Analysis Report: Optimization of a Q-Learning Agent for Blackjack

Clara Glotin

February 18, 2026

Abstract

This project implements and analyzes a Reinforcement Learning agent using the Q-Learning algorithm to solve the game of Blackjack within the Gymnasium environment. The first objective is to train an autonomous agent to learn the optimal decision-making policy solely through interaction with the environment, without prior knowledge of the game rules. The second objective is to analyze its performance regarding the choice of parameters. This report details the implementation of the tabular Q-Learning method, the tuning of hyperparameters such as the exploration rate (ϵ) and learning rate (α), and the analysis of the agent's convergence. Our results demonstrate that the agent successfully converges towards the theoretical optimal policy described in the literature, approaching the "Basic Strategy" and validating the Q-Learning approach for discrete stochastic environments.

Contents

1 Introduction to Blackjack

This report presents a performance analysis of a Reinforcement Learning agent applied to Blackjack. Blackjack is a classic casino card game where the player's objective is to beat the dealer by obtaining a hand total closer to 21 than the dealer's, without exceeding it. Face cards (Jack, Queen, King) count as 10, while Aces count as either 1 or 11, depending on which value benefits the hand most (a hand with an Ace counted as 11 is called a soft hand, otherwise it is a hard hand). Based on their own hand and the dealer's single face-up card (the upcard), the player must strategically decide whether to hit (draw a card) or stick (keep the current hand).

However, it is important to note that Blackjack is a mathematically solved game. A deterministic optimal policy, known as "Basic Strategy", was established by Edward O. Thorp [1], proving that the house edge could be minimized through rigorous probability. Consequently, the application of Q-Learning in this study is not driven by the necessity to solve an unknown problem, but rather serves as a validation benchmark. Since the theoretical upper bound of performance is known, we can evaluate the algorithm's ability to converge towards this optimal decision-making policy, a task frequently used as a reference in reinforcement learning literature. We do not use the precise "Basic Strategy" from Thorp as our reference but rather the work of Sutton and Barto on Q-Learning [2] because it corresponds to the exact environment in which we are training our agent.

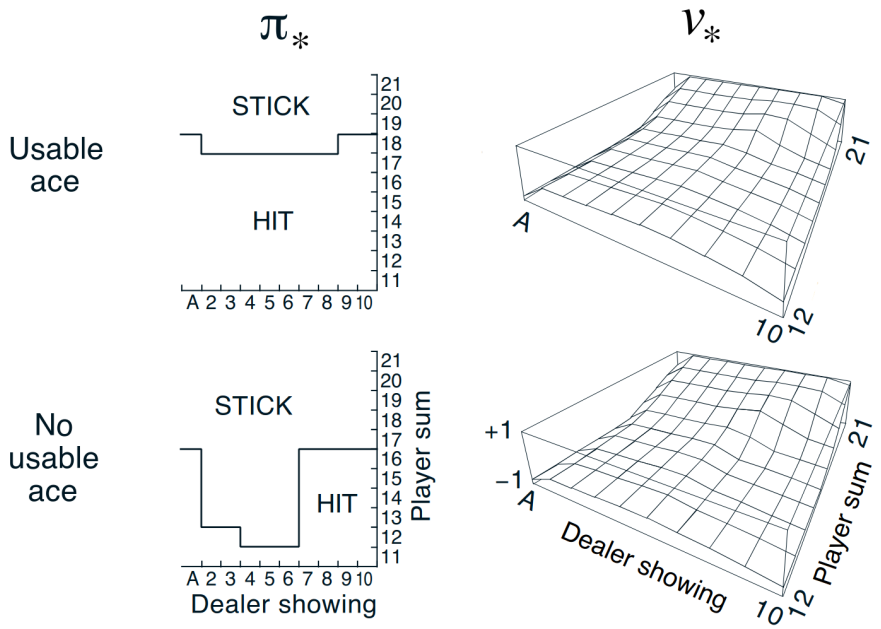

Figure 1: Optimal Strategy Tables learned by the Q-Learning agent from Sutton and Barto, using a Monte Carlo Algorithm [2]

The tables presented in Figure 2 illustrate our ground truth strategy. This specific matrix format represents our agent's internal knowledge base. Initially, these matrices are initialized to zero, as the optimal strategy is unknown to the agent at the start of the training. Following a training cycle, the agent populates them according to its optimization algorithm. Throughout this study, we use these target tables to visualize and quantify the accuracy of our Q-Learning agent's learned policy.

