FAI2025_FINAL

B11902091 姚權維 2025/6/15

1 Task Description

The objective of this project is to implement a Texas Hold'em AI to compete in the CSIE Casino.

The **game settings** are as follows:

- Each player has 3 available actions: `call`, `raise`, or `fold`.
- The small blind is 5 chips, and the big blind is twice the small blind.
- Each player starts with an initial stack of 1,000 chips.
- After 20 rounds, the player with the most chips remaining is declared the winner.
- There is no upper limit on the raise amount, players can go all-in at any time.
- Players must take action within 10 seconds each turn, or treated as a `fold`.
- Players must take a valid action, or treated as a `fold`.

The CSIE Casino competition consists of **three stages**:

1. **Baseline Competition**: The AI plays one-on-one matches against seven predefined baseline agents (baseline1 to baseline7).
2. **Round-Robin Tournament**: Each player is randomly assigned to a group of six, facing five opponents. Rankings are determined by the number of matches won.
3. **Single-Elimination Stage**: The top 32 players with the highest cumulative chips (summed over five matches) advance to the knockout round. The top 4 players receive bonus points for their final grades.

In this project, we focused exclusively on the Baseline Competition stage, allowing us to control the environment and directly evaluate the performance of different AI strategies. All algorithmic adjustments and experimental evaluations were specifically tailored to this setting.

2 Baseline and Motivation

During the lecture on the history of AI, our professor introduced expert systems — symbolic AI frameworks popular in the 1980s that relied on handcrafted rules and logic to make decisions. It were also my first exposure to AI back in middle school, often depicted humorously in memes.

"AI" in the early days — basically just a bunch of if-else rules

Therefore, I chose to use `expert_player.py` as the **baseline** for my experiments, so that I could better identify potential issues in parameter tuning or implementation of other methods. For the more advanced approaches, I implemented `mcts_player.py`, which leverages the **Monte Carlo Tree Search (MCTS)** algorithm, and `deepq_player.py`, which is based on **Deep Q-Learning**.

3 Experimental Methods

3.1 Expert System

The expert system uses manually crafted rules to guide decisions based on hand strength and game stage.

- **Pre-flop**: A custom function assigns a strength score (0–10) to hole cards, considering pairs, high cards, suitedness, and connectivity. The AI adjusts its aggression based on position: more aggressive on the button/small blind, and more defensive on the big blind, raising or calling according to hand strength and opponent actions.
- **Post-flop**: The AI evaluates combined hole and community cards to classify hands (monster, strong, one pair, draws). It bets or raises aggressively with strong hands, slow-plays occasionally to avoid predictability, and calculates pot odds to decide on calling or semi-bluffing with draws. Bluffing occurs rarely in dry boards without prior bets.

Below is the result of running `expert_player.py`:

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Run 7**	**Run 8**
Score	30.5	27.8	30.4	29.3	29.8	27.8	28.4	31
Baseline 1	5	5	5	5	5	5	5	5
Baseline 2	5	5	5	5	5	5	3	5
Baseline 3	5	5	5	5	5	5	3	5
Baseline 4	5	5	5	5	5	5	5	5
Baseline 5	3	5	3	5	5	5	5	5
Baseline 6	2.5	2.4	5	2.4	1.8	2.4	2.4	3
Baseline 7	5	2.4	5	1.9	5	3	5	5

The results were quite impressive, especially considering that expert systems are often dismissed as outdated or simplistic. This demonstrates that well-designed rule-based heuristics can still be highly competitive in controlled environments such as the baseline competition.

3.2 Monte Carlo Tree Search (MCTS)

Our MCTS agent simulates many random game continuations to estimate the expected value of each action. The process includes:

- **Selection & Expansion:** All legal actions are expanded at the root; UCB1 guides which branch to explore.
- **Simulation:** For each action, the agent randomly completes unknown cards and uses `HandEvaluator` to determine win/loss outcomes.
- **Backpropagation:** Results are propagated to update win and visit counts.
  After simulating within a fixed time budget, the action with the most visits is selected.

Below is the result of running `mcts_player.py`:

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8
Score	30.5	29	25.5	33	25.7	32.5	30.5	21.7
Baseline 1	5	5	5	5	5	5	5	5
Baseline 2	5	5	5	5	5	5	5	2.4
Baseline 3	5	5	5	5	5	5	5	5
Baseline 4	5	3	2.5	3	2.3	5	5	3
Baseline 5	2.5	3	3	5	3	2.5	2.5	2
Baseline 6	3	3	2.5	5	3	5	5	1.8
Baseline 7	5	5	2.5	5	2.4	5	3	2.5

Although MCTS is typically known for its strength in strategic planning, the results here present a more nuanced picture. In approximately half of the runs, `mcts_player.py` outperformed `expert_player.py`, demonstrating its potential in handling complex decision-making scenarios. However, in some cases, its performance fell short, likely due to **high variance in simulation outcomes** or **insufficient hand strength filtering** during the early stages of the game.

3.3 Deep Q-Learning (DQN)

Deep Q-Network (DQN) approximates the Q-function using a deep neural network, enabling effective decision-making in complex.

- **State Representation**: A 14-dimensional input vector captures hand strength, draw potential, game stage (one-hot encoded), stack/pot sizes, call cost, pot odds, and position. All features are normalized.
- **Action Space**: We discretized actions into five options: `fold`, `call`, `raise 0.5 pot`, `raise 1 pot`, and `all-in`.
- **Network Architecture**: A 5-layer neural network with `BatchNorm`, `Dropout`, and `LeakyReLU` activations is used as the Q-network.
- **Training Setup**:
  - **Experience Replay**: Transitions are stored and sampled randomly to stabilize training.
  - **Reward Function**: Based on chip change per action, with a large terminal reward for winning/losing.
  - **Double DQN**: Reduces overestimation using separate policy and target networks.
  - **Epsilon-Greedy**: The agent gradually shifts from exploration to exploitation.

