Blackjack Report

Author: 202104636 Rui Filipe Castro Martins

1 Introduction

Reinforcement Learning (RL) is a powerful way to train models without relying specifically in supervised learning. Based on the concept of action-reward, the agent (the one to be trained) is placed inside the environment where it can interact with it according to its own observations of the environment. From those observations, the agent can act. Actions can be either benefitial (the agent gets closer to the goal) or detrimental (the agent can end up "losing").

In this project, the main goal is to design and train an agent to successfully learn how to play a game of blackjack. However, it is worth noticing that this is not a traditional blackjack; this one is called Easy21 and is based on a set of rules that sets it apart from the original game played in casinos.

2 Easy21

Easy21 can be seen as a spinoff of the original Blackjack game, where, for instance, cards like aces and faces cannot be drawed and the probability of drawing a red card is smaller than the probability of drwaing a black one. Also, red card should be counted as negative values for the final sum while black card as positive.

This simplified version allows the player to choose from two independent actions: HIT or STICK. The player, in a first moment, is showed two card: one corresponding to the initial card of the dealer, and one that belongs to himself. From this observation the player chooses wether he wants to gamble and receive another card by HITting or if he prefers to simply STICK to his current hand.

The winner is found by summing the value of the cards in each hand and evaluating which one is greater or by checking if either the player or the dealer busted by going over 21 or below 1.

There are four scenarios when it comes to the final result:

- Both player and dealer get the same value when summing their hands which means there is a DRAW;
- The player has a greater hand than the dealer and WINS;
- The dealer has a greater hand than the player and WINS;
- Either the player or the dealer busts and the other side WINS.

Given the nature of this game, it is necessary to create a custom environment class that handles all the logic behind it. The subsequent snippets of code, shown in Listing 1 highlight the most important parts of the codebase.

This custom environment allows us to quickly generate synthetic data to train the agent.

3 Agent

To better implement a modular agent that adapts to different scenarios, a custom class and script were developed to guarantee this characteristic. Despite being based on past implementations such as the one found under OpenAI's Gymnasium documentation, this agent was fully built from scratch with simplicity in mind.

As seen in Listing 2, a config file was created to quickly modify the parameters of both environment and agent without having to access them internally. Listing 3 briefly describes this config file.

3.1 Monte Carlo

A Monte Carlo simulation is a computer-based mathematical technique that uses repeated random sampling to model the probability of different outcomes in an uncertain process. Since, in this case, we developed a model of the environment and agent, it is possible to use Monte Carlo simulation to quickly come up with a steady state estimation of the Value Function.

Some rules were stablished to ensure reproducibility and consistency.

- Value function must be initialized to zero;
- The Learning Rate value must follow the equation 1;

$$\alpha_t = \frac{1}{N(s_t, a_t)} \tag{1}$$

- The Epsilon value, the one who dictates the ration between exploration and exploitation, must follow the equation 2;

$$\epsilon_t = \frac{N_0}{N_0 + N(s_t)} \tag{2}$$

By default, the expected value of N0 is 100, though other values should also be tested in case they produce better results in comparison.

In the end, and with the output provided by the Monte Carlo simulation, it is possible to compute and plot a 3D visualization of the Value Function for different combinations of observations. A heatmap and a 3D representation of the Value Function were created and can be seen in Figures 1 and 2.

3.2 TD Learning with Sarsa(λ)

TD (Temporal Difference) learning is a reinforcement learning method that combines ideas from Monte Carlo and dynamic programming to predict the value of states. Unlike Monte Carlo methods that wait until the end of an episode, TD learning updates value estimates after each step based on subsequent estimates.

In this project, Sarsa(λ) was implemented, which extends the basic Sarsa algorithm with eligibility traces. Sarsa (State-Action-Reward-State-Action) is an on-policy reinforcement learning algorithm that learns by following its current policy. The addition of the λ parameter allows the algorithm to bridge between one-step TD learning ($\lambda = 0$) and Monte Carlo methods ($\lambda = 1$).

