Applying Reinforcement Learning to the card game of Cheat
Teaching a Computer to Bluff

“There are three kinds of lies: lies, damned lies, and statistics.” - Mark Twain

Ronen Tamari
Hebrew University of Jerusalem
The Rachel and Selim Benin School of Computer Science and Engineering
Jerusalem, Israel

Erez Peterfreund
Hebrew University of Jerusalem
The Rachel and Selim Benin School of Computer Science and Engineering
Jerusalem, Israel

Abstract—Teaching a computer to play the card game Cheat (in Hebrew "שקרן") using reinforcement learning methods and probability analysis.

I. INTRODUCTION

Cheat (also known as "Bullshit" and "I Doubt It") is a card game where the players aim to get rid of all of their cards. Normally played with at least three players, it is often classed as a party game, and is a game of deception. We quickly realized that the game is highly complex since information is very limited and utilities of states are functions of multiple parameters, including opposing players' psychology and strategy, which are very hard to quantify. Last but not least, luck is also a factor, especially in the end game where moves are more limited. Nonetheless, we were attracted by the idea of "teaching" a computer agent to bluff, lie and suspect, since deception is a component of intelligence not normally associated with computers.

II. GAME RULES [1]

Normally, a standard pack of 52 playing cards is used, but the game can be played with multiple packs of cards and often includes the jokers as wild cards. A dealer is chosen and the cards are shuffled and dealt until all the cards are dealt. The first player is either the first player dealt to or sometimes in variants the first person with a specified card. Play proceeds in the order of the deal. The objective of the game is to be the first player to get rid of all their cards.

On their turn a player places a number of face-down cards into the middle of the table, from their hand, and makes a claim as to what those cards are (e.g. "two sevens"). The player must call either the same rank as the previous player, or one rank above or below, with aces counting both high and low. This claim may be a lie, and may have to be a lie if the player has no cards of the required ranks.

Once a player has made a claim, every other player has until the next player starts their turn to call "cheat", if they think the player's claim was a lie (the phrase called out is normally the same as the game name). When someone calls "cheat", the play stops and the pile of played cards is turned face-up. If the most recent cards were a valid play and the same as the claim, then the first player to call "cheat" has to add the entire pile to his or her hand. Otherwise, the player who made the false claim has to pick up the pile, and play continues.

The first player to empty their hand (and not lose a challenge on the final play) is the winner.

III. PROGRAM DESIGN & STRUCTURE

The program was written from scratch in Python, (GUI based on PyGame). We divided the game into a few principal classes:

A. Game: Handles the creation of a new game, holding current game state, managing the different agents and updating them with information, and recording of statistics.

B. GameState: Holds the current hands of all players, current pile, and the last cards played.

C. Agent: Handles the brains of each player - playing of cards (offense) and calling "liar" (defense). Each action of the agent alters the gamestate.

The main objective of the project was to design the learning agent, who we named Bluffbot.

IV. NOTATIONS

A. Offense:
By "offense", we mean the playing of a card/cards by a player on his turn.

B. Defense:
By "defense", we mean the phase in the turn when a player can choose to accuse the offensive player of lying.

C. Last declared card (LDC):
The current card on the table, played by the previous player. Or no card, if the pile is empty and the round is starting.
D. Moves

An offensive move is defined by two components: the "real" part, which is the card/s actually played, and the "said" part which is the card/s declared by the offensive player. We can then distinguish between 2 kinds of moves:

- True moves: When real = said.
- Lying moves: These are moves where real ≠ said.

V. GENERAL PROBLEM STRUCTURE

We can model this game as a POMDP, since given any current state we have a fixed number of actions available, but our agent can only partially observe the state it's in, due to the hidden information in other player's hands as well as their actions. The learning elements in offense and defense allow the agent to better assess what the current state is, and given that, the actions selected will be more optimal. In the two following sections, we will detail the learning process on offense and defense.

