Play Liars-Bar With Humanlike AI Agents

Abstract

This project aims to build and systematically compare heuristic, search, and reinforcement learning (RL) agents for the incomplete-information game *Liars Bar*. We incorporate *Humanlike player modeling* and utilize *Linear Q-Learning* and *Deep Q-Networks (DQN)* for strategy learning. Our main contributions include introducing *environment randomization (Shuffle)* and *psychological trait probability encoding*, which address the sequential overfitting issue of DQN in multi-player games and significantly enhance the model's generalization ability.

1 Introduction

1.1 Motivation

Incomplete-information games pose significant challenges in artificial intelligence, requiring agents to make decisions with hidden information and under uncertainty. The motivation for this project is to explore AI capabilities in a specific high-risk, psychological-bluffing game—*Liars Bar*. This game not only tests card management and probability calculation but also crucially relies on psychological gameplay through bluffing and challenging, coupled with a "Russian Roulette"-style stochastic penalty. Developing AI agents capable of effective, human-like decision-making in such complex environments holds significant value for advancing AI applications in strategic games, automated negotiation, and dynamic decision-making systems.

1.2 Problem Definition

Liars Bar is a turn-based card game with the following core mechanics:

1. **Target Card**: Each round, a target card (Q, K, or A) is randomly designated.
2. **Card Declaration**: The current player must claim to play a number of target cards (they may actually play real or fake cards) and move the corresponding cards from their hand to the discard pile.
3. **Challenge Phase**: The next player may choose to challenge the declarer. If the challenge is successful (the declarer played fake cards), the declarer suffers a "gunshot" penalty. If the challenge fails (the declarer played real cards), the challenger suffers the penalty.
4. **Penalty Mechanism**: The penalty involves pulling the trigger of a 6-chamber revolver with only one live bullet. The bullet position is random, and the chamber rotates after each shot. If the bullet is fired, the player is eliminated.

5. **Victory Condition**: The last surviving player wins.

This project aims to design and implement AI agents that can participate in this game. The core challenge lies in the fact that agents must make decisions without seeing opponents' real hands (incomplete information). They must dynamically decide what cards to play (whether to cheat) and whether to challenge opponents based on game history, opponent behavioral patterns, and their own risk (bullet position).

2 Game Framework & Agent Player Design

2.1 Game Rules

The game framework (`game.py`) strictly implements the aforementioned rules. Each game supports 2-4 players. The main game loop `Game.play_round()` sequentially handles card playing, challenging, penalty resolution, and round resetting. The `GameRecord` class (`game_record.py`) fully records the state and actions of each round, providing data support for AI decision-making and post-game analysis.

2.2 Baseline Strategies

To establish performance baselines, we designed two rule-based agents (`player.py`):

2.2.1 SimpleStrategyPlayer

- Playing Strategy: Only plays real target cards or `Joker`. Plays none if unavailable. The strategy is completely honest.
- Challenging Strategy: Always challenges.
- Role: Serves as the most basic rule-based proxy for testing other agents' ability to handle "mindless challenging".

2.2.2 SmarterStrategyPlayer

- Playing Strategy: Prioritizes using `Joker`, then real target cards.
- Challenging Strategy: Decides whether to challenge based on a fixed random threshold (e.g., 0.5). Introduces randomness but lacks adaptability.
- Role: Represents a slightly optimized, simple probabilistic AI compared to the `Simple` strategy.

2.3 Humanlike Modeling

To simulate human psychological gameplay, we designed the `HumanLikeStrategyPlayer`. Its core idea is to adjust its own strategy based on real-time statistical characteristics of opponents.

1. Opponent Modeling:

- The agent internally maintains statistics for each opponent, including `total_plays`, `cheat_plays`, `total_challenge_opportunities`, and `challenges_made`.
- Dynamic calculation of opponent `cheat_rate` and `challenge_rate` via the `OpponentStatsManager` (`opponent_stats.py`) `get_features()` method.

