course of action was to make a bet that it knew it could not successfully fulfill. This turned out to be 22% (only counting states where it is possible to bluff, and not states when no bet or no bluffing bet are possible, such as a game over state).

Even when I removed the reward for successful bluffing or greatly increased it (up to 100x the normal reward for winning a game), the ratio remained at 22%. Changing the discount coefficient γ also did not change the bluff ratio. Adding an epsilon value ϵ to allow random walking of the action space during policy iteration greatly affected the number of iterations to convergence, but only changed the final bluff ratio a very small amount, from 22% to 24% (Figure 2).

I speculated that the ratio remains the same because the reinforcement learning agent knows the exact state of the opponent's stack at all times, and knows the probability of each bet be-

Figure 2: Results of policy iteration on a full-knowledge MDP.

ϵ	Iterations	Bluff Rate
0	7	0.22
0.05	19	0.23
0.1	86	0.23
0.15	311	0.23
0.2	2124	0.24

Figure 3: Results of policy iteration on a limited-knowledge pseudo-POMDP.

ϵ	Iterations	Bluff Rate
0	6	0.12
0.05	6	0.12
0.1	6	0.12
0.15	27	0.12
0.2	48	0.12

ing successful. I attempted to test the validity of this theory in the next section.

5 POMDP

A more accurate way to formulate the game of Skull is as a partially observable Markov decision process (POMDP) because the reinforcement learning agent does not have full knowledge of the state. It only knows the size of its opponent's stack, not its contents (i.e. the location or absence of the skull tile).

However, formulating Skull as a POMDP is not entirely straightforward. Generally, our observations are fixed for each possible state. That is, while the RL agent may observe that its opponent's stack contains 3 tiles, that observation remains the same regardless of whether the opponent has placed their skull tile on top of the stack, on bottom, or has not placed the skull tile at all. The RL agent must the therefore maintain a uniform belief distribution over every possible state according to the size of its opponent's stack.

I modified the MDP formulation so that the RL agent maintains a uniform belief distribution over every currently possible state at each step. For example, if the opponent's stack contains 1 tile, the belief distribution is uniform over the states in which the opponent's stack contains 1 tile, and zero everywhere else. This is not quite a traditional POMDP setup, but it captures the uncertainty of the agent about the exact features of the state.

The POMDP policy iteration algorithm converged sooner, in only 5 iterations, possibly due to the smaller space of the belief distributions (only 213 unique belief distributions versus 750 unique states). I again calculated the RL agent's bluff ratio and found it to be 12%. Again, this ratio remained relatively consistent regardless of how the bluff reward, discount coefficient γ, and random walk parameter ϵ were adjusted (Figure 3).

6 Analysis

After formulating Skull as both an MDP and a POMDP and calculating the RL agent's bluff rate in both problems, I found two major results. The first result was that the bluff rate remained unchanged through adjustment of the bluff reward in both problems. The second result was that the RL agent made significantly fewer bluffs when it was uncertain about the exact location of the skull in its opponent's stack.

The first result was initially surprising, given that I had theorized the unchanging bluff ratio was due to the RL agent's perfect knowledge of the state in the MDP. The result of the POMDP formulation indicates that even with uncertainty, the RL agent still does not take the additional rewards for bluffing into account. This could be because successful bluffs are strongly linked to game victory, meaning that in any scenario where the RL agent has a chance of bluffing successfully, that chance is equal to the RL agent's chance of winning the game, and the reward is therefore positive regardless of whether there is an additional reward for bluffing successfully.

On further reflection, this result makes sense. Skull is mechanically a very simple game if its deception aspects are removed, and its simplicity means that the result of a game between two experienced opponents is likely to depend almost entirely on their ability to make successful bluffs. The RL agent was able to "figure out" this part of the game's design from simple policy iteration.

The second result is less easily explainable and therefore more interesting. Why would the RL agent be less inclined to lie to its opponent if it is uncertain about the exact state of the

Figure 4: Accuracy of bluff detection in games against an RL agent (only best actions).

	RL Win	Human Win	Total
Correct	3	14	17
FP	1	1	2
FN	0	0	0
Wins	2	8	10

Figure 5: Accuracy of bluff detection in games against an RL agent (70% best action, 30% runner-up).

	RL Win	Human Win	Total
Correct	3	5	8
FP	3	1	4
FN	1	1	2
Wins	5	5	10

game? While the answer no doubt lies in the mathematics of the POMDP formulation, I was taken by the parallels between the RL agent's behavior and real human behavior that I have observed while playing games of deception.

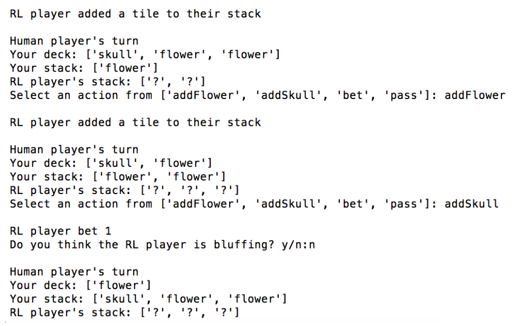
In games of Mafia, players typically start out with very little information, and are unlikely to make bold moves initially such as claiming a role different from their own. However, as the game goes on and players become more aware of others' identities and the exact state of the game, they become more confident and make bold claims that drastically alter the course of the game. Although this observation is more in the realm of social psychology than machine learning, I find it fascinating that the RL agent also seemed to possess this apparent human tendency to be honest when it is uncertain, and crafty when it is confident.