VI. OFFENSE ANALYSIS

A. Number of possible actions:

Assuming a hand of size H, playing up to c cards per turn, the amount of possible actions $A_{tot}$ is

$$A_{tot} = \sum_{i=1}^{c} \binom{H}{i}$$

For $c = 4$, $H = 13$ (the parameters of a standard game with full deck), $A_{tot} = 1092$. As a first step, we tried to simplify the problem by lowering $H$ and $c$. Also, since there is no importance to the suit (just the number), the size of the hand is effectively smaller. Since we wanted to tackle the full size problem with as few simplifying assumptions as possible, we developed strategies and heuristics in order to make an efficient search through the space of possible actions.

A few observations about types of actions:

- True moves: The number of true moves for any given turn is small, and bounded by $3 \cdot c$ where 3 is the number of legal card numbers (LDC+1,LDC,LDC-1) and $c$ is the maximum number of cards allowed to be played per turn.
- Lying moves: Lying moves are the dominant contributor to the large amount of possible actions, since any card or combination of up to $c$ cards is legal. Hence it is important to keep the amount of lying moves small. To do so, we can notice that when lying, many lies are equivalent in the sense that there is no clear advantage, for example, in getting rid of a 4 instead of a 5, when we are saying that we're playing a 10. The rule of thumb is that we want to lie with cards which are far away from the current declared card.

B. Lowering effective search space

To avoid checking many essentially equivalent moves, we maintain a "worst cards list" updated each turn according to the last card declared on the table. Cards are rated in terms of their current usefulness, for example if the last declared card is a 7, the worst cards would be those farthest from the 7, such as 1 or ace (due to cyclicity). The most useful cards would be those within a 1-card radius of 7 - 6,7,8. We also take into account the current direction we want to move- if the LDC is 5 and we have 1,2,7,8,9- the lowest rated cards would be the 1 and 2 since we want to move "up" towards the 7,8,9 group.

C. Types of lies

We can further distinguish between types of lies to prioritize one type over another:

- "Good Lies": When real ≠ said, but real ∈ hand. We call this a good lie since it is harder for the opponent to detect, since if he is using probability analysis this move will not be abnormal. The amount of good-true moves possible is equal to the amount of true moves.
- "Bad Lies": When real ≠ said, but real ∉ hand. This is a bad lie, since it is easier to detect by an opponent using probability analysis.

D. Example of legal actions from a given state

In summary, when creating the list of legal moves, we are doing the following:

For each type of said action allowed we first try to find a true move, if one is possible we also add a good lie corresponding to that move. Otherwise, we add a bad lie.

Let's say we're playing with a 20-card deck of 2 suits, and have the hand:

<table>
<thead>
<tr>
<th>Card#</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td># of cards in hand</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Where LDC = 5.

Then the "worst cards list" would be (1, 1), (1, 7), (2, 7), (1, 4), (2, 4), (2, 5), (1, 6), (2, 6) (where the first number is the suit and the second is the card number). This represents the fact that we want to get rid of the 1 first. One of our strategies (to be explained later) attempts to maintain a concentration of cards (low variance distribution) and this move fits that strategy. It is not necessarily the optimal move.

The actions to be considered would be as follows (each cell contains the real part of the move, while the row and column correspond to the said part). For example, in the table below, the cell marked in gray would correspond to the move:

"Play one Ace (1) and one 7, and say that they're two 4s" (lie!)
The Q-learning algorithm is a model-free reinforcement-learning algorithm that abstracts states and compares them by extracting features of (state, action) pairs, and there are too many states and actions to be considered - the amount of possible card configurations alone is upwards of 10^{11}. Also, similar approaches have been used for other bluffing-based card games.

Each turn, on offense our agent does the following:

1. Selects an offensive action (cards to play) \( a \) such that:
   \[
   a = \text{argmax}_a Q(s, a) \quad \text{with} \quad p = 1 - \epsilon \text{ (random action)}
   \]
2. Observes reward which is \( R(s, a) = |H_i| - |H_{i+1}| \)
3. Update Q:
   \[
   w_i \leftarrow w_i + \alpha \cdot \text{correction} \cdot f_i(s, a)
   \]
   \[
   \text{correction} = \left[ R(s, a) + \gamma \cdot Q(s', a) - Q(s, a) \right]
   \]
4. Decay alpha.