2. Adaptive Playing Strategy:
   - Decision Basis: Primarily considers the next player's `challenge_rate`.
   - Strategy Logic: If the next player's challenge rate is high (> 0.5), tends to play real cards to avoid successful challenges. If low, attempts to play fake cards (bluff) with an exploration probability. The exploration rate decays over the game.
3. Adaptive Challenging Strategy:
   - Decision Basis: Primarily considers the previous player's `cheat_rate`.
   - Strategy Logic: If the previous player's cheat rate is high (> 0.5), tends to challenge; otherwise, randomly decides with an exploration probability.
4. Continuous Learning:
   - In the `reflect` method after each round, updates all opponents' statistics by parsing `round_action_info` and `round_result`.

This model transforms the agent from a static rule executor into one capable of observing, learning, and adapting to specific opponent styles, providing a strong opponent environment and feature engineering inspiration for subsequent RL agents.

3 Methods

3.1 Minimax Search

3.1.1 Design Philosophy

The `MinimaxAgent` is built upon the classic Minimax adversarial search algorithm, operating under the assumption that opponents always adopt the most detrimental strategy toward the current player. By constructing a game tree within a finite depth, the agent recursively evaluates all possible actions to select a decision that maximizes its own minimum expected utility.

3.1.2 State Modeling and Search Process

We utilize a `GameState` class to abstract the game environment, incorporating features such as the current hand, opponent hand counts, target cards, acting player, search depth, and revolver status (bullet and chamber positions).

During decision-making, the algorithm enumerates all legal actions. Depending on whether the current layer is a "Maximizing" or "Minimizing" node, it recursively calculates the evaluation values of subsequent states.

3.1.3 Evaluation Function Design

The evaluation function is critical due to depth constraints. It primarily balances two factors:

- **Play Utility**: Assigns high value to target cards, moderate value to `Jokers`, and penalties to non-target cards.
- **Risk Assessment**: Assigns significant negative scores when the bullet position is imminent in the revolver sequence.

The state evaluation value $V(s)$ is formally defined as:

$$V(s) = w_1 \cdot C_{target} + w_2 \cdot C_{joker} - w_3 \cdot P_{death} \tag{1}$$

where C_{target} represents the utility of target cards, C_{joker} is the value of the Joker, P_{death} accounts for the risk of elimination, and w_i are the respective weighting coefficients.

3.1.4 Limitations in Incomplete-Information Environments

Experimental results indicate that Minimax faces significant challenges in *Liars Bar*:

- **Information Asymmetry**: The agent cannot access opponents' real hands. The search relies on hypothetical approximations, causing many branches to deviate from the actual game state.
- **Stochasticity**: The randomness of bullet positions makes future returns difficult to predict via deterministic search. Increasing search depth does not consistently improve performance, as Minimax's theoretical premises are undermined in non-perfect information games.

3.2 Heuristic Search

3.2.1 Design Philosophy

The `HeuristicAgent` employs a decision-making process based on predefined heuristic rules. Unlike search-based agents, it does not construct a game tree. Instead, it directly scores all legal actions based on the current visible state and executes the action with the highest score, prioritizing computational efficiency and implementation simplicity.

3.2.2 Limitations and Challenges

While the `HeuristicAgent` offers advantages in stability and speed, its quality is capped in complex psychological environments:

- **Static Nature**: Heuristic rules are static summaries of experience and fail to dynamically model opponents' hidden information or evolving strategies.
- **Exploitability**: In a game defined by bluffing, fixed rules are easily exploited by adaptive opponents. While locally rational, the heuristic approach struggles to maintain a consistent advantage in a shifting strategic landscape.

3.3 Reinforcement Learning

In our implementation, we explicitly define a structured state dictionary and then derive a fixed-length feature vector for RL agents, ensuring compatibility with both linear function approximation and deep networks.

3.3.1 State & Feature Engineering

State. The environment state is represented as a structured dictionary whose fields differ slightly between the *play phase* and the *challenge phase*. Core fields include: round index (`round_id`), target card type (`target_card` $\in \{Q, K, A\}$), current bullet position (`current_bullet`), hand composition counts (`q_count`, `k_count`, `a_count`, `joker_count`), total hand size (`hand_size`), and the number of cards that can serve as the target (`target_count`). In the challenge phase, we additionally include the opponent's claim (`claimed_cards`) and an opponent profile vector (`opponent_features`).

