Mastering Blackjack with Reinforcement Learning

Sai Surya Mada
(Student, MAI program)
Technische Hochschule Wurzburg-Schweinfurt
Wurzburg, Germany
saisurya.mada@study.thws.de

***Abstract*—The goal of the traditional casino game blackjack is to get your hand total closer to 21 than the dealer's without going over 21. In order to create an agent that can learn the best playing policies, this work explores the use of model-free reinforcement learning algorithms, including Q-Learning and on-policy SARSA. With a win rate of 43.10%, a loss rate of 48.33%, and a push rate of 8.57%, the results show that the SARSA algorithm performs the best, effectively mastering the popular "Complete Point Count System". The study also assesses the agent's ability to learn more sophisticated, memory-based card counting policies and adapt to diverse rule changes, offering a quantitative measure of its performance. This work validates SARSA as a highly effective algorithm for discovering robust and profitable strategies in the stochastic environment of Blackjack.**

***Index Terms*—Reinforcement Learning (RL), Blackjack, Q-Learning, SARSA, Basic Strategy, Optimal Policy, Model-Free Control, Complete Point Count System.**

I. Introduction

Blackjack is a game played between player and a dealer where the task is to get a hand total of 21 or closer to 21 without exceeding the value. In this game standard cards are used, numbered cards are kept as it is, face cards (Jack, King, Queen) are worth 10 and an Ace can be valued as 1 or 11. The game starts with the player and dealer each receiving two cards; one of the dealer's cards remains face-down. The player must then decide whether to "hit" (take another card) or "stand" (end their turn). If a player's total exceeds 21, they "bust" and lose the round. Once the player stands, the dealer reveals their face-down card and must play according to fixed house rules, which typically require them to hit on any total of 16 or less and stand on 17 or moreIn this game, both the player and the dealer are dealt two cards to begin; one of the dealer's cards is initially hidden. The player then has to choose between taking an additional card ("hit") or concluding their turn ("stand"). Should a player's card total exceed 21, they "bust" and are out of the round. After the player stands, the dealer turns over their hidden card and must follow predetermined house rules, generally meaning they must hit on a total of 16 or less and stand on 17 or more. [2]

The process of decision-making under uncertainty makes Blackjack an ideal problem for Reinforcement Learning (RL), a field of artificial intelligence where an agent learns to optimize its actions through trial and error [1].1 An agent interacts with an environment in Reinforcement Learning framework by observing its current state (e.g., the player cards and dealer's face up card), choosing an action (hit or stand), and receiving a reward based on the final outcome (e.g., +ve value for winning the game and -ve value for losing the game). The agent goal is to learn an optimal policy. Policy, basically a strategy that maps states to actions to maximize its cumulative reward over time.Because the game can be formally as a Markov Decision Process (MDP). SARSA and Q-learning, both model-free reinforcement learning (RL) algorithms, offer a robust method for directly deriving optimal policies from simulated gameplay.

The process of decision-making under uncertainty makes Blackjack an ideal problem for Reinforcement Learning (RL), a field of artificial intelligence where an agent learns to optimize its actions through trial and error [1]. An agent interacts with an environment in Reinforcement Learning framework by observing its current state (e.g., the player cards and dealer's face up card), choosing an action (hit or stand), and receiving a reward based on the final outcome (e.g., +ve value for winning the game and -ve value for losing the game). The agent goal is to learn an optimal policy. Policy, basically a strategy that maps states to actions to maximize its cumulative reward over time.Because the game can be formally as a Markov Decision Process (MDP). SARSA and Q-learning, both model-free reinforcement learning (RL) algorithms, offer a robust method for directly deriving optimal policies from simulated gameplay.

This research objective is to determine if a reinforcement learning agent can independently develop optimal Blackjack strategies and to measure the expected profits from these learned approaches. To address this, the study evaluates and applies model-free reinforcement learning techniques, specifically on-policy SARSA and Q-Learning methods. The primary objective is to train an agent that, through Gameplay simulations, learns an optimal policy for choosing actions to maximize its rewards. Finally, these techniques are utilized to devise effective game plans and to measure the agent's ability to maximize its long-term expected profit within the game's unpredictable environment.