2 Methodology

2.1 The Q-Learning Algorithm

2.1.1 Principles

Alan Turing famously stated in 1947, "What we want is a machine that can learn from experience." Q-Learning has emerged as one of the fundamental methods developed to achieve this goal. As a model-free reinforcement learning algorithm, it is designed to learn the value of an action a in a particular state s [3]. The core of this process is the Bellman Optimality Equation [2], which iteratively adjusts the Q-values:

$$Q^{new}(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{1}$$

Where:

- $Q(s,a)$ is the current estimate of the expected future reward.
- α (Learning Rate) determines to what extent newly acquired information overrides old information.
- R is the immediate reward received after taking action a in state s.
- γ (Discount Factor) weights the importance of future rewards.
- $\max_{a'} Q(s',a')$ represents the maximum predicted reward for the next state s'.
- The term in brackets represents the *Temporal Difference Error*.

This iterative sampling relies on the Law of Large Numbers [4]: as the agent interacts with the environment over many episodes, the empirical average of the observed returns for each state-action pair converges toward the true expected value.

2.1.2 Exploration Strategy

To ensure the agent gathers enough data for convergence, we employ an exploration-exploitation trade-off via an ϵ-greedy strategy. For this study, we opted for a linear decay of exploration (ϵ) to transition from discovery to exploitation:

$$\epsilon_t = \max \left(\epsilon_{min}, \epsilon_{start} - t \cdot \frac{\epsilon_{start} - \epsilon_{min}}{n_{episodes}/2} \right) \tag{2}$$

Where:

- ϵ_{start} and ϵ_{min} are the initial and final exploration rates, respectively.
- t represents the current episode index.
- $n_{episodes}$ is the total number of training episodes.
- ϵ_t is the resulting probability of choosing a random action at episode t.

This formula implies that the agent reaches its minimum exploration rate halfway through the training process ($t = n_{episodes}/2$), focusing the remaining half on exploiting the learned Q-values.

2.2 The Gymnasium Framework

2.2.1 Environment Specification

For computational experimentation, we utilize the `Blackjack-v1` environment from the Gymnasium library [5]. The environment models the game as a Markov Decision Process (MDP) with:

- **Observation Space:** A 3-tuple consisting of the player's current sum, the dealer's visible card, and the presence of a usable ace.
- **Action Space:** A discrete space with two possible actions: *Stick* (0) or *Hit* (1).
- **Reward Function:** The agent receives $+1$ for a win, -1 for a loss, and 0 for a draw (push).

2.2.2 Initial Setup and Non-Optimized Parameters

Before optimization, the agent is initialized with baseline parameters to establish a performance reference. The default experimental setup is defined as follows:

- Learning Rate (α): 0.01
- Number of training episodes: 10,000
- Initial Epsilon (ϵ_{start}): 1.0
- Final Epsilon (ϵ_{min}): 0.1
- Discount Factor (γ): 0.95

2.3 Hyperparameter Optimization and Evaluation

To ensure a robust analysis of the agent's performance and to mitigate the high stochasticity of Blackjack, we implemented four specific metrics.

2.3.1 Mean Reward During Training

This is the rolling mean (calculated over 1,000 consecutive episodes) of the rewards obtained. It monitors the learning dynamics and identifies when the agent reaches a stable state (convergence).

2.3.2 Mean Win Rate During Training

Also calculated over a 1,000-episode sliding window, this metric compares the agent's success against the theoretical optimal win rate, evaluating the immediate effectiveness of the learned strategy during the process.

2.3.3 Mean Strategy Win Rate (Pure Policy)

To isolate the agent's actual knowledge from the noise of exploration (ϵ), we impose a pause in training every 200 steps. We then evaluate the current strategy over 10,000 trials (see $n\_eval\_episodes = 10\,000$ in the code) with $\epsilon = 0$. This "pure" performance metric is crucial for observing the true quality of the Q-table.

2.3.4 Statistical Averaging (20 Experiments)

Due to the high variability of card draws, every simulation is repeated 20 times. The final curves presented in the results are the average of these 20 independent runs, ensuring statistical significance through the Law of Large Numbers.