19-dim feature vector. We extract a fixed 19-dimensional feature vector $\phi(s)$ from the state to support both Linear Q-Learning and DQN. The features cover: round id (1); target card one-hot (3); bullet/risk encoding (3) including `current_bullet`, `6-current_bullet`, and a start flag; target-card sufficiency (3) including `target_count`, `target_count/hand_size`, and `has_target`; hand size and empty flag (2); joker count and joker availability (2); claimed cards (1); opponent psychological profile (4): `cheat`, `challenge`, `aggression`, `confidence`.

3.3.2 Action Space

We define a **discrete** action space using one-hot encoding:

- **Play actions:** an N-dimensional one-hot vector indexing a fixed set of *play templates* (i.e., allowable "card-count / truthfulness" templates).
- **Challenge actions:** a 2-dimensional one-hot vector: `challenge` vs. `no_challenge`.

Thus, the total action dimension is $N + 2$. In practice, we apply an **illegal-action mask** to rule out actions that are infeasible under the current hand/state, restricting decisions to legal actions.

3.3.3 Reward Design

Rewards are dominated by **game outcomes** (win/loss) and **survival / penalty results**. To accelerate learning and prevent degenerate policies, we add **reward shaping**: successful and reasonable challenges (and surviving the current round) receive positive reward, while incorrect challenges and high-risk actions that lead to failure receive negative reward. To discourage overly conservative, passive play, we also add a small penalty for excessive conservatism that causes the agent to become "stuck" in a losing posture.

3.3.4 Linear Q-Learning Agent

We implement a linear function approximation variant of Q-learning, where the action-value is:

$$Q(s, a) = \mathbf{w}_a^\top \phi(s),$$

with one parameter vector $\mathbf{w}_a$ per discrete action. The agent selects actions via ϵ-greedy exploration and updates parameters using the TD target:

$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'), \quad \delta_t = y_t - Q(s_t, a_t), \quad \mathbf{w}_{a_t} \leftarrow \mathbf{w}_{a_t} + \alpha \delta_t \phi(s_t).$$

This approach is fast, interpretable, and sample-efficient, but its representational capacity is limited and may underfit complex psychological interactions.

3.3.5 Deep Q-Network (DQN)

To capture richer, nonlinear decision patterns, we further adopt a DQN[1] agent that approximates the optimal action-value function with a neural network:

$$Q(s, a; \theta) \approx Q^*(s, a).$$

We employ two standard stabilizers:

- **Target network:** a periodically updated network with parameters θ^- is used to compute a stable TD target

$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-).$$

- **Experience replay:** transitions (s_t, a_t, r_t, s_{t+1}) are stored in a replay buffer $\mathcal{D}$ and sampled uniformly to break temporal correlations during training.

The network is trained by minimizing the TD error:

$$\mathcal{L}(\theta) = \mathbb{E}\Big[(y_t - Q(s_t, a_t; \theta))^2\Big].$$

Exploration is also handled via ϵ-greedy to balance exploration and exploitation during learning.

4 Optimization: Addressing Overfitting

4.1 Problem Identification

During training, we observed that the DQN agent became highly dependent on a fixed player order. When the player order was swapped at evaluation time, the win rate dropped substantially compared to the training-time performance, indicating **sequential overfitting**: the agent was implicitly memorizing positional patterns rather than learning transferable opponent behaviors.

4.2 Proposed Solution

To mitigate this issue and improve generalization, we introduced two complementary optimizations:

Environment Randomization. We shuffle (randomly permute) the player order every other epoch. This breaks spurious correlations between absolute player index and opponent behavior, forcing the policy to rely on observable signals rather than fixed seat/order identities.

