Reinforcement Learning for Agentic AI Applications

Learning to Win: Reinforcement Learning and the Autonomous Discovery of Optimal Blackjack Strategy

Ekene Nick Iheanacho

Flutter International

Introduction

Blackjack is among the most popular casino card games worldwide, renowned not only for its simplicity but also for the depth of strategy it allows. Although basic strategies to minimise the house edge are well documented, the dynamic and probabilistic nature of the game offers room for further optimisation through intelligent decision making.

Recent advances in artificial intelligence, particularly Reinforcement Learning (RL), have opened up innovative opportunities for creating autonomous systems capable of making optimal decisions in uncertain and evolving environments.

Reinforcement learning, a core methodology within Agentic AI, AI systems capable of acting autonomously and adaptively, is particularly suited to the challenges presented by blackjack. This is because RL naturally addresses sequential decision-making tasks where the outcomes depend on both immediate actions and future decisions.

In this report, I detail the development and training of a Reinforcement Learning agent designed to autonomously discover and adopt optimal blackjack strategies through self-play over 50,000 episodes. My aim is not only to demonstrate the practical applicability of RL but also to illustrate how an agentic approach can lead to improved performance and insightful strategies beyond conventional rule-based play.

Specifically, I used Q-learning, a widely used RL algorithm, to allow the agent to learn the best possible actions through exploration and exploitation iteratively. Through this approach, the agent autonomously developed strategies that approach the theoretical optimal play.

This report provides a concise review of existing research, followed by a detailed description of the methodology and experimental results.

Methodology

0.1. System Architecture

The developed system comprises four main components:

- **Game Logic Core**: Implements the rules and mechanics of blackjack, including card management, dealing, player and dealer actions, and outcome determination. It supports standard blackjack rules such as the dealer standing on all 17s and a 3:2 payout on player blackjack, but in this case, more ”reward”.
- **Reinforcement Learning (RL) Agent**: An autonomous decision-maker using the Q-learning algorithm. The agent observes the current game state, selects actions (Hit or Stand), and learns from rewards based on game outcomes over many episodes.
- **Visualisation and User Interface (UI)**: Built using Pygame, this component provides a real-time graphical representation of the game state, including cards, scores, and key metrics like win rate and epsilon value, allowing monitoring of the agent's learning progress.
- **Simulation Controller**: Orchestrates the training loop over 50,000 episodes, managing the flow of games, synchronizing between the game logic and UI, and enabling controls to start, pause, and reset the simulation.

0.2. Game Logic Implementation

The blackjack environment was designed with modular classes for clarity and flexibility:

- **Card and Deck Classes**: Cards are defined by rank and suit, with numerical values assigned appropriately (e.g., face cards as 10, Aces as 1 or 11). The simulation is configured to use a single 52-card deck, which is reshuffled for each game to ensure reproducibility via seeded shuffling.
- **Hand Class**: Represents the player's or dealer's hand, with methods to calculate hand value dynamically accounting for Aces (soft vs. hard hands). The `has_usable_ace` flag is a critical part of the agent's state representation.
- **Game Flow**: The `BlackjackGame` class manages the deal sequence, player and dealer turns, and determines the result (win, loss, or push) according to standard casino rules.

0.3. Reinforcement Learning Agent Design

The agent was implemented using the Q-learning algorithm with the following details:

- **State Space**: Each state is a tuple $(player\_sum, dealer\_upcard\_value, usable\_ace)$ representing the player's hand total, the dealer's visible card value, and whether the player has a usable ace (soft hand). This compact representation captures the essential strategic information.
- **Action Space**: Two discrete actions are available: `0` for Stand and `1` for Hit.
- **Reward Function**: Designed to reinforce winning outcomes:
  - Win: +1.0
  - Player Blackjack (natural 21): +1.5
  - Loss: -1.0
  - Push (tie): 0.0
- **Q-learning Algorithm**:
  - The Q-table is implemented using a dictionary with default zero values, storing expected rewards for each state-action pair.

  - An epsilon-greedy policy balances exploration and exploitation, with epsilon decaying from 1.0 to 0.01 over the training episodes.
  - The Bellman equation updates Q-values after each action using learning rate (α) and discount factor (γ) hyperparameters.

0.4. Training Setup

- The agent trained over 50,000 episodes of simulated blackjack games.
- Hyperparameters were set for steady learning progress:
  - Learning rate (α): 0.05
  - Discount factor (γ): 0.95
  - Epsilon (ϵ) decay schedule: Exponential decay from 1.0 to a minimum of 0.01, with a decay rate of 0.99995 per episode.
- Performance was tracked through real-time metrics including win rate, total wins/losses, and epsilon value.