All these metrics and the resulting performance data are visualized through comparative curves and discussed in detail in the following section.

3 Results and Discussion

3.1 Effectiveness of Q-Learning in Blackjack

The results presented in this section demonstrate that Q-Learning is effective for identifying the optimal strategy in Blackjack. As shown in the "Mean Strategy Win Rate" (Figure 3, right graph), the agent systematically and rapidly reaches the theoretical optimal win rate of approximately 42% within about 2,000 episodes, provided the learning rate is non-zero. This aligns with the theoretical maximum win rate derived by Edward O. Thorp [1] ($\approx 42.5\%$).

3.2 Strategy Tables obtained

After training—specifically once the Learning Rate (LR) has converged, which occurs after approximately ten minutes—our agent outputs its computed optimal strategy table. As demonstrated in the graphs in the following sections, this strategy table exhibits a slight variance. Consequently, within our training duration, each experiment may yield a slightly different version of the strategy table. While these results closely approach the Sutton and Thorp references and produce success rates very near the theoretical optimum of 42.5%, they do not systematically replicate the target strategy table exactly. Our results oscillate around the target model and frequently resemble variants of the figure shown below.

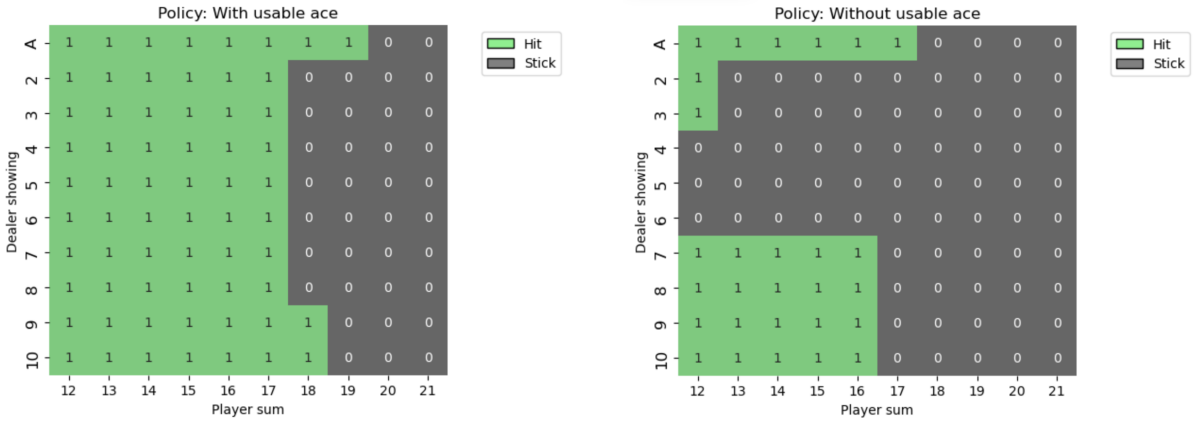

Figure 2: Example of the Strategy Tables obtained by our agent (Left: Soft Hand, Right: Hard Hand)

3.3 The Dominant Effect of ϵ-Decay

To analyze our agent's performance, we first plotted the training curves representing the mean reward as a function of the epoch for different learning rates ranging from 0 to 1. The exploration-exploitation trade-off, controlled by ϵ-decay, significantly influences the training curves. We observe this effect strongly in the left and middle curves of Figure 3.