The \( \alpha \) and \( \epsilon \) values were chosen through trial and error, the final settings used were \( \alpha = 0.8, \epsilon = 0.1 \).

The Q-values of each state are calculated using the features of a given state multiplied by the weight assigned each feature (with the weight being learned over time):

\[
Q(s, a) = \sum_{i=1}^{n} f_i(s, a)w_i
\]

### E. The Approximate Q-Learning Algorithm

This model-free reinforcement-learning algorithm abstracts states and compares them by extracting features of (state, action) pairs, and there are too many states and actions to be considered - the amount of possible card configurations alone is upwards of 10^{11}. Also, similar approaches have been used for other bluffing-based card games.

Each turn, on offense our agent does the following:

1. Selects an offensive action (cards to play) \( a \) such that:
   \[
   a = \text{argmax}_a Q(s, a) \quad \text{with} \quad p = 1 - \epsilon \text{ (random action)}
   \]
2. Observes reward which is \( R(s, a) = |H_i| - |H_{i+1}| \)
3. Update Q:
   \[
   w_i \leftarrow w_i + \alpha \cdot \text{correction} \cdot f_i(s, a)
   \]
   \[
   \text{correction} = \left[ R(s, a) + \gamma \cdot Q(s', a) - Q(s, a) \right]
   \]
4. Decay alpha.

The \( \alpha \) and \( \epsilon \) values were chosen through trial and error, the final settings used were \( \alpha = 0.8, \epsilon = 0.1 \).

The Q-values of each state are calculated using the features of a given state multiplied by the weight assigned each feature (with the weight being learned over time):

\[
Q(s, a) = \sum_{i=1}^{n} f_i(s, a)w_i
\]

### F. Features used by feature extractor

1. **Nearest Largest Group**
   We tried to find the largest group of cards which doesn’t contain at most one card which is in the range (e.g. if we have in the hand: 1, 2, 4, 5, 6, 9, 10. When there are 1-13 cards with one suit, the largest group is 1-6). The feature favors actions which bring the pile closer to the closest edge of the group. This is a more local search than the Center of Mass feature to be detailed next.

2. **Distance from Center of Mass**
   We tried to use the physical term of center of mass, because of the cyclic way of cards laws. Therefore we formalized it as: m-number of the max card
   \[
   \text{hand}[i] - \text{number of cards of number } i, \text{ which the player has in his hand}
   \]
   \[
   CM = \sum_{i=1}^{m} \frac{\text{hand}[i]}{\text{CardsInHand}} \cdot \frac{2\pi}{m}
   \]
   We returned an inverse function of the distance, so the feature favors the action which minimizes the distance from the center of mass.

3. **Variance**
   We used the notion of variance to express the relative concentration of cards around the CM. In general a solid strategy is to keep the cards “bunched” in order to have flexibility in making true moves. This was calculated by
   \[
   \text{Var[Hand]} = \mathbb{E}[\text{distance}(\text{hand}[i], \text{CM})^2] \\
   \approx \frac{1}{\text{CardsInHand}} \sum_{i=1}^{m} \text{distance}(\text{hand}[i], \text{CM})^2
   \]
   This is an approximation since we’re assuming uniform distribution on card types. We returned an inverse function of the distance, so the feature favors the action which minimizes variance.

4. **IsGoodLie, IsBadLie, IsTrue, IsWin**
   Indicator variables indicating which type of action we’re trying (isTrue=1 if the move is true, etc). The idea is to try and learn the values of the various action types, which depend on the opponent we’re playing and game situation.

Another optimization of the Q-learning algorithm we used was using separate weight vectors for different situations (see performance analysis section).

### VII. DEFENSE ANALYSIS

The defense we found to be trickier than offense. The main problem is that most of the information necessary to make the decisions isn’t available. With 2 players, the problem of hidden information is less acute, since in the beginning of each round (when there’s no pile) we have full information, since we know that the opponent has exactly the cards we don’t.
rounds though, again things get much less clearer, since we don't know whether the earlier claims made were true.