0.5. Evaluation Metrics

The agent's performance was evaluated based on its ability to converge toward known basic strategy benchmarks. Win rate progression and Q-values for key states were monitored to validate learning and policy quality.

0.6. More explanations

The initial stages of choice is in the epsilon-greedy action selection, which is determined by a value (epsilon) which decays over time (reduces). It determines which actions the agent makes. At the beginning, since epsilon is high, the agent makes random actions, but with the Q table and update rule, it learns and convergences to the best strategy.

The Bellman's equation is a mathematical foundation for updating the Q-value.

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

The state-action pair is updated based on the reward, the discount factor, and the maximum expected future reward for the next state.

The complete equation used in updating Q-value is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big)$$

The Q table is the most important part of q learning. For this solution this is how our Q-table dictionary looks:

```
q_table = {
    # Key: A State Tuple                  # Value: A NumPy array of Q-values for each action
    # (player_sum, dealer_up, usable_ace)   [Q-value for Stand, Q-value for Hit]
    # ---------------------------------   ---------------------------------------
    (17, 7, 0)                          : np.array([-0.1498, -0.3613]),
    (12, 4, 0)                          : np.array([-0.3286, -0.3526]),
    (18, 10, 0)                         : np.array([-0.2184, -0.5717]),
    (11, 7, 0)                          : np.array([ -0.4866, 0.2844]),

    # ... and so on for every state the agent has ever encountered ...

    # A new, unseen state would be created on-the-fly like this:
    (14, 8, 1)                          : np.array([ 0.0, 0.0 ])
}
```

This is also more effective as we are using defaultdict, which doesn't raise a key error creates zeros for a new state The state space represents the player's current sum, the dealer's upcard, and whether the player has a usable ace (a boolean). This state is used to decide whether to hit or stand (the action). Each action is assigned a Q-value, which reflects how good that action is in that state.

Initially, the Q-value is set based on what happens after taking that action — whether the player wins, loses, or pushes. I assigned rewards as 1.0 for a win, -1.0 for a loss, 0.0 for a push, and optionally 1.5 for a blackjack.

The Q-value update combines the immediate reward with the discounted expected future reward. The discount factor (a number between 0 and 1) controls how much future rewards influence the current Q-value. Since each blackjack game is unique and short, the discount factor is high to prioritise immediate rewards.

Specifically, the immediate reward plus the discount factor multiplied by the maximum Q-value of the next state forms the target Q-value. The temporal difference is calculated by subtracting the old Q-value from this target. Finally, the temporal difference is multiplied by the learning rate, which controls how much the agent updates its Q-value based on new experiences.

0.6.1. Q-Learning Update Example

Consider the state where the player has 11, the dealer shows 7, and the player does `HIT` (action = 1):

$$\text{State} = (11, 7, 0)$$

- Initial Q-value:
$$Q(s, a) = Q((11, 7, 0), \text{HIT}) = 0.20$$
- Action taken: `HIT`
- Outcome: Player draws a 10, resulting in a hand value of 21 and wins the game.
- Reward:
$$r = 1.0$$
- New state:
$$s' = \text{None (terminal state)}$$
- Update rule:

$$\begin{aligned}
\text{old_q_value} &= 0.20 \\
\text{target_q_value} &= r + \gamma \max_{a'} Q(s', a') = 1.0 + 0 = 1.0 \\
\text{TD error} &= \text{target_q_value} - \text{old_q_value} = 1.0 - 0.20 = 0.80 \\
\text{learning rate } \alpha &= 0.05 \\
\text{update amount} &= \alpha \times \text{TD error} = 0.05 \times 0.80 = 0.04 \\
\text{new Q-value} &= \text{old_q_value} + \text{update amount} = 0.20 + 0.04 = \boxed{0.24}
\end{aligned}$$

By repeating this process thousands of times for every possible situation, the Q-values slowly converge towards their true long-term expected values, and the agent's policy becomes the optimal strategy.