3.2.1 Eligibility Traces

Eligibility traces provide a mechanism for efficient credit assignment by maintaining a memory of which state-action pairs have been recently visited. The trace for each state-action pair is updated as shown in Equation 3.

$$E(s_t, a_t) \leftarrow E(s_t, a_t) + 1 \tag{3}$$

After each step, all eligibility traces decay according to Equation 4.

$$E(s, a) \leftarrow \gamma\lambda E(s, a) \tag{4}$$

where γ is the discount factor (set to 1.0 for this undiscounted task) and λ controls the degree of bootstrapping.

3.2.2 Update Rule

The Q-value updates in Sarsa(λ) use the TD error δ_t and eligibility traces as shown in Equations 5 and 6.

$$\delta_t = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{5}$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha_t \delta_t E(s, a), \quad \forall s, a \tag{6}$$

This update rule allows the TD error to propagate backward through the episode to all previously visited state-action pairs, weighted by their eligibility traces.

3.2.3 Experimental Setup

The same learning rate and exploration schedules from the Monte Carlo implementation were maintained for consistency:

- Learning rate: $\alpha_t = 1/N(s_t, a_t)$
- Exploration rate: $\epsilon_t = N_0/(N_0 + N(s_t))$ with $N_0 = 100$

Experiments were conducted for $\lambda \in \{0.0, 0.1, 0.2, \ldots, 1.0\}$, with each configuration running for 1000 episodes. The mean-squared error (MSE), defined in Equation 7, was computed to compare the learned Q-values against the optimal values Q^* obtained from the Monte Carlo simulation with 1,000,000 episodes.

$$\text{MSE} = \sum_{s,a} (Q(s, a) - Q^*(s, a))^2 \tag{7}$$

For $\lambda = 0$ and $\lambda = 1$, the MSE was tracked at each episode to analyze the learning dynamics throughout training.

Metric	Value
Total Episodes	1,000,000
Total Rewards	141,417.0
Average Reward per Episode	0.14
Win Rate	56.66%

Table 1: Monte Carlo simulation results after 1,000,000 episodes.

4 Results

4.1 Monte Carlo

The Monte Carlo control algorithm was run for 1,000,000 episodes to obtain an accurate estimate of the optimal Q-values (Q*). This serves as the baseline for evaluating the TD learning methods. Table 1 summarizes the performance metrics.

The results show that the optimal policy achieves a win rate of approximately 57%, which is better than random play (approximately 42% based on game dynamics). The positive average reward indicates that the learned policy successfully exploits the game structure.

Heat Map and 3D Map of the Value Function:

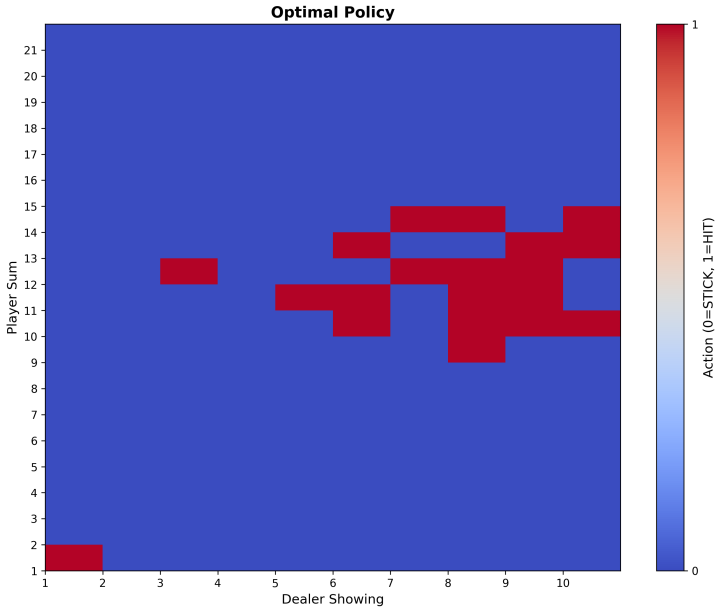

Figure 1: Monte Carlo - Heat Map.

4.2 TD Learning with Sarsa(λ)

The Sarsa(λ) algorithm was tested with 11 different values of λ ranging from 0.0 to 1.0 in increments of 0.1. Each experiment ran for 1000 episodes, and the mean-squared error was computed by comparing the learned Q-values against the optimal Q* values from the Monte Carlo baseline.