7 Evaluation

After obtaining an optimal policy from policy iteration on Skull as a POMDP, I decided to test the believability of the RL agent's lies by playing against it. I wrote a program that would allow me to play against the RL agent, who chose its moves according to the policy learned from the pseudo-POMDP policy iteration. Every time the RL agent made a bet, the program queried whether or not I believed it was a bluff. At the end of the game(s), the program tallied up the number of times I was correct, the number of false positives, and the number of false negatives.

Unfortunately the quantitative results of this analysis were not very useful initially, since the version of Skull I was playing against the RL agent was simple enough that I was able to quickly understand the RL agent's strategy and predict its moves in every game. After 10 games, I had won all but the first two and had correctly guessed whether the RL agent was bluffing 89% of the time (Figure 4).

Figure 6: Terminal output while playing Skull against the RL agent.



```
RL player added a tile to their stack

Human player's turn
Your deck: ['skull', 'flower', 'flower']
Your stack: ['flower']
RL player's stack: ['?', '?']
Select an action from ['addFlower', 'addSkull', 'bet', 'pass']: addFlower

RL player added a tile to their stack

Human player's turn
Your deck: ['skull', 'flower']
Your stack: ['flower', 'flower']
RL player's stack: ['?', '?', '?']
Select an action from ['addFlower', 'addSkull', 'bet', 'pass']: addSkull

RL player bet 1
Do you think the RL player is bluffing? y/n:n

Human player's turn
Your deck: ['flower']
Your stack: ['skull', 'flower', 'flower']
RL player's stack: ['?', '?', '?']
```

In order to make the games a little more interesting and harder to predict, I modified the policy iteration algorithm so that it selected the top two actions for each state. During each game, the RL agent performed the top action 70% of the time and the runner-up 30% of the time.

Playing against the RL agent using its new strategy was more challenging. I won only 5 out of 10 games, and it became significantly harder to guess whether the RL agent was bluffing. The accuracy of my guesses lowered to 57%, only slightly better than random guessing (Figure 5).

8 Limitations

While my experiment in teaching an RL agent how to bluff in Skull was entertaining and produced some surprising results, its actual research value is questionable due to the many limitations of the setup.

The primary obstacle in this kind of problem is the lack of an accurate model with which to train the RL agent. I had to hardcode the "human" player used in training in order to fake enough data for training; the alternative was playing against the RL agent myself hundreds or thousands of times, which is not really feasible for a 30 to 40-hour-long research project. Hardcoding the human player's moves means that the transition probabilities for the MDP are fixed and known, which reduces the value of exploration to zero. It's also a matter of opinion whether my estimates of the transition probabilities are even accurate; is a human player 20% likely to bluff on any given bet? 30% likely? My choice of these probabilities was largely arbitrary, so I was really training the RL agent to play against a computer with behavior programmed by a human, not a real person.

While I still believe teaching machine learning agents to lie is a valuable research goal (perhaps a controversial opinion), similar experiments should be done on games more complicated then my simplified version of Skull in order to yield interesting results. I chose Skull because the state space is small and the actions that can be taken are few, qualities which lend themselves well to a reinforcement learning problem. However, these very qualities mean that the opportunities for deception are scarce and relatively uninteresting. In a game like Mafia, there are hundreds of different ways a game could unfold due to the various deceptions of the participants, and even highly skilled players find it extremely difficult to see through the lies of others[4].

One promising future area of study is social learning as a way to train reinforcement learning agents to play games where data on previous games is unavailable or infeasible to use for training. Using social learning, one can initialize several different agents with different parameters (i.e. transition probabilities) and train them against each other. Previous attempts at social learning have produced RL agents that are able to out-perform agents trained solely by self-play (playing against themselves or agents with exactly the same parameters)[5]. One can extrapolate from these results and theorize that social learning might also outperform agents trained against a hardcoded human opponent, like our Skull player.

9 Code

All code for this project can be found in the SkullRL repository on Github (thequeenofspades/skullRL).

10 References

[1] Imran Ghory (2004) *Reinforcement learning in board games.* Technical Report CSTR-04-004, Department of Computer Science, University of Bristol.

[2] Jamie Hayes, George Danezis (2017) *Machine Learning as an Adversarial Service: Learning Black-Box Adversarial Examples.*

[3] Katyanna Quach (2017) *Facebook tried teaching bots art of negotiation – so the AI learned to lie.* The Register.

[4] Sergey Demyanov, James Bailey, Kotagiri Ramamohanarao, Christopher Leckie (2015) *Detection of Deception in the Mafia Party Game.* Department of Computing and Information Systems, The University of Melbourne.

[5] Vukosi N. Marivate, Tshilidzi Marwala (2008) *Social Learning Methods in Board Games.*