Opponent Profiling. We incorporate global historical statistics into the observation by encoding opponent behavior tendencies as features, e.g., `cheat_rate`, `challenge_rate` (and related attributes such as aggression/confidence proxies). These features are appended to the agent's feature set so the policy can condition actions on *how an opponent behaves* rather than *where the opponent sits.*

4.3 Effectiveness

Empirically, these changes shift the learned strategy from "remembering the order" to "learning behavior patterns." After applying shuffling and opponent profiling, evaluation under permuted player orders shows a significantly improved win rate, demonstrating stronger robustness and generalization.

5 Experiments & Results Analysis

5.1 Experimental Setup

To comprehensively evaluate agent performance, we designed multiple sets of experiments:

1. Training Environment: Used `RLTrainer` to train `DQNAgent` in a comparative setup: one group with fixed player positions (`position_test_fixed`) and another with randomly shuffled positions each game (`position_test_shuffled`). Opponents were `["humanlike", "smarter", "smarter"]`, trained for 400 episodes.
2. Evaluation Environment:
   - Cross-Matchup Testing: Used `game_analyze.py` to conduct large-scale (e.g., 500 games) pairwise matches among agents like `HeuristicAI`, `MiniAI` (Minimax), `SimpleAI`, `Smarter`, and `QLearner`, counting win rates (as shown in Figures 1, 2, and 3).
   - Training Process Monitoring: The trainer automatically recorded and plotted key metric curves, including train/evaluation win rates, exploration rate (`Epsilon`) decay, and opponent average cheat and challenge rates (as shown in Figures 4 and 5).

5.2 Performance Comparison

Experimental results clearly reveal the performance hierarchy of different agents and the effectiveness of optimization measures.

1. Heuristic vs. Search Agent Comparison (based on Figures 1, 2, and 3): Figures 1, 2, and the command-line statistics (Figure 3) show the performance of `HeuristicAI` and `MiniAI` (Minimax) against other baseline agents:

- Against `SimpleAI`: `HeuristicAI` achieved an overwhelming victory (80.2% win rate), and `MiniAI` also performed well (54.7%). This is because `SimpleAI`'s fixed "always challenge" strategy is easily predicted and exploited; `HeuristicAI` and `MiniAI` can find countermeasures through rules or search.

- Against `Smarter`: Both win rates dropped significantly (`HeuristicAI`: 40.3%, `MiniAI`: 34.1%). The random challenging introduced by `Smarter` increases game uncertainty, posing a challenge to heuristic and limited-depth search algorithms that rely on deterministic scenarios.
- Mutual Matchup: `HeuristicAI` vs. `MiniAI` win rate was 56.8%, and `MiniAI` vs. `HeuristicAI` was 43.2%, indicating close strength but `HeuristicAI`'s rule design held a slight edge in these tests.
- Against `QLearner`: A key finding: both had a low win rate against `QLearner` (referring to an early or insufficiently optimized Q-learning agent) over 500 games. This preliminarily indicates that even a basic RL agent can learn strategies surpassing traditional heuristic and search methods, especially against fixed-strategy opponents.