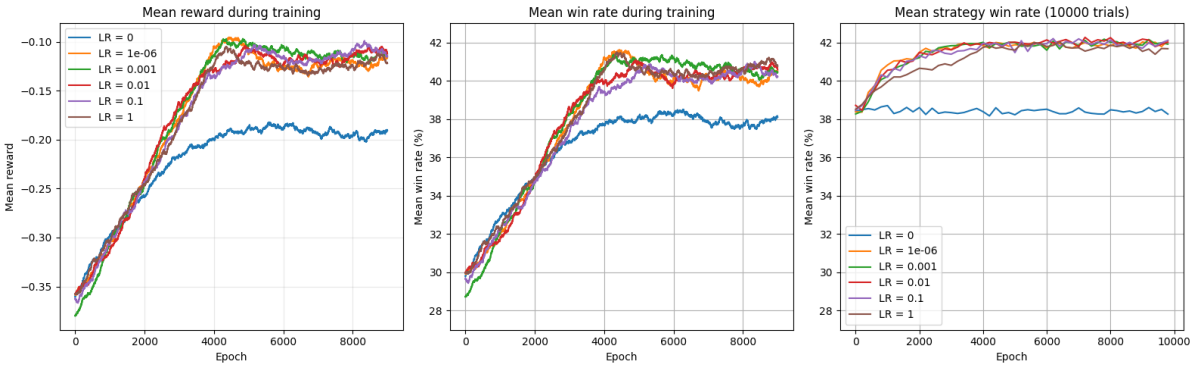

Figure 3: Performance curves during training and evaluation. Runtime: about 2 hours.

- **Initial Phase (Epoch 0):** With $\epsilon = 1$, the performance reflects 100% random behavior, yielding a success rate of $\approx 30\%$ and a mean reward slightly below -0.35.
- **Exploration vs. Strategy:** The apparent improvement in "Mean Reward" and "Mean Win Rate" during training is more heavily influenced by the reduction of random moves (ϵ-decay) than by actual policy updates. This is evidenced by the **Learning Rate = 0** case: the agent still appears to improve, but this is merely the transition from random actions to a constant "Stick" (0) response (as weights initialized at 0 never evolve). This baseline strategy of "never hitting" achieves a mechanical success rate of $\approx 38\%$.
- **Convergence:** By the end of training ($\epsilon = 0.1$), the learned strategy is dominant, and the training performance stabilizes near the theoretical optimum.

3.4 Sensitivity Analysis of the Learning Rate (α)

By isolating the strategy (evaluating with $\epsilon = 0$), we observe the following:

- $\alpha = 0$**:** No strategy improvement occurs, confirming that Q-values do not evolve without updates.
- **Standard Range:** Learning rates between 10^{-6} and 10^{-1} show similar dynamics. The stochastic nature of Blackjack creates higher variability within a single α test than the difference observed between different rates in this range.
- **High α:** A learning rate of 1 is less effective. Excessive weight modifications at each episode cause instability and prevent the agent from stabilizing at the maximum.

3.5 Impact of Exploration Strategy: Decay vs. Greedy

To isolate the importance of exploration, we also compared an agent using Epsilon Decay ($\epsilon_{start} = 1.0$) against a Constant $\epsilon = 0$ (purely greedy) baseline. To ensure statistical reliability, these experiments were performed with a fixed Learning Rate of 0.001 averaged over 6 complete runs.