Thought process

0.7. Reflective Development Process

When I received the invite on Monday, the first question I asked myself was, *"What can I build in four hours that is both enough and reflective of my goals within this industry?"* I was at the gym at the time, but I quickly decided to leave early. I realised the challenge wasn't about creating complex systems; instead, it was about generating a strong, viable idea rapidly. The majority of my initial time was spent brainstorming. I first asked myself, *What exactly do you want to do?"* I immediately thought, *Something AI-related,"* and quickly remembered, *Of course, it's a GenAI role."*, But that felt somewhat vague. I refined the question further: *What specific domain am I building for?"*, The answer became clear—gambling and casino. Given that the company was Flutter International and PokerStars was mentioned explicitly by the manager, the gambling domain felt perfectly suited. I recalled a previous interview where I'd discussed my reinforcement learning (RL) project. Given that most Agentic AI applications involve reinforcement learning to facilitate autonomous decision-making, RL became a natural fit. However, the primary challenge remained—time constraints. *"How can I realistically build this within four hours?"* I was confident in my knowledge of Q-learning and understood that with careful prompting and leveraging existing tools, I could rapidly prototype something meaningful. Reflecting on my process from a previous RL project, I knew it would be best to begin with the user interface (UI), then integrate the underlying logic. I sourced UI assets for a blackjack game via Google, leveraging images generated through Google AI Studio. My detailed prompts ensured precision and clarity, resulting in UI elements such as the main table felt, wooden player rail, card backs, chips, buttons, money box, and helper icons. The meticulous prompts ensured consistency and aesthetic realism in these assets:

- **Main Table Felt:** Create a primary green playing surface. It should mimic the texture of casino felt, not perfectly smooth, but with subtle variations, fibers, and a slight directional nap. The color should be a rich, deep emerald green, with a very subtle, soft gradient from slightly darker at the edges to marginally lighter towards the center, suggesting gentle illumination.
- **Wooden Player Rail:** Create a curved, polished wooden barrier where players sit and chips are placed. This should have visible wood grain, a warm brown tone (medium-dark), and significant glossy reflections/highlights, especially along its top edge, suggesting a polished surface reflecting light. It needs to appear thick and solid, with a clear sense of depth. The curve should be smooth and consistent, matching the arc shape in the original image, should be transparent (alpha channel) (think of it as a placement on a green background but this image would be without the green background) Roughly 1800-2000 pixels wide and 300-400 pixels tall. The actual wood part will be less. Ensure enough transparent padding around the arc for precise placement, The area above and below the wooden rail should be transparent so the felt background shows through, It would be placed above the image generated with this prompt "Create a primary green playing surface. It should mimic the texture of casino felt, not perfectly smooth, but with subtle variations, fibers, and a slight directional nap. The color should be a rich, deep emerald green, with a very subtle, soft gradient from slightly darker at the edges to marginally lighter towards the center, suggesting gentle illumination." should have an white background and be transparent.
- **Card Back:** Create a the generic back design of a playing card. It should be rich and detailed, like a standard casino card back (e.g., intricate pattern, logo, or solid color with a textured feel).
- **Chip:** Create distinct image files for each chip denomination you plan to use (e.g., $1, $5, $25, $100, $500). Each chip should be rendered as a 3D-looking disc. Show depth (sides of the chip), subtle texture on the top surface, clear highlights, and realistic drop shadows that make them appear to rest on the table. The colors should be vibrant and distinct (e.g., red, blue, green, gold/yellow, black). The denomination value (number) should be clearly visible on the chip face. Render them slightly angled, as if viewed from above at a table. It should all be in a grid so it can be cropped to make individual images. Transparent background.

- **Blackjack title:** Create the prominent ”BLACKJACK” title at the top of the screen. Use a strong, legible, and slightly stylized sans-serif font. The color should be pure white. Apply a very subtle drop shadow beneath the text to lift it slightly from the background. Transparent background.
- **Money box:** Create a blue rectangular box at the bottom center displaying the player's funds. It should have rounded corners, a smooth blue gradient (darker at top, lighter at bottom, or vice versa), and a subtle inner shadow/bevel effect to give it depth. Nothing should be written on it. Transparent background.
- **Dollar sign Icon:** Create a crisp, white dollar sign icon. Transparent background.
- **Buttons:** Create a generic button base that will be used for Normal State: Rounded rectangular shape, warm orange color (like the original). Apply a subtle gradient (e.g., slightly darker top to lighter bottom) and a gentle outer shadow to give it a 3D, pushable appearance. And Hover State: Same shape, but with a noticeable visual change indicating interactivity (e.g., slightly brighter orange, a more pronounced glow/shadow, or a subtle border). Both in a grid. All buttons would not have text. Transparent background.
- **Helper buttons:** Create the gear (settings) and question mark (help) icons. These should be clean, white, and anti-aliased, matching the minimalist style of the original image's icons. Transparent background.
- **Card images source:** `https://opengameart.org/content/playing-cards-vector-png`

The initial pygame equivalent of the mini table was quickly prototyped from a basic image (Figure 1). With further refinement, the fully realised UI was completed (Figure 2). With the UI firmly established, I transitioned back to reinforcement learning logic. Through systematic prompting, I ensured clarity and accuracy, avoiding hallucinations by separating the UI creation and RL logic into distinct interactions. My prompt—*”Given the RL logic and the UI, dynamically implement a simple blackjack system assuming all required images are available”* led to a fully functional prototype. Additional features, such as dynamic win rate calculations and periodic win rate displays, provided essential performance metrics. Finally, I implemented analytical code and created a Jupyter notebook for visualising the learning progress and win rates. This methodical and reflective approach, clearly outlined in my initial written plan (see Figures 4 and 5), allowed me to efficiently produce a cohesive working solution that was exactly in line with my objectives.

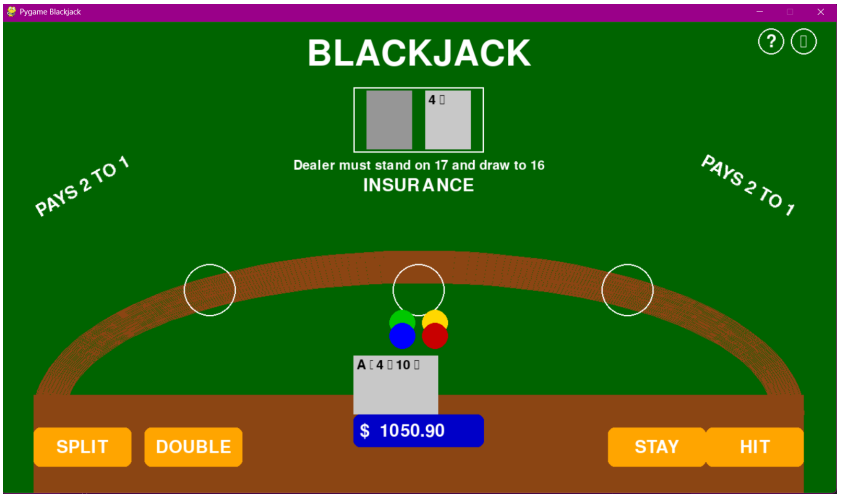

Figure 1: Initial Pygame Equivalent Table

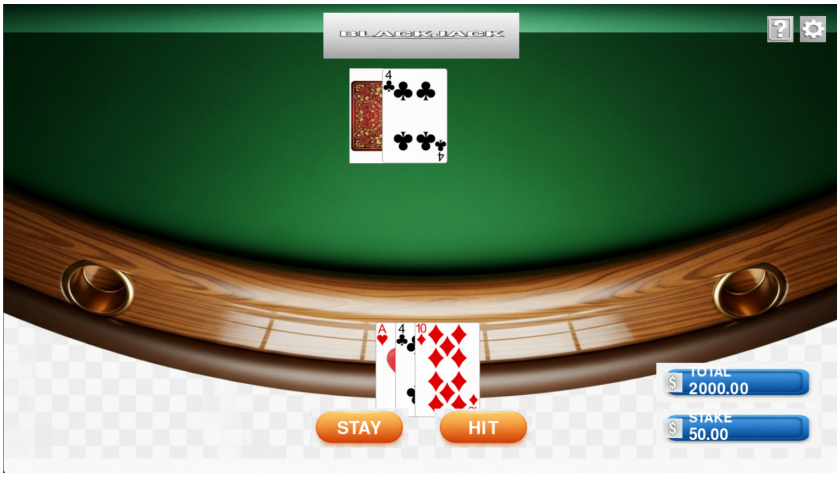

Figure 2: Final Blackjack UI in Pygame

Figure 3: Main Blackjack Table Image

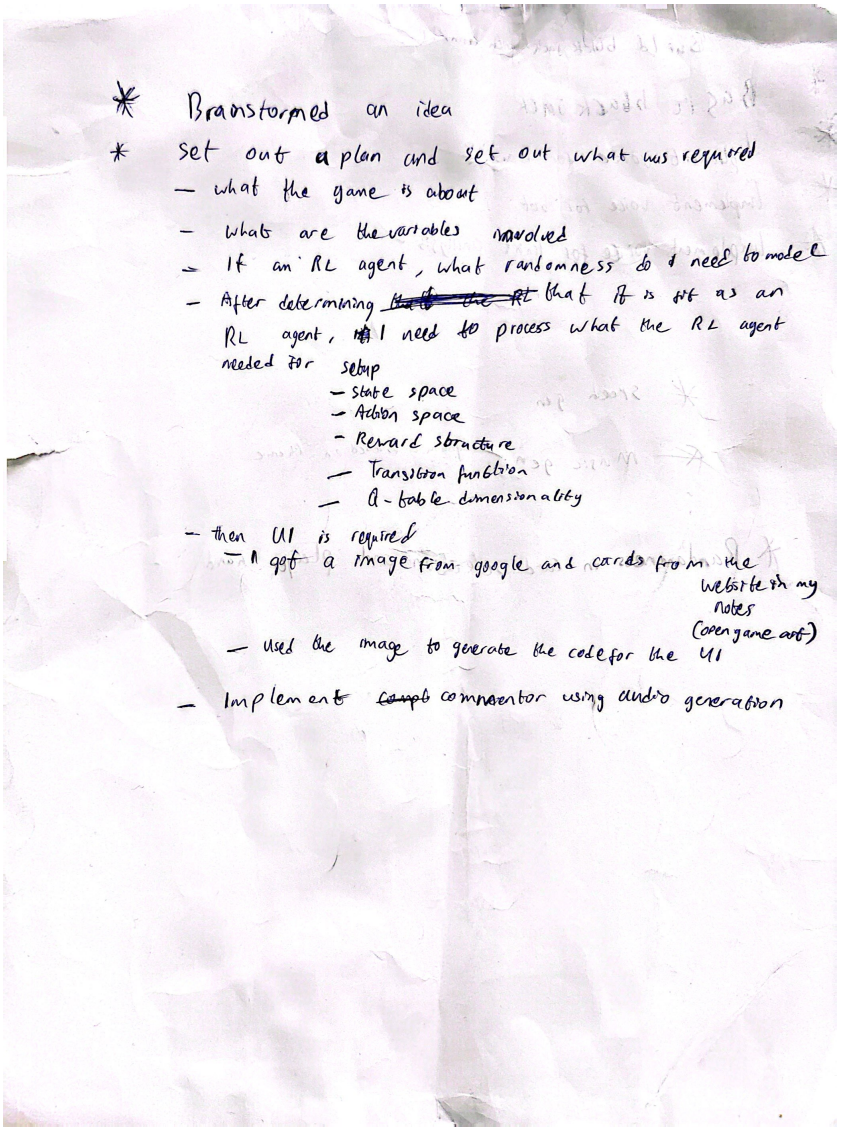

Figure 4: Initial Written Plan (Front)

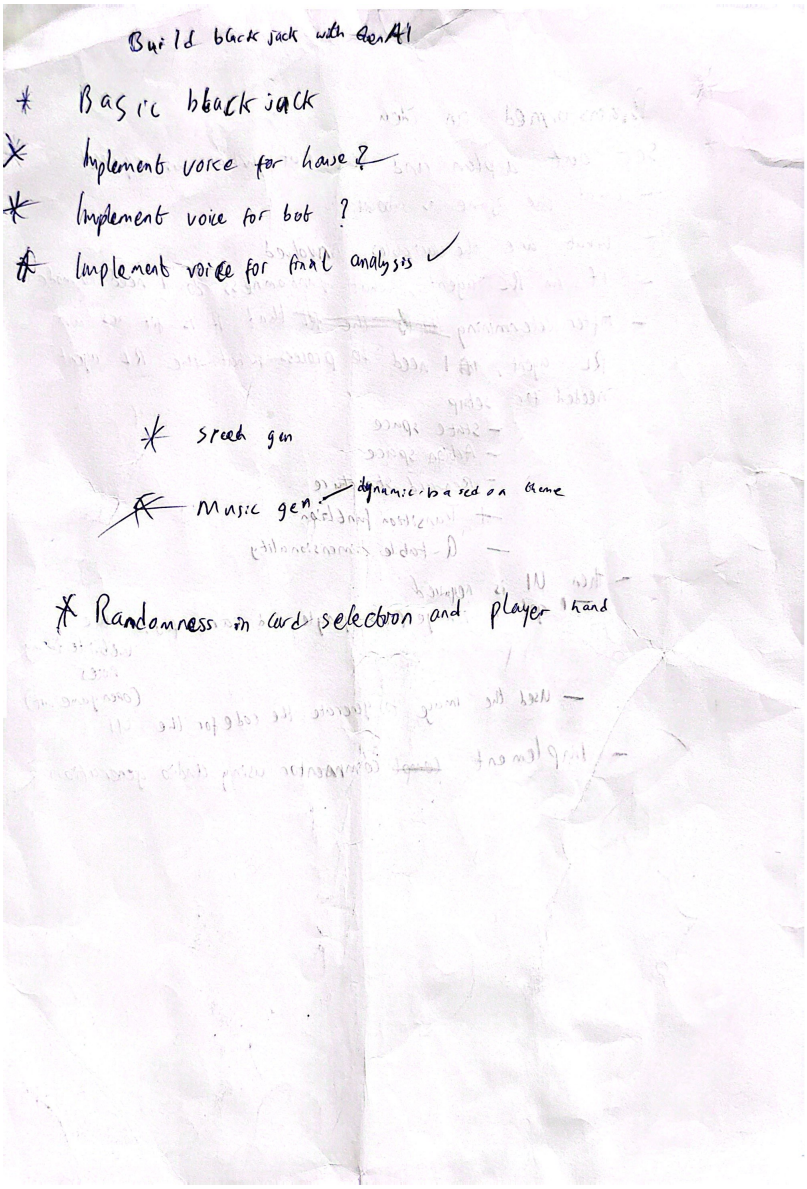

Figure 5: Initial Written Plan (Back)

0.8. Analysis and Results

The learned policies for both hard and soft hands are visualized in Figures 6 and 7 respectively. These heatmaps display the agent's preferred action—*Hit* or *Stand*—for different combinations of player hand totals and dealer upcard values.

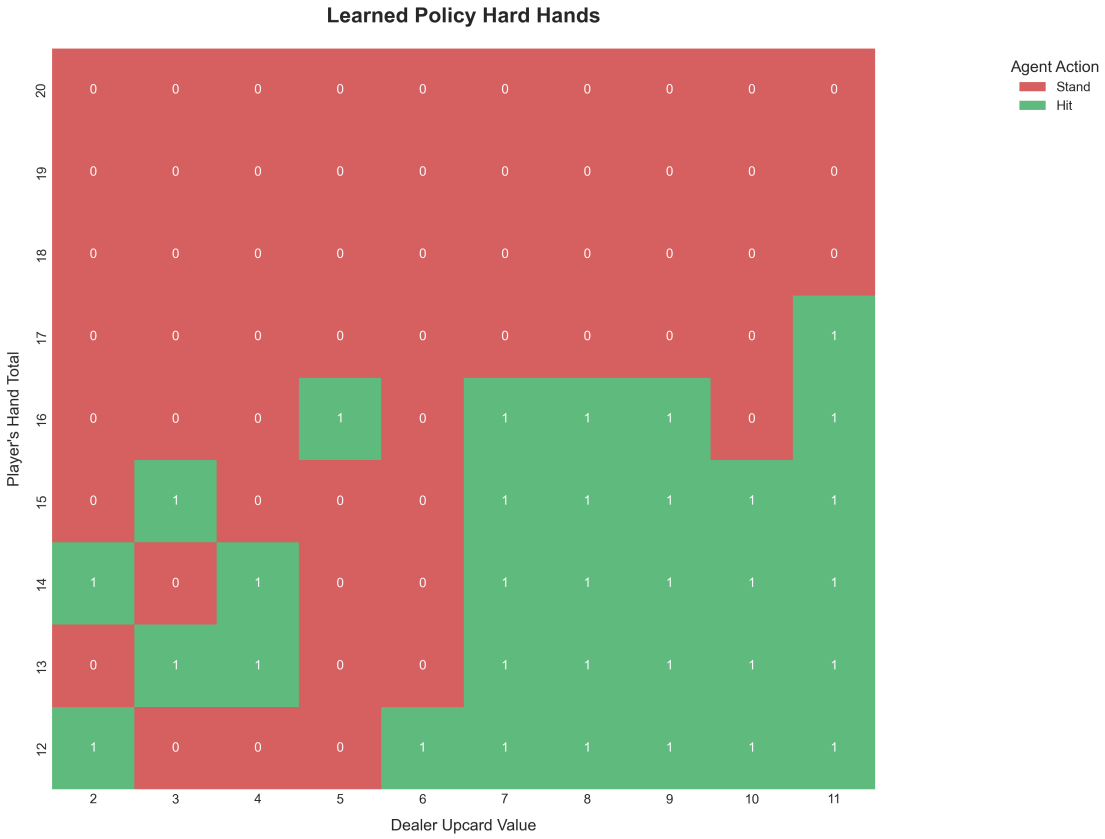

Figure 6: Learned Policy for Hard Hands: The agent's actions align closely with known blackjack basic strategy. Red blocks indicate standing, while green blocks indicate hitting.

The agent's training progression, shown in Figure 8, illustrates a steady improvement in the win rate over 50,000 episodes. The blue line tracks the agent's interval win rate, which rises towards and eventually stabilizes near 43%. This matches the expected win rate achievable using the perfect basic blackjack strategy, typically around 42–43%. The red dashed line marks this benchmark for reference.

These results demonstrate that the reinforcement learning agent successfully approximates the optimal blackjack strategy. The agent's decisions reflect the core principles of basic strategy, which is known to reduce the house edge to roughly 0.5% to 1%. Consequently, the agent achieves a player win rate in line with theoretical expectations.

Further insight can be drawn from the Q-table values for selected states (Table 1). For example, in a state where the player has 11 and the dealer shows 7, the agent values the *Hit* action at 0.2488,

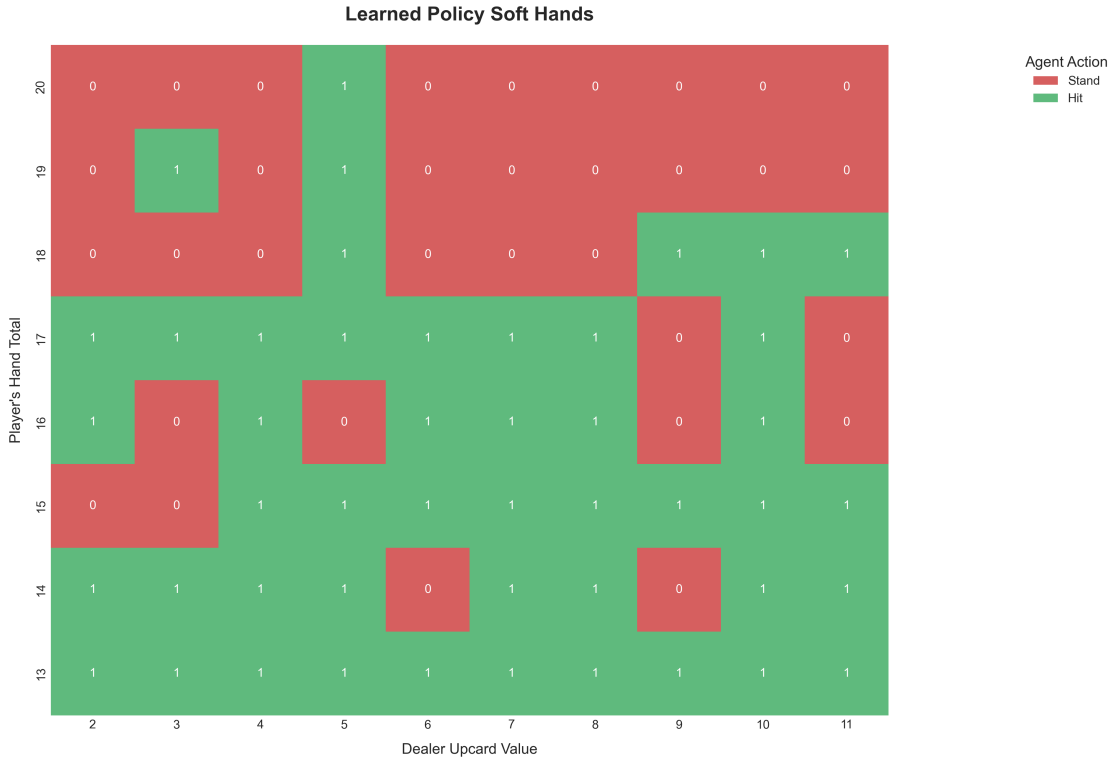

Figure 7: Learned Policy for Soft Hands: The agent adapts its strategy based on the presence of a usable ace, again reflecting optimal blackjack play.

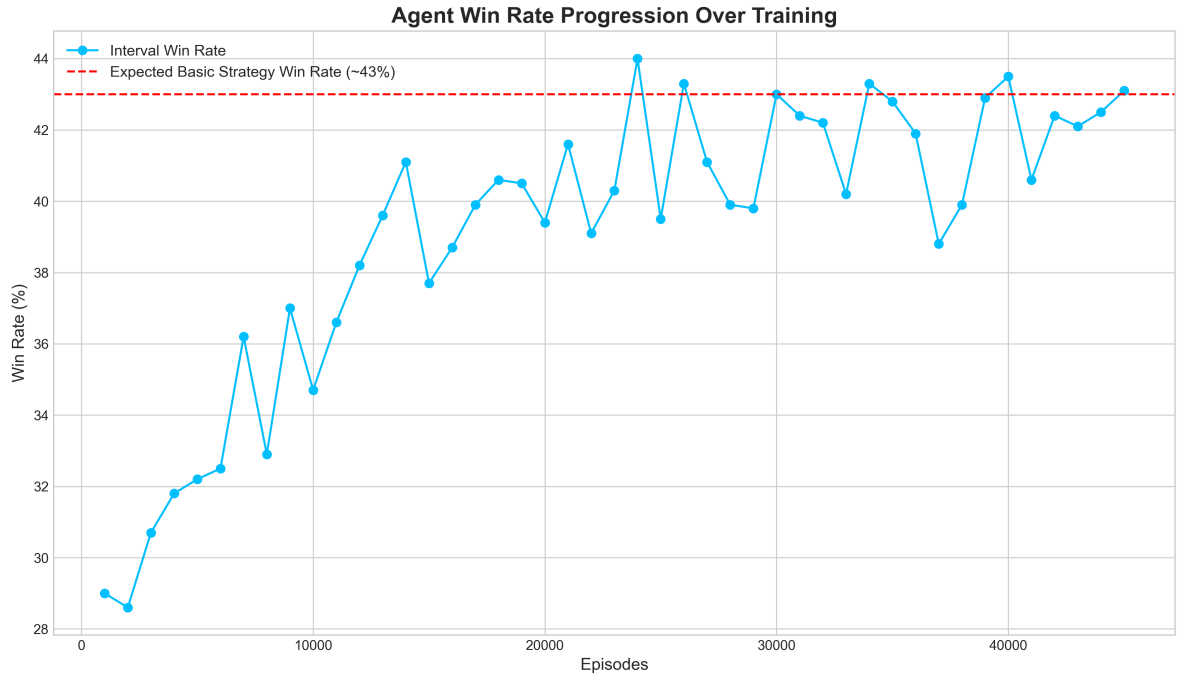

Figure 8: Agent Win Rate Progression Over Training: The agent converges to a win rate close to the expected basic strategy win rate (approximately 43%).

considerably higher than *Stand* at -0.4108, indicating a clear preference consistent with basic strategy. Similarly, for the state (17, 7, 0), the agent prefers *Stand* with a Q-value of -0.2059 over *Hit* at -0.4424.

Table 1: Sample Q-table Values for Selected States

State (Player Sum, Dealer Upcard, Usable Ace)	Action	Q-value
(17, 7, 0)	Stand	-0.2059
	Hit	-0.4424
(12, 4, 0)	Stand	-0.2089
	Hit	-0.3837
(18, 10, 0)	Stand	-0.3213
	Hit	-0.5813
(11, 7, 0)	Stand	-0.4108
	Hit	0.2488

In summary, the agent's learning closely matches the expected optimal blackjack play, validating the effectiveness of Q-learning for this task and its potential applicability in casino-related AI systems.

Conclusions and Real World

0.9. Application to Casino Industry

The reinforcement learning-based blackjack agent offers several practical benefits for the casino industry:

- **Training Dealer Bots:** The agent can serve as an autonomous dealer simulator, enabling casinos to train and test dealer software or staff with realistic and adaptive gameplay scenarios.
- **Player Strategy Tutorials:** By modelling optimal play, the agent can power interactive tutorials that teach players the best strategies, improving player engagement and satisfaction.
- **Player Behaviour Analysis:** The agent's decision-making patterns can be used as benchmarks to analyse and classify player behaviour, helping identify deviations from optimal play, which may indicate novice players or potential problem gamblers or even bad business.
- **Dynamic Game Balance:** Integrating the RL agent into live games could enable real-time adjustments of game difficulty or dealer strategies, ensuring fair play while maintaining house profitability and player enjoyment.

Overall, this RL solution demonstrates how AI can enhance operational efficiency, player education, and responsible gaming initiatives within casino environments.

This project is available in *my GitHub repository*