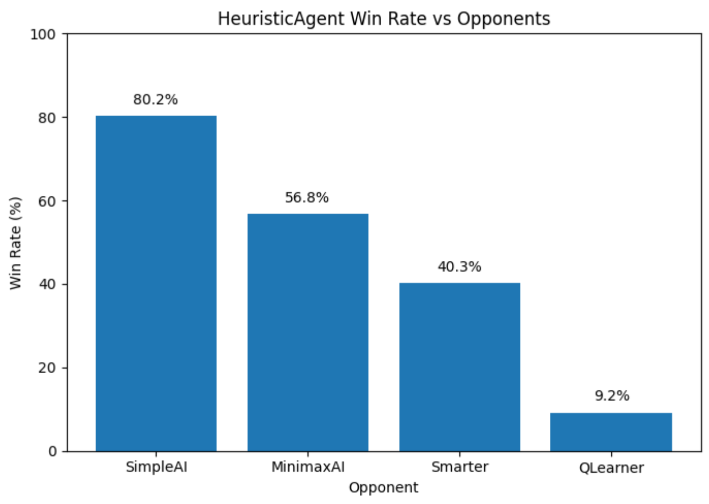


Figure 1: HeuristicAgent win rates against different opponents

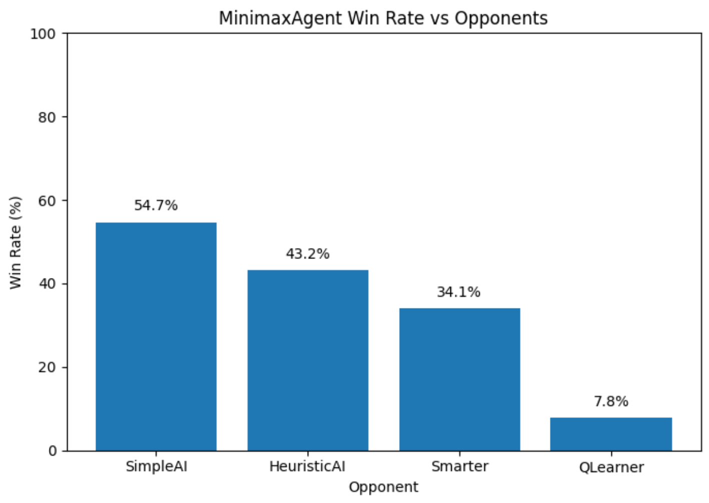
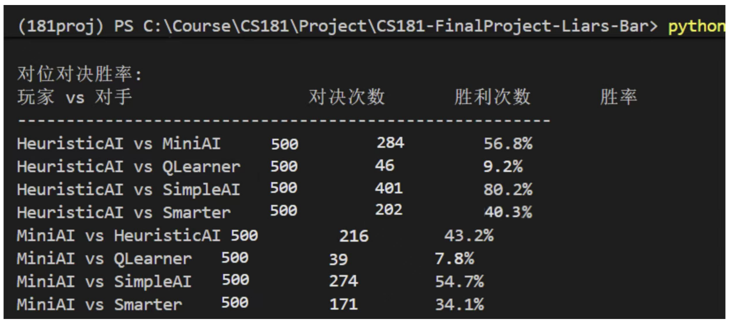


Figure 2: MinimaxAgent win rates against different opponents



```
(181proj) PS C:\Course\CS181\Project\CS181-FinalProject-Liars-Bar> python

对位对决胜率:
玩家 vs 对手                  对决次数      胜利次数      胜率
--------------------------------------------------------
HeuristicAI vs MiniAI      500        284         56.8%
HeuristicAI vs QLearner    500        46          9.2%
HeuristicAI vs SimpleAI    500        401         80.2%
HeuristicAI vs Smarter     500        202         40.3%
MiniAI vs HeuristicAI 500         216        43.2%
MiniAI vs QLearner    500        39         7.8%
MiniAI vs SimpleAI    500        274        54.7%
MiniAI vs Smarter     500        171        34.1%
```

Figure 3: Detailed matchup statistics from command-line output

2. Reinforcement Learning Training Optimization Comparison (based on Figures 4 and 5): Comparing the training curves of `position_test_fixed` (Figure 4) and `position_test_shuffled` (Figure 5) clearly shows the decisive impact of position shuffling on model generalization:

- Train Win Rate: Both quickly rose to a high level of about 0.8, indicating that models could find effective strategies within their respective training environments.
- Evaluation Win Rate - The Key Metric:
  - Fixed Position Model (Figure 4): The evaluation win rate fluctuated drastically and remained low (peaking at only ∼0.1), forming a huge "generalization gap" with the 0.8 train win rate. This clearly indicates severe overfitting: the model merely memorized strategies for a specific seating order rather than learning general gameplay logic.
  - Position-Shuffled Model (Figure 5): Although the evaluation win rate still fluctuated, its peak was significantly higher (reaching ∼0.4) and overall level closer to the train win rate. This demonstrates that by forcing the model to face different seating arrangements each game, it was compelled to learn decisions based on player identity and behavioral features (provided by `OpponentStatsManager` as `cheat_rate`, `challenge_rate`, etc.), thereby gaining strong generalization ability.
- Opponent Behavior Convergence: In both figures, opponent average cheat rate and average challenge rate decreased and stabilized with training. In the shuffled model (Figure 5), initial rates were higher, possibly due to more chaotic strategies during early exploration. Their final stabilized values (cheat rate ∼0.01, challenge rate ∼0.52) reflect the equilibrium behavioral patterns opponents (`HumanLike` and `Smarter`) converged to under random position pressure.
- Exploration Decay (`Epsilon Decay`): In Figure 4, exploration started from a lower value and decayed quickly to the minimum, while in Figure 5, it started from a higher value and underwent a longer decay process. This suggests that in a more complex and variable environment (random positions), maintaining longer and higher exploration is necessary to find robust strategies.