II. Methodology

The technical framework detailed in this section is used to train the reinforcement learning agent. The problem is shaped as a Markov Decision Process (MDP), and the agent learns an optimal policy through interaction with a simulated Blackjack environment.

A. Basic Strategy

The primary objective is to train the agent to learn the "Basic Strategy," that represents the optimal set of decisions for a player who has no knowledge of previously played cards for a player who is unaware of cards played previously. This memoryless policy is an important foundation for validating the impact of our learning algorithm. The agent functions in a simulated multi-deck environment that replicates casino gameplay, including deck penetration and reshuffling rules. The "Stand on Soft 17" rule is followed by the dealer. To empower the agent to learn this policy, the environment is formulated as a finite MDP. The state (S) is defined by the useful information available to the player at any decision point and is represented by a tuple containing the player's current sum, the value of the dealer's visible card, and a boolean flag indicating if the hand is "soft" due to a usable ace. [1] The agent's set of possible actions (A) includes hit, stand, and double down. Based on the simulation's rules,double-down action is limited and is only available as the first move of a hand, on the player's initial two cards. The agent learns from a terminal reward signal (R) issued at the conclusion of each hand. A standard win receives a reward of +1, a loss -1, and a push 0. For specific outcomes, this structure is modified: a natural blackjack is rewarded with +1.5, and a successful double down action yields a reward of +2, while an unsuccessful one results in a penalty of -2.

B. Complete Point Count System

To help the agent learn a policy based on memory, we move beyond the limitations of the Basic Strategy by integrating a card counting system. By tracking the arrangement of the remaining deck, the agent can make smart decisions. The method implemented is the popular Hi-Lo system, a strategy whose principles were innovated in the foundational work of Edward O. Thorp. This system is effective because it tracks the ratio of high-value cards (which favor the player) to low-value cards (which favor the dealer) [2]. The system functions by assigning a point value to each card as it becomes visible during play. Cards numbered from 2 to 6 are assigned with a value of $+1$. Cards from 7 to 9 are neutral and assigned with a value of 0. High-value cards, including 10s, face cards, and Aces, are assigned with a value of -1. As cards are dealt, a running count is maintained by the agent, where point values are added. To make this count useful in a multi-deck game, it is normalized to produce a "true count." In our simulation, the true count is determined by dividing the running count by the number of decks remaining in the shoe. This value provides a standardized measure of player advantage. To enable the agent to utilize this information, the state space (S) is extended to include this new variable. The resulting state representation is a tuple: $(\mathit{player\_sum}, \mathit{dealer\_up\_card}, \mathit{usable\_ace}, \mathit{true\_count})$. This expanded state enables the agent to learn a more advanced policy, where the optimal action for a given hand can change dynamically based on the composition of the remaining cards.

C. Q-Learning

Q-learning is a model-free, an off-policy [3] TD control algorithm learns the optimal action-value function to identify the optimal policy, $Q^*(s,a)$. The main feature of Q-learning is its "off-policy" nature; it means the algorithm learns the optimal policy value regardless of the agent's actions. The learned policy (the target policy) is to always select the action with the highest Q-value, while the policy used to interact with the environment and generate behavior (the behavior policy) can be more exploratory, such as an ε-greedy policy [3]. The agent can explore many actions and simultaneously learn an optimal, deterministic strategy due to this separation. The learning process is mainly dependent on Q-learning's update rule, which is applied after each state transition. The equation for this update is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{1}$$

In this formula, the state-action pair's current estimate, $Q(S_t, A_t)$, is updated using the TD error. The term $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ represents the TD target. The main element here is $\max_a Q(S_{t+1}, a)$, which is the maximum Q-value for the next state, S_{t+1}, across all possible next actions. This term denotes the estimated value of optimal behaviour from the next state. Continuously using this maximum value in the update, Q-learning directly approximates the Q-values for the optimal policy, regardless of which action the behavior policy actually selected in the next step. The learning rate, α, controls the scope of the update, while the discount factor, γ, determines the present value of future rewards.