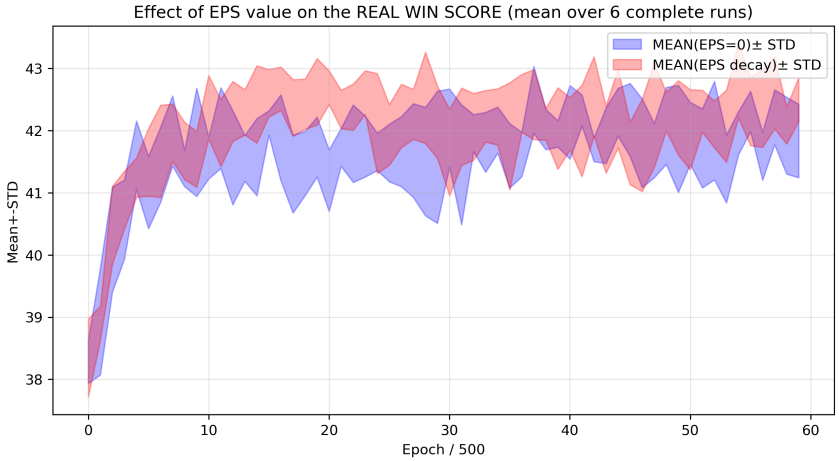

Figure 4: Statistical Analysis: Mean REAL WIN SCORE ± Standard Deviation

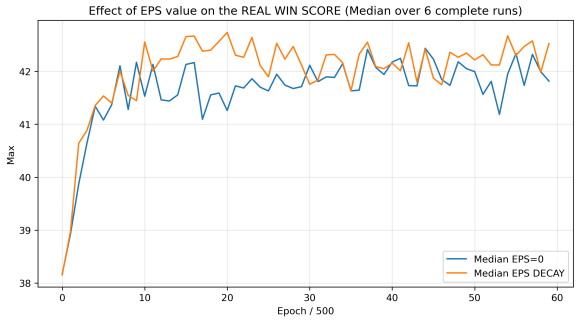

Figure 5: Median performance over 6 runs

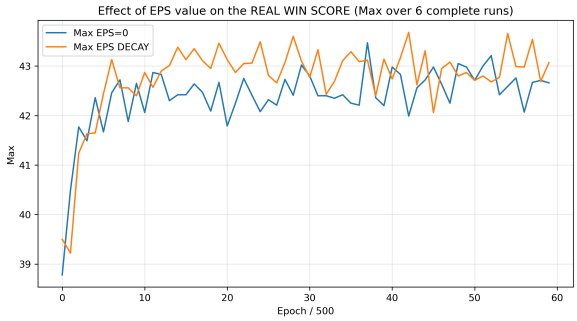

Figure 6: Maximum recorded win score

Comparison of exploration strategies: the Epsilon Decay (orange) consistently discovers optimal policies faster than the purely greedy approach (blue). Runtime: about 30 minutes.

The analysis of these experimental results highlights three key observations:

- **Accelerated Convergence:** As shown in the Median performance (Figure 5) and Maximum recorded win score (Figure 6) curves, the Epsilon Decay strategy outperforms the greedy approach. The high initial exploration allows the agent to discover optimal state-action pairs much earlier in the training process (around $Epoch/500 = 20$).

- **Statistical Robustness:** The Mean ± STD analysis (Figure 4) confirms that this performance gap is not an artifact of variance. The t Student test for the 6 runs around $Epoch/500 = 20$ indeed shows a p-value $p \approx 0.0004$ for averages of $42.7 + -0.25$ vs $41.3 + -0.5$. It demonstrates a highly significative difference between the 2 compared strategies (EPS decay vs EPS=0). The greedy agent's improvements are significantly slower as it lacks the exploration necessary to escape

sub-optimal local policies, whereas the decay strategy maintains a higher and more stable plateau.

- **Final Convergence and Robustness:** Toward the end of training, both curves begin to converge as ϵ approaches 0. However, the "head start" gained during the exploration phase results in a more robust final policy, with peak values reaching the theoretical.

Beyond simple performance metrics, these results highlight that epsilon acts as the primary engine for discovery and robustness in stochastic environments. Without exploration ($\epsilon = 0$), the agent remains a prisoner of its initial luck, often settling for sub-optimal local policies because it never attempts alternative actions that might yield higher long-term returns. In contrast, the initial high-exploration phase of the Epsilon Decay strategy forces the agent to confront a wider variety of Blackjack scenarios, effectively filling the Q-table with diverse experiences. This process not only accelerates convergence toward the 42% threshold but also ensures that the final policy is statistically robust, as evidenced by the stable plateau and lower variance compared to the purely greedy baseline. optimum of $\approx 43\%$ more consistently.

3.6 Future Avenues for Optimization

Our analysis confirms that Q-Learning is sufficient to solve Blackjack. However, the static parameter tuning used here suggests two avenues for faster and more reliable learning:

3.6.1 Dynamic Parameter Control (Meta-Learning)

Instead of a pre-determined linear decay, implementing a feedback control loop—treating the agent as a "controlled plant"—could adjust α and ϵ in real-time. This follows the methodology of Filieri et al. [6], using control-theoretical techniques to maximize stability while minimizing settling time.

3.6.2 Curriculum Learning: Controlled Stochasticity

In early stages, the agent is sensitive to "outliers" (rare hands). Following the *Curriculum Learning* concept by Bengio et al. [7], we propose organizing training by increasing difficulty. By initially presenting "standard" representative hands and gradually reintroducing full randomness, we hypothesize that the agent would learn the "Basic Strategy" more effectively while avoiding early divergence.

4 Conclusion

This study demonstrates that the tabular Q-Learning algorithm is highly effective for solving the Blackjack environment, successfully replicating the theoretical "Basic Strategy". By leveraging the Gymnasium framework, we were able to observe a rapid convergence of the agent toward a win rate of approximately 42%, matching the mathematical optimum for this stochastic game.