I have trained the model for 10 full rounds of **baseline competition**. The model will be saved and loaded from `MODEL_PATH`. However, the model showed no clear signs of convergence across training cycles. Even after enlarging the network and tuning parameters, the performance remained unstable. This suggests that the state space in this game may be too complex for vanilla DQN to effectively learn.

4 Results and Discussion

We conducted experiments for `expert_player.py`, `mcts_player.py`, and `deepq_player.py` in the **Baseline Competition**, each executed **8 times** to obtain a reliable average performance.

All experiments were performed on `ws5.csie.ntu.edu.tw` using a Conda environment with the package versions matching the TA-provided `requirement.txt`.

> For full reproducibility, please refer to our GitHub Link. Before executing, please carefully read the `README.md`.

Below is the result of their performance in the Baseline Competition, averaged over 8 runs:

	Expert	MCTS	Deep Q
Score	29.375	28.550	22.425
Baseline 1	5.000	5.000	4.000
Baseline 2	4.750	4.675	3.750
Baseline 3	5.000	5.000	4.000
Baseline 4	4.500	3.600	2.375
Baseline 5	2.738	2.938	3.250
Baseline 6	4.038	3.5378	2.425
Baseline 7	3.350	3.800	2.625

According to the experimental results shown as above, `mcts_player.py` and `expert_player.py` demonstrated comparable overall performance in the baseline competition. However, as shown in the detailed results from Sections 3.1 and 3.2, the expert system exhibited **greater stability and consistency** across all runs. This consistency ultimately led to our decision to adopt `expert_player.py` as our final agent.

Interestingly, based on informal discussions with classmates, it appeared that MCTS was widely considered the strongest baseline. Motivated by this, I invested the majority of my time optimizing the `mcts_player.py` implementation. I even generated custom `game_log.txt` outputs to trace the agent's decision-making and identify potential weaknesses for targeted improvements.

Despite these efforts and several heuristics introduced to address early-phase misplays, the performance plateaued during the later stages of the project. Additionally, experiments with `deepq_player.py` did not yield better results either.

5 Conclusion

Given the experimental findings and comparative performance analysis, we ultimately

$$\boxed{\text{selected} < \text{expert\_player.py} > \text{as our final agent}}$$

Details of the expert system implementation are as below:

- **Pre-flop**: decision-making is driven by the `_calculate_hole_card_strength` function, which quantifies the strength of the two hole cards on a scale from 0 to 10.
  - **Scoring Heuristics:**
    - **High Pairs:** `JJ` and above score 10, `88` to `TT` score 8.
    - **Ace Holdings:** `AK` scores 9, suited `AQ–AT` score 7, off-suit `AQ–AT` score 6, and suited `A2–A9` score 4.
    - **Suited Connectors:** High cards like `KQs`, `QJs` score 7; others with a gap ≤ 4 score 3.
    - **Off-suit Connectors:** Only high-card connectors like `T9o`, `JTo` receive 1 point.
    - All other combinations score 0.
  - **Positional Strategy (Heads-up):**
    - **Small Blind (SB):** Plays aggressively. Raise 75% of the pot if strength ≥ 8; raise 50% if strength ≥ 2; otherwise, limp.
    - **Big Blind (BB):** Plays defensively.
      - If opponent limps: raise if strength ≥ 3.
      - If opponent raises: re-raise if strength ≥ 7; call if ≥ 4; if pot odds are favorable (< 40%), speculative hands with strength ≥ 1 will also call.
- **Post-flop**: This phase incorporates public card evaluation and more sophisticated tactics.
  - **Hand Strength & Action:**
    - **Monster Hands (Full House or Better):** Bet 75% of the pot aggressively. On the flop, a 20% chance of slow play by just calling is added for deception.
    - **Very Strong Hands (Flush, Straight, Trips, Two Pairs):** Always bet 75% of the pot.
    - **One Pair:** If unchecked, value bet 50% of the pot. If facing a bet, only call when pot odds < 30%.
  - **Draws & Bluffing:**
    - **Draws:** `_calculate_draw_potential` estimates "outs." If the drawing odds are favorable compared to pot odds, the agent will call. For strong draws with 8+ outs, there is a 15% chance of **semi-bluffing** with a standard raise.
    - **Pure Bluffing:** When no one bets and the board is dry, the agent bluffs with a 5% chance by betting 40% of the pot.

- **Advantages:**
  - **Strong and Stable Performance:** As demonstrated in experiments, it was the top-performing AI in our tests.
  - **Transparent and Controllable Logic:** Every action is traceable to a clear rule, making it easy to debug, analyze, and iterate.
  - **Highly Efficient:** Decisions are made instantly with minimal computation, completely unaffected by time constraints.
- **Disadvantages:**
  - **Rigid Strategy, Vulnerable to Exploitation:** Fixed rules mean opponents can learn and exploit patterns (e.g., betting sizes tied to hand strength).
  - **Lacks Adaptability:** The AI does not adjust its strategy based on opponent behavior, treating aggressive bluffers and passive callers the same.
  - **High Maintenance Overhead:** As more scenarios are considered, the rule set becomes increasingly complex and harder to maintain or scale.

6 Reference

- lecture slides
- Gemini 2.5 Pro
- https://www.pokerstrategy.com/strategy/fixed-limit/pre-flop-hand-categories/
- https://medium.com/pyladies-taiwan/reinforcement-learning-%E9%80%B2%E9%9A%8E%E7%AF%87-deep-q-learning-26b10935a745