D. SARSA

SARSA, which stands for State-Action-Reward-State-Action, is an on-policy temporal difference (TD) control algorithm used to find an optimal policy [1]. Being "on-policy" means to enable the agent to make decisions based on the learned value of its current policy, including any exploratory actions. This approach contrasts with off-policy methods, which grasp the optimal policy's value regardless of the agent's immediate actions. Because SARSA defines its unique exploration method (e.g., an ε-greedy policy), it tends to learn more traditional or "safer" policies, avoiding risky states where an exploratory move could be costly. In high-risk environments, this approach can lead to more stable learning and improved performance during training.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right] \tag{2}$$

The TD error is used to update the current estimate of the state-action pair, $Q(S_t, A_t)$. This adjustment is made by a fraction (α), known as the learning rate. The TD error is the difference between the current estimate and the TD target, $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$. The essential component that defines

SARSA as on-policy is the term $Q(S_{t+1}, A_{t+1})$. Here, A_{t+1} is the actual action that the agent selects in the next state, S_{t+1}, according to its current policy. The algorithm learns a realistic value of its own behavior by updating its value estimates based on the actions it actually takes, rather than relying on the theoretical value of a purely optimal (greedy) policy.

III. Implementation

A. Basic Strategy

The initial phase of this research focuses on training reinforcement learning agents to master the memoryless "Basic Strategy." This is attained through a custom Python-based Blackjack simulation that serves as the learning environment. The implementation includes two different learning agents, one based on the off-policy TD (Q-learning) algorithm and the other on the on-policy SARSA algorithm

Environment and Rule Variations

Two rule variations are implemented in the Blackjack environment for the benefit of the player which is Reinforcement Learning agent in our case:

Stand on Soft 17: The dealer's behavior is fixed by a non-learning policy within the `_play_dealer_hand` method. The policy requires the dealer to stand on any hand total of 17 or greater, including "soft" hands (hands containing an Ace counted as 11). In casino Blackjack, this is a frequently encountered and significant rule variation.

Undo Hit: This rule can be used only once per game. After taking a "hit" action, the agent is given a one-time option to "undo" that action, returning the game to its state just before the hit. This is managed through boolean flags (`undo_available`, `undo_used`) and a state-saving functional method (`pre_hit_state`) within the environment.

The agent's learning process mostly relies on its state representation. In this implementation, the state is a comprehensive tuple where all essential information is stored for decision-making under these specific rules: `(player_sum, dealer_up_card, is_soft, has_doubled_down, undo_available, undo_used)`. The integration of the final three boolean flags is essential for the agent to learn policies that effectively use the double down and custom undo actions.

Q-Learning Agent Implementation

The `TDAgent` class implements an off-policy Q-learning algorithm. The agent's goal is to learn the optimal action-value function, $Q^*(s, a)$, this denotes the highest anticipated future reward achievable by performing action a in state s.

The agent's knowledge is stored in a dictionary named `self.q_table`. The agent's memory or strategy guide is stored within this Q-table, mapping learned Q-values with their corresponding state-action pairs. In the context of our Blackjack agent, a Q-value, such as `self.q_table[((15, 7, False,...), 'h')]`, represents the agent's current estimate of the long-term reward for choosing to 'hit' when its hand is a hard 15 and the dealer shows a 7. Initially, this table is empty, and the `_get_q_value` method returns 0 for any unvisited state-action pair. With thousands of games simulated, the agent undergoes continuous updates, these Q-values based on the final rewards, learning to gradually assign higher Q-values to actions that lead to wins, while those leading to losses should have lower ones [3].