With 3 or more players, lack of information is acute from the outset since there is no certainty about an opponent's cards except based on what we have (if we have 3 fours and he plays 2 fours, it must be a lie). This made it hard to abstract the game state with features and weight vectors as we had done for the offense, so we created a layered defense combining multiple approaches which performed well against other computer opponents.

A. "No-brainer" lie detection

The first layer of the defense is designed to protect against obvious lies, such as the example stated above. Or more formally, whenever the opponent declares the cards to be something he couldn't possibly have based on the cards in our own hand. This is because in general, the chance of the opponent having the right cards in the end of the game is small.

B. "Back-against-the-wall" accusations

The 2nd layer is defense against situations where an opponent has 3 or less cards (i.e. is close to winning). In this case we accuse him of lying with a probability inversely proportional to the amount of cards the opponent has left. This is because in general, the chance of the opponent having the right cards in the end of the game is small.

C. Don't accuse if it can help offense

The 3rd layer is a check whether to abstain from lying or this would help offensively. For example, we may think that the opponent is lying but since he played a card good for our own hand. This is most effective against a single opponent, and at the start of a round (when the pile is empty). At this point we know exactly what cards the opponent has.

D. Accusation based on prior history

The central check (if none of the above apply) is an attempt to assess the probability that the opponent is lying, based on his previous actions and playing style. Our agent maintains an "intelligence file" on each opponent. Each time an opponent is caught lying, we inspect the pile to detect the playing patterns of all players. This gives the opportunity to learn about the prior moves players made. In effect, we try to assess the probability

\[ P(\text{opp played } Y \text{ cards truthfully} \mid \text{opp has already played } X \text{ cards of that type this round}) \]

Of course, this probability is not only dependent on \(X\) but also other factors, such as the number of cards of that type other opponents have played, their tendency to lie, etc. The reason we chose this probability is that it is relatively easy to compute empirically, and also is a significant factor which we found to be relatively invariant for a given player. In addition, it is something a human cannot hope to do and utilizes the natural skills of the computer. Finally, we compared this method with others and found that it gave solidly improved results (see performance analysis section).

Following is a (real) example of the table we keep for each opponent. As can be seen, it is based on prior knowledge gathered over time of each given type of event.

<table>
<thead>
<tr>
<th># cards played this turn</th>
<th># cards of same number played this round</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>

Assuming for example that our opponent (whose history is stored in the given table) plays two 4s, and on his next turn another two 4s and finally on his third turn another two 4s (this obviously means he has lied, the question is when...), by checking the table in the gray cell we can predict an \( \frac{8}{8+2} = 80\% \) chance he's lying on his third turn, based on previous playing patterns.

E. Pile size factor

Finally, our agent also takes into account the current pile size. The logic behind this is that the risk of calling a lie is lower if the pile is smaller (since if we make a false accusation we only take a few cards). We used an exponential decay model for this kind of strategy, because we assumed that as long as the pile gets bigger, the risk gets bigger.

VIII. PERFORMANCE ANALYSIS

To test our learning agent (designated B for short), we played him against a few other AIs in varied scenarios to measure his performance.

A. Test AIs

1) Naivebot (N):

   a) Offense: Plays as many cards as possible truthfully. Tiebreakers decided by simple "nearest neighbor" heuristic. If we have a 4, 5, 10, .12 and can choose to play a 6 or 8, Naivebot will prefer the 6 since it's closest to the 5. If no cards can be played truthfully, Naivebot will attempt to lie using the same heuristic.

   b) Defense: Will call "liar" on random with a given probability \( p \in \{0.1, 0.15, 0.2, 0.25\} \) (randomly chosen at beginning of game). Will always call "liar" if the opponent is in winning position (puts down his last cards).