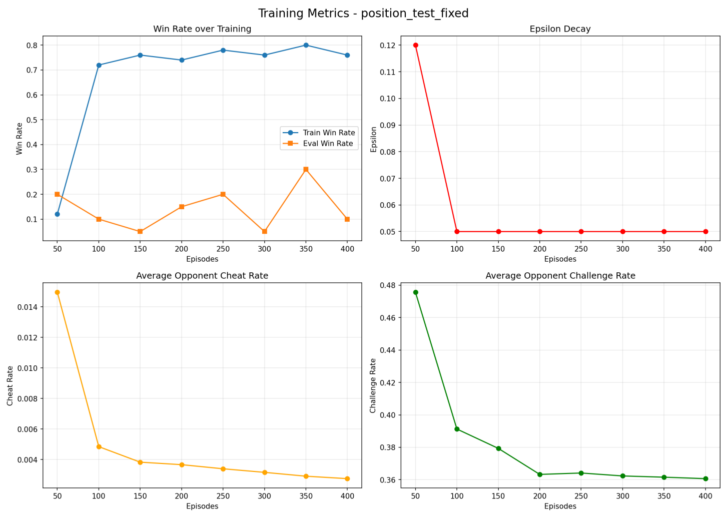


Figure 4: Training metrics for fixed position model

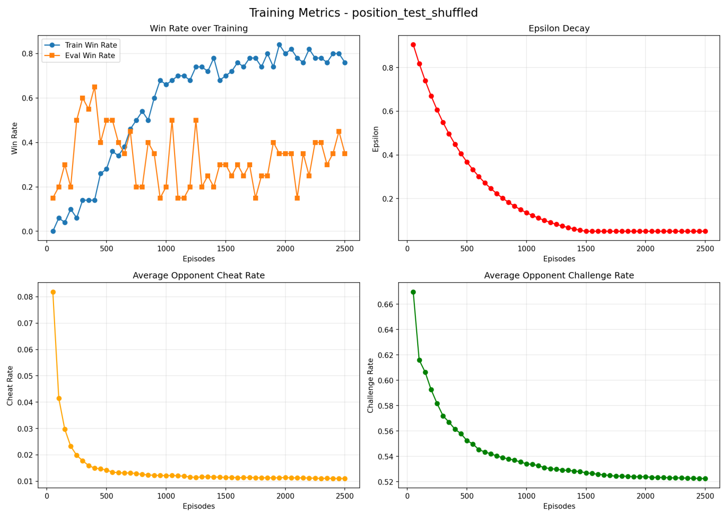


Figure 5: Training metrics for shuffled position model

6 Conclusion

This project systematically developed and evaluated AI agents for the psychological bluffing game Liars Bar, demonstrating that while deep reinforcement learning holds strategic promise, it suffers from position overfitting in static training. By implementing player position randomization and opponent psychological trait encoding, we forced agents to learn abstract behavioral patterns rather than memorizing fixed seating arrangements, increasing evaluation win rates from 10% to 40%. Our findings highlight that robust AI for human-like psychological games requires integrating behavioral modeling, feature engineering, and environmental randomization—a framework applicable to intelligent negotiation and dynamic decision-making systems.

External Resources

- Libraries: PyTorch, NumPy, tqdm, matplotlib.

- Code Repository: https://github.com/LYiHub/liars-bar-llm

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.