Action is selected by an ε-greedy policy, which can be tuned by the `self.epsilon` parameter. This parameter is important for addressing the balance between exploration (trying new actions) and exploitation (using known good actions). In the `select_action` method, the line `if np.random.random() < self.epsilon:` determines the agent's choice. If the random number is less than epsilon, the agent explores by choosing a random valid action. For this Blackjack agent, this means it might try hitting on a 19 or standing on a 12. Although these exploratory moves might result in immediate challenges, they are important for the agent to develop a comprehensive Q-table and understand the outcomes. If the random number is greater than epsilon, the agent exploits its knowledge by selecting the action with the highest Q-value for the current state. Typically, epsilon decays as training advances, leading the agent to reduce exploration and increasingly depend on its acquired, profitable strategy.

SARSA Agent Implementation

The `SARSA_Agent` class implements an on-policy temporal difference control algorithm. Unlike Q-learning, SARSA (State-Action-Reward-State-Action) learns the value of the policy it is currently following, including its exploratory moves. Because the agent learns from its own real actions—including the occasional random, exploratory ones—it typically adopts a more careful approach. In the context of Blackjack, this means the SARSA agent might learn to avoid specific actions in certain states where an exploratory move could be disastrous (e.g., hitting on a high hand value), as the negative rewards from those mistakes directly influence its Q-value estimates [1].

The algorithm's name reflects the sequence of events required for a single update. In the implementation, the agent starts in a state S_t, uses `select_action` to choose an action A_t, and observes the resulting reward R_{t+1} and next state S_{t+1}. Before it can update the value of the original action, it must use `select_action` again to determine the actual next action, A_{t+1}, it will take from the new state S_{t+1}.

B. Complete Point Count System

To build an agent capable of learning a memory-based strategy, the implementation was expanded to include the Hi-Lo card counting system. This phase enables the Q-learning and SARSA agents to utilize information about previously seen cards to make decisions that offer greater strategic advantages.

State Space Expansion and Discretization: The primary change was to expand the state representation to include the true count. This was achieved within the `BlackjackEnvironment` by adding a `get_true_count` method, which calculates the running count based on all visible cards and normalizes it by the number of decks remaining. Because a continuous true count value would create an excessively large state space for tabular learning methods, this value is discretized into a finite number of bins within the agent's `_get_binned_state` or `_discretize_state` methods. This process maps the continuous count to a specific category (e.g., "low," "neutral," "high"). To maintain a manageable state space for the Q-table, this binning is important. The final state representation used by the agents is therefore an expanded tuple that includes this binned count: `(player_total, dealer_up, is_soft, true_count_bin, undo_available,...)`.

Q-Learning Agent Implementation: The Q-Learning Agent seamlessly incorporates the Complete Point Count System by utilizing the expanded, discretized state as the identifier for its Q-table. The method `_get_binned_state` analyzes the raw state of the environment alongside the computed true count to create this identifier. As a result, the agent learns and stores unique Q-values for identical player-dealer hand combinations, adapting to varying true count conditions. The update method employs the conventional off-policy Q-learning strategy; however, since the state now reflects the card count, the agent can evaluate the value of an action based on the remaining deck's composition.

SARSA Implementation: The SarsaAgent has been modified to utilize the discretized state that includes the true count, enabling it to develop an on-policy strategy that is aware of counts. A key parameter in this implementation is the management of the exploration rate, which is regulated by the epsilon value. The code for the agent integrates `epsilon_decay` and `epsilon_min` parameters, which are utilized in the `decay_epsilon` method to slowly lower the likelihood of selecting a random action as training continues. This gradual decrease is essential for learning in a larger state space; it guarantees that the agent researches a diverse array of state-action pairs (including various true counts) during the early stages of training before settling into a more refined, exploitative policy based on its own experiences.

IV. Experiments and Evaluation

A. Basic Strategy

To evaluate the effectiveness of the implemented reinforcement learning algorithms, a series of experiments were conducted. Both the *off-policy* Q-learning and *on-policy* SARSA agents were trained for 500,000 episodes using basic Strategy which includes the "Stand on Soft 17" and "Undo Hit" rule variations to allow their policies to converge. Following the training phase, to evaluate each agent's learned policy, 400,000 hands of Blackjack were simulated to gather reliable performance metrics.