2) Bunkerbot (P): This agent was featured in a previous work [5] and could not be overcome by the learning algorithm detailed in the project, so we took it upon ourselves to meet the challenge presented by it.
a) Offense: Like Naivebot.

b) Defense: Will call liar randomly only 1% of the time, except for the end game (if the current offensive player has 3 or less cards in his hand) in which case Bunkerbot will always call "liar". This causes a sudden change in playing style which threatens to confuse our learning agent. More details on how we coped with this challenge in the results section.

3) Old Bluffbot (O):

a) Offense: Like Bluffbot.

b) Defense: Uses the version of defense without accusation based on prior history. We used this agent in order to evaluate the final defensive approach as compared to earlier attempts.

B. Simulation types

We ran hundreds of simulation games in multiple scenarios to test performance. Two main kinds of games were run:

- Head-to-head: These games are less prone to the luck of the draw since information concerning the other players' hand is higher than with more than 2 players.
- 4 player games: More challenging, and more prone to luck since information about opponents' hands is much more limited.

C. Results

1) Head-to-head: Generally, in the head-to-head simulations, the final version of Bluffbot dominated computer opposition. This is largely due to effective defense compared to random coin flipping for other bots. Fig. 1 summarizes the win percentage against the various AIs

a) Vs. Bunkerbot: This agent posed a surprising challenge, calling "liar" against any opponent with 3 cards or less in his hand. Since Bunkerbot doesn't call lies at all in other situations, by the end-game (3 cards or less) our learning agent learns the wrong lesson- that any lie is ok since it goes unpunished. Upon reaching the end-game the weight vector is so skewed in favor of lies such that Bluffbot sometimes cannot make the drastic adjustment needed in time. These games were usually very long since Bunkerbot is poor on offense (simple Naivebot AI), and became attrition matches of over 1000 rounds (against Naivebot, games averaged less than 100). We considered a number of approaches to beat Bunkerbot's defense, and eventually decided on the addition of a second weight vector to be used only in the endgame. Thus, each part of the game was learned separately. As can be seen in the graph, this optimization achieved good results. To check its performance against other playing styles we also tried it against other agents and mixed environments (against Naivebot and Bunkerbot simultaneously).

b) Vs Old Bluffbot: This test shows a dominance over the old version of Bluffbot, (the version without the defense which records the prior history). In head-to-head games, the increased accusation rate and accuracy of the improved defense showed its superiority over the earlier version without it.

![Fig 1: Head-to-head result summary (over 100 games)](image)

![Fig 2: Win %, 4 plyr. games, 300 games](image)
2) **4-player games:** The results are summarized in fig. 2 below. The three main tests were:

a) BOOO - Bluffbot vs 3 Old Bluffbots (without defense optimization). Here we can see the effects of partial information, increased reliance on luck, and the need to learn multiple opponents which all contribute to a lower win rate than the head-to-head games. The learning process takes much longer than for the head-to-head game, but by as can be seen in fig. 3, by roughly 100 games, Bluffbot holds around 35% of all total wins, with this percentage staying constant throughout the rest of the simulation. Fig. 4 shows the improved performance of the defense with the optimization—almost twice as many true accusations, and also a higher amount of accusations in total.

b) Mixed: Bluffbot vs 2 Naive and 1 Bunkerbot. (NBPN) In this environment, our agent gathers the relative majority of the wins, but still has considerable more trouble than against each of the agents head-to-head. The relative performance of the defense and offense is shown in Figs. 5,6.

c) Mixed: Bluffbot vs 1 Naive, 1 Bunkerbot and 1 Old Bluffbot (NBOP). Mixed environments in general proved to be the biggest challenge to our agent, and this scenario was the "most mixed". It was also the environment in which Bluffbot performed the worst, raking dead last against opponents which he could beat easily 1-on-1. The reasons for the poor performance in this environment require further research, but we believe it to be due in part to an "over-accusation" problem, since it can be noticed that the old Bluffbot is dominant in this scenario, and he makes far less accusations in total than the new Bluffbot.

3) **Testing value of experience:** This simulation we conducted in order to measure the value of experience, all other parameters being equal. We played an experienced Bluffbot against a fresh one, the results are displayed in Fig. 7,8. Clearly, the experienced agent has initial dominance, and with time the newbie agent gradually improves gradually.
Fig. 5: Scenario (b) Average accusation count & accuracy per game

Fig. 6: Scenario (b) Avg. Lying Success per game

Fig. 7: Win count over 60 games

Fig. 8: Newbie win % over 60 games
D. Games against humans: Not enough data exist to provide a detailed picture of the performance of our algorithm vs. human players. The main problem is the need to play multiple consecutive head-to-head games (estimated at least 50) to build a good learning agent, which can be very time consuming. A few test games were run using willing volunteers, against a learning agent who had trained against other Bluffbots. Results were inconclusive, but the authors can personally attest to having lost a match against 3 Bluffbots (2 Old Bluffbots and 1 new), and was also at times surprised by the agent's moves. This indicates that our learning agent could be developed into a viable opponent against humans.

The problem to be research is that it wins even that said algorithm performs well in a more general problem setting then achieved in a previous work [5] – we allow playing a variable number of cards per turn, and we also win against heuristics that defeated the previous attempt (see section on Bunkerbot).

Multiple player mixed environments proved to be the weakest point of our learning agent, see the section on future research for possible approaches to address this.

E. Conclusions

The performance analysis shows that our learning agent dominates simple algorithms in 1-on-1 matches, and still wins the relative majority of games against multiple AI opponents.

Our algorithm performs well in a more general problem setting then achieved in a previous work [5] – we allow playing a variable number of cards per turn, and we also win against heuristics that defeated the previous attempt (see section on Bunkerbot).

Multiple player mixed environments proved to be the weakest point of our learning agent, see the section on future research for possible approaches to address this.

IX. SUMMARY & FUTURE RESEARCH

Teaching our agent to play Cheat proved difficult, and incorporated quite a few challenges which we coped with, some for better and some less. All in all, we feel that Bluffbot learned to bluff naturally and can be quite adept at it.

What's for sure, is that we probably learned at least as much as our agent did.

The game's high complexity suggests many open problems to be researched in order to improve strategy.

A. Defense:

1) Q-Learning defense: Our current defense is learning in the sense that it uses past history to learn opponents plaining patterns. But it is not flexible in the sense that the various heuristics used are given predetermined weights. This approach shows its weakness when playing in mixed environments and could remedied in part by applying Q-learning to defense as well, in order to learn the various weights for each feature per opponent.

2) Using entropy & information gain: The main idea of this concept is to try to maximize the information gain we can gain from each player's offense in each round. We would take into account the probability of that the player is lying only when the entropy of the lying probability is high (for example >1/2). In training mode we would call "liar" to gain information in a way that would maximize the information gain of the given probability. We started implementing this strategy, and found out that it wins even in mixed environments which earlier versions failed in. The problem was that it wasn't stable enough and we haven't yet found a good solution to avoid getting stuck in local minima. We think that further research in this direction could bring this project another level up.

B. Combined Strategies: Our learning agent currently has only limited communication between the offense and defense. For us human players, defense and offense are naturally closely related- we may choose not to call "liar" since the card called is advantageous to us in terms of our offensive situation. Or we may choose to play cards on defense in order to corner our opponent in a situation where he must lie. These are just some of the ways to combine offense and defense, and adding the capabilities to Bluffbot would no doubt make him more formidable.

C. Performance against humans: This would require multiple games against same human to see if his/her playing style can be learned. The main problem here is finding a human that would be willing to play 100 games against Bluffbot. Playing against humans opens up many other lines of research, such as facial analysis to detect lies, analysis of playing against humans to see if his/her playing style and introducing more game modes we may choose to play cards on defense.

IX. SUMMARY & FUTURE RESEARCH

Teaching our agent to play Cheat proved difficult, and incorporated quite a few challenges which we coped with, some for better and some less. All in all, we feel that Bluffbot learned to bluff naturally and can be quite adept at it.

What's for sure, is that we probably learned at least as much as our agent did.

The game's high complexity suggests many open problems to be researched in order to improve strategy.

REFERENCES