A key finding of this analysis is the distinction between apparent performance and actual policy acquisition. Our results highlight that the improvement observed in standard training metrics is largely a mechanical artifact of the ϵ-greedy decay schedule. The

evaluation of a "Pure Strategy" ($\epsilon = 0$) was therefore essential to isolate and confirm the agent's real learning. Furthermore, we showed that while the algorithm is remarkably resilient to varying learning rates (α), the initial exploration phase remains vital: a purely greedy approach fails to explore the state-action space efficiently, leading to slower and less robust convergence.

In conclusion, while Q-Learning proves sufficient for this discrete environment, further performance gains could be achieved by transitioning from static to dynamic parameters. Implementing meta-learning loops to adapt the learning rate in real-time or adopting a curriculum learning approach to reduce early-stage noise are promising avenues to accelerate stability. This benchmark validates the reliability of reinforcement learning for solving decision-making problems in uncertain environments, provided that the evaluation metrics are carefully decoupled from exploration noise.

References

[1] Edward O. Thorp. *Beat the Dealer: A Winning Strategy for the Game of Twenty-One*. Random House, New York, 1966.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2nd edition, 2018.

[3] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[4] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, New York, 1968.

[5] Till Zemann. Training an agent to play blackjack. Gymnasium Documentation, 2023. `https://gymnasium.farama.org/tutorials/training_agents/blackjack_q_learning/`.

[6] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, pages 299–310, 2014.