The training process for both agents showed successful policy convergence. As shown in the learning curve for the SARSA agent (Figure 1), the moving average of the reward per episode shows a consistent increase from approximately -0.45, eventually settling around -0.1, which indicates that an effective policy has been learned. The Q-learning agent displayed a comparable positive learning trajectory. The epsilon decay graph (Figure 2) illustrates the gradual shift from a strategy that emphasizes exploration ($\varepsilon \approx 1.0$) to one that focuses more on exploitation ($\varepsilon \approx 0.08$) throughout the training process. This procedure is important for enabling the agents to initially identify and subsequently adjust their strategies.

A qualitative assessment of the policies learned from the agents' final `Q-tables` indicates that both algorithms formulated a coherent strategy. The learned Q-values for the "Stand" action reveal that both agents attributed high positive values to standing on strong hands (such as 19-21) and significant negative values to standing on weak hands (like 12-15), confirming that the fundamental concepts of Blackjack strategy were effectively learned.

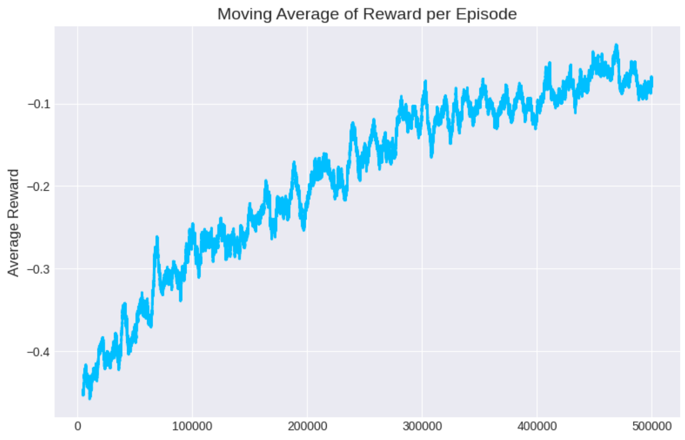

Fig. 1. The learning curve for the SARSA agent. The plot shows the moving average of rewards per episode over 500,000 training episodes, indicating policy convergence.

The quantitative evaluation, benchmarked over 400,000 hands, revealed a nuanced performance difference between the two algorithms, summarized in the table below.

TABLE I
PERFORMANCE METRICS OF Q-LEARNING AND SARSA AGENTS

Metric	**Q-Learning Agent**	**SARSA Agent**
Return to Player (RTP)	97.96%	97.66%
Win	42.81%	42.98%
Loss	48.07	48.15
Push	9.13	8.87
Undo Rate	29.86%	21.52%

While the *off-policy* Q-learning agent achieved a slightly better financial outcome in terms of RTP, the *on-policy*

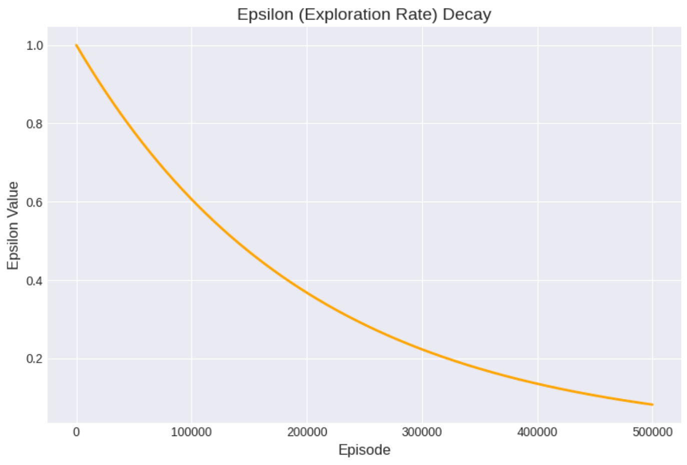

Fig. 2. Decay of the epsilon. The exploration rate is gradually reduced over 500,000 episodes, shifting the agent's strategy from exploration to exploitation

SARSA agent developed a more effective and robust playing strategy, clearly demonstrated by its higher win rate (42.98% vs. 42.81%). The Main difference in their learned behaviors appears in their strategic approach to the "Undo Hit" rule.

The SARSA agent utilized the "Undo Hit" option in only 21.52% of hands, less often than the Q-learning agent's 29.86%. This difference in behavior arises from their fundamental learning processes. SARSA, being *on-policy*, learns the value of its policy that includes self-exploration and is directly penalized for risky moves that lead to unfavorable states requiring an "undo." This encourages it to learn a more cautious or "safer" policy that avoids such situations. In contrast, the *off-policy* Q-learning agent acquires knowledge of the optimal policy's value, irrespective of the exploratory actions it performs. This allows it to learn a more aggressive strategy that relies more extensively on the "undo" feature as a safeguard. Although the Q-learning agent's aggressive strategy was slightly more profitable, the SARSA agent's higher win rate and more cautious policy demonstrate a superior and a more stable understanding of the game's dynamics.

B. Complete Point Count System

In the second phase of experiments, the agents were assigned the task of developing a memory-based policy by utilizing the Hi-Lo card counting method. The state space was expanded to include a discretized true count, and the agents were trained for a duration of 2,000,000 episodes to learn the more complex strategy. The ultimate policies were assessed over an extensive number of hands (500,000 for Q-learning and 1,000,000 for SARSA).

The training process for both agents showed a clear, slow pace, convergence toward a stable policy. The learning curve for the SARSA agent (Figure 3) shows a steady increase in the moving average of rewards per episode, indicating a successful learning process in associating states, including the card count, with long-term rewards. The related epsilon decay graph (Figure 4) illustrates the slow decrease in exploration, enabling the agent to take advantage of its progressively refined `Q-table` throughout the prolonged training duration.

The impact of the "Undo Hit" rule variation showed significant differences in the learned strategies. The Q-learning agent used the 'undo' action 138,497 times over 500,000 hands which is 27.7% of hands, while the SARSA agent used it 194,333 times over 1,000,000 hands which is 19.4% of hands. Considering the various evaluation lengths, the frequency of SARSA's usage was significantly lower. This indicates that the *on-policy* SARSA agent developed a more conservative policy, since its Q-values are modified according to the actual actions performed, which include expensive exploratory moves that might necessitate a reversal. In contrast, the *off-policy* Q-learning agent learns the optimal path without directly factoring in the potential negative consequences of its own exploration policy, leading to a more strong approach that relies more on corrective actions.

The quantitative evaluation confirms a sharp contrast in performance, with the *on-policy* SARSA agent significantly outperforming the *off-policy* Q-learning agent.

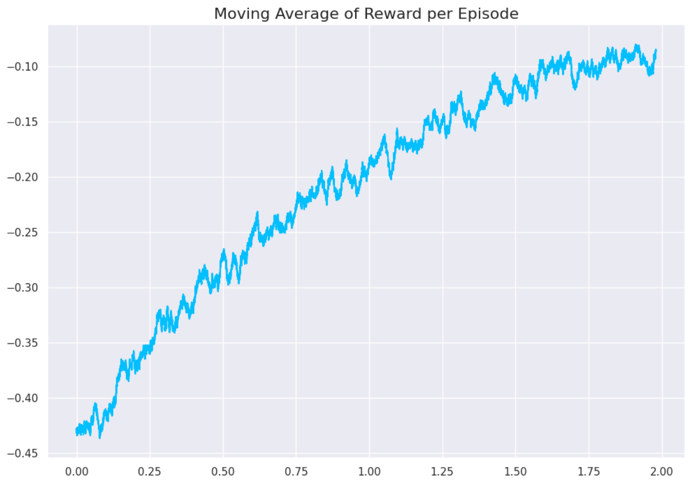

Fig. 3. Performance of the on-policy SARSA agent with a memory-based card counting strategy

TABLE II
PERFORMANCE METRICS WITH CARD COUNTING

Metric	**Q-Learning Agent**	**SARSA Agent**
Win Rate	41.37%	43.10%
Loss Rate	49.15%	48.33%
Push Rate	9.47%	8.57%
Player Advantage	-6.87%	-0.019%

The SARSA agent exhibited a thorough understanding of the card counting strategy, achieving a win rate of 43.10% and a player advantage of -0.019%. This barely profitable performance is a remarkable result, indicating that the agent learned an expert-level policy that effectively nullifies the house edge.

The Q-learning agent found it difficult to integrate the card count into its strategy. It achieved a win rate of only

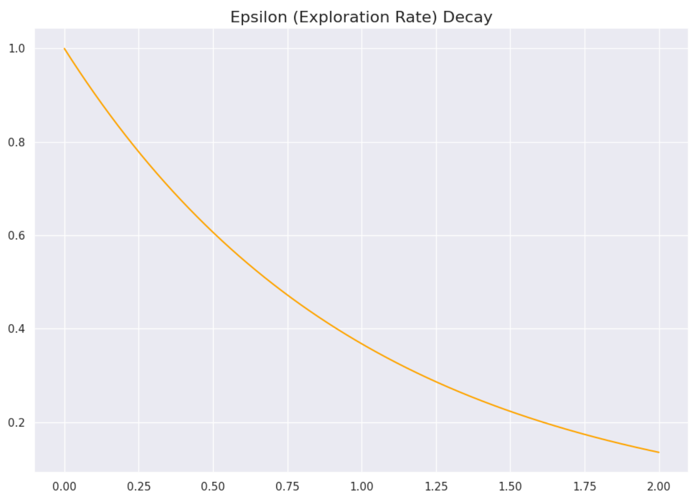

Fig. 4. Epsilon Decay of the on-policy SARSA agent with a memory-based card counting strategy

41.37% and a substantial player disadvantage of -6.87%. This outcome suggests that the *off-policy* nature of Q-learning, that learns the optimal action irrespective of the policy followed, was less effective in navigating the Challenges of a memory-based strategy. The *on-policy* approach of SARSA, which learns from the consequences of its own actions, proved to be far more robust for this task, enabling it to establish a highly effective and improved card counting strategy.

V. Conclusion

This paper demonstrated the application of model-free reinforcement learning to develop optimal Blackjack approaches, providing a clear quantitative comparison between *on-policy* SARSA and *off-policy* Q-learning algorithms. Both agents successfully developed effective policies for the memoryless Basic Strategy, significant findings were derived from the memory-based Complete Point Count System. The agent trained with SARSA on the card counting strategy was the best-performing model, achieving a good player advantage of -0.019% and the highest win rate of 43.10%. This performance eliminates the house edge, highlighting the major drawbacks experienced by the Basic Strategy players and the Q-learning card counting agent, which failed to learn a profitable policy.

Results show that for complex, memory-dependent tasks like card counting, the *on-policy* SARSA algorithm is a more strong and dependable choice than its *off-policy* alternative. This study serves as a practical delivery of the capability of reinforcement learning to excel at complex tasks, advanced techniques in uncertain settings. Considering the future, this research opens up several potential areas. One extension for this implementation would be to explore deep reinforcement learning architectures, such as Deep Q-Networks, to manage the state space without discretization. Additionally, Including a dynamic betting strategy—where the agent learns to vary its wager size based on the true count—could be the key to transforming the agent from a breakeven player into one with a consistent, positive advantage.

References

[1] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[2] E.O. Thorp. *Beat the Dealer: A Winning Strategy for the Game of Twenty-One*. A Vintage book. Knopf Doubleday Publishing Group, 1966.

[3] Christopher Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992.

Acknowledgement

I would like to express my sincere thanks to Prof. Dr. Frank Deinzer for his valuable lectures, which provided a strong foundation for this work. I also wish to acknowledge the use of QuillBot for paraphrasing assistance and Grammarly for grammar checking, which were valuable in refining this paper.