[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 41–48, 2009.

```
# ADAPTED FROM
# Author: Till Zemann
# License: MIT License

from __future__ import annotations
from collections import defaultdict
import matplotlib.pyplot as plt
import numpy as np
import random as rd
from tqdm import tqdm
import gymnasium as gym

class BlackjackAgent:
    def __init__( self,
                  env,
                  learning_rate: float,
                  initial_epsilon: float,
                  epsilon_decay: float,
                  final_epsilon: float,
                  discount_factor: float = 0.95,):
        self.q_values = defaultdict(lambda: np.zeros(env.action_space.n))
        self.lr = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = initial_epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon
        self.training_error = []

    def get_action(self, env, obs: tuple[int, int, bool], force_greedy: bool = False) -> int:
        # If force_greedy is True (for evaluation), we ignore epsilon
        if not force_greedy and np.random.random() < self.epsilon:
            return env.action_space.sample()
        else:
            return int(np.argmax(self.q_values[obs]))

    def update( self,
                obs: tuple[int, int, bool],
                action: int,
                reward: float,
                terminated: bool,
                next_obs: tuple[int, int, bool],):
        future_q_value = (not terminated) * np.max(self.q_values[next_obs])
        temporal_difference = reward + self.discount_factor * future_q_value - self.q_values[obs][action]
        self.q_values[obs][action] = self.q_values[obs][action] + self.lr * temporal_difference
        self.training_error.append(temporal_difference)

    def decay_epsilon(self):
        self.epsilon = max(self.final_epsilon, self.epsilon - self.epsilon_decay)

# --- Parameters ---
LR = [0, 1e-6, 1e-3, 0.01, 0.1, 1] # learning rates to test
n_episodes = 10_000 # training steps
```

```
eval_interval = 200 # Evaluate every 200 episodes
n_eval_episodes = 10_000 # evaluation steps
NBE = 20 # number of trials for each learning rate
start_epsilon = 1.0
epsilon_decay = start_epsilon / (n_episodes / 2)
final_epsilon = 0.1
rolling_length = 1000 # rolling window size for average

# Storage for plots
all_reward_curves = []
all_win_curves = []
periodic_eval_curves = [] # To store points (step, success_rate)
rd.seed(1)

# FOR EACH LR
for i in range(len(LR)):

    # FOR EACH TRIAL
    for e in range(NBE):
        # Create training environment
        env = gym.make("Blackjack-v1", sab=True)
        env = gym.wrappers.RecordEpisodeStatistics(env, buffer_length=n_episodes)
        learning_rate = LR[i]
        agent = BlackjackAgent(  env=env,
                                 learning_rate=learning_rate,
                                 initial_epsilon=start_epsilon,
                                 epsilon_decay=epsilon_decay,
                                 final_epsilon=final_epsilon,)

        # Temporary lists for this LR
        current_lr_eval_scores = []
        current_lr_eval_steps = []
        print(f"\n--- Start Training LR={learning_rate} ---")

        # TRAINING PHASE WITH PERIODIC EVALUATION
        for episode in tqdm(range(n_episodes), desc=f"Training"):
            # 1. PERIODIC EVALUATION (Before playing the episode)
            if episode % eval_interval == 0:
                # Create a temporary env to avoid polluting training stats
                eval_env = gym.make("Blackjack-v1", sab=True)
                wins = 0
                # Fast evaluation loop (no tqdm to avoid spamming the console)
                for _ in range(n_eval_episodes):
                    obs_eval, _ = eval_env.reset()
                    done_eval = False
                    while not done_eval:
                        # Force greedy=True to use the pure Q-table
                        action_eval = agent.get_action(eval_env, obs_eval, force_greedy=True)
                        next_obs_eval, reward_eval, term_eval, trunc_eval, _ = eval_env.step(action_eval)
                        if reward_eval == 1.0:
                            wins += 1
                        done_eval = term_eval or trunc_eval
```

```
                    obs_eval = next_obs_eval
                eval_env.close()
                win_rate = (wins / n_eval_episodes) * 100
                current_lr_eval_scores.append(win_rate)
                current_lr_eval_steps.append(episode)

            # 2. STANDARD TRAINING
            obs, info = env.reset()
            done = False
            while not done:
                action = agent.get_action(env, obs)
                next_obs, reward, terminated, truncated, info = env.step(action)
                agent.update(obs, action, reward, terminated, next_obs)
                done = terminated or truncated
                obs = next_obs
            agent.decay_epsilon()

        # Retrieve results
        returns = np.array(env.return_queue).flatten()
        reward_moving_average = (np.convolve(returns, np.ones(rolling_length), mode="valid") / rolling_length)
        # win average
        wins_train = (returns == 1.0)
        win_moving_average = (np.convolve(wins_train, np.ones(rolling_length), mode="valid") / rolling_length)

        if e == 0:
            sum_current_lr_eval_scores = np.array(current_lr_eval_scores)
            sum_reward_moving_average = np.array(reward_moving_average)
            sum_win_moving_average = np.array(win_moving_average)
        else:
            sum_current_lr_eval_scores = sum_current_lr_eval_scores + np.array(current_lr_eval_scores)
            sum_reward_moving_average = sum_reward_moving_average + np.array(reward_moving_average)
            sum_win_moving_average = sum_win_moving_average + np.array(win_moving_average)

        # End of simulation
        env.close()

    # Average and store results for each LR
    periodic_eval_curves.append((current_lr_eval_steps, sum_current_lr_eval_scores / NBE))
    all_reward_curves.append(sum_reward_moving_average / NBE)
    all_win_curves.append(sum_win_moving_average / NBE)

# SAVE
np.savetxt("reward.csv", all_reward_curves, delimiter=',')
np.savetxt("win.csv", all_win_curves, delimiter=',')
np.savetxt("eval.csv", [i[1] for i in periodic_eval_curves], delimiter=',')

# PLOTS
plt.figure(figsize=(16, 5))

# Plot 1: Reward (Training)
plt.subplot(1, 3, 1)
for idx, curve in enumerate(all_reward_curves):
```

```
    if len(curve) > 0:
        plt.plot(curve, label=f'LR = {LR[idx]}')
plt.title("Mean reward during training")
plt.xlabel("Epoch")
plt.ylabel("Mean reward")
plt.legend()
plt.grid(True, alpha=0.3)

# Plot 2: Success Rate (Training - with Epsilon)
plt.subplot(1, 3, 2)
for idx, curve in enumerate(all_win_curves):
    if len(curve) > 0:
        plt.plot(curve * 100, label=f'LR = {LR[idx]}')
plt.title("Mean win rate during training")
plt.xlabel("Epoch")
plt.ylabel("Mean win rate (%)")
plt.grid(True)
plt.ylim([27, 43])

# Plot 3: PURE EVALUATION (New!)
plt.subplot(1, 3, 3)
for idx, (steps, scores) in enumerate(periodic_eval_curves):
    plt.plot(steps, scores, label=f'LR = {LR[idx]}')

plt.title(f"Mean strategy win rate ({n_eval_episodes} trials)")
plt.xlabel("Epoch")
plt.ylabel("Mean win rate (%)")
plt.legend()
plt.grid(True)
plt.ylim([27, 43])
plt.tight_layout()
plt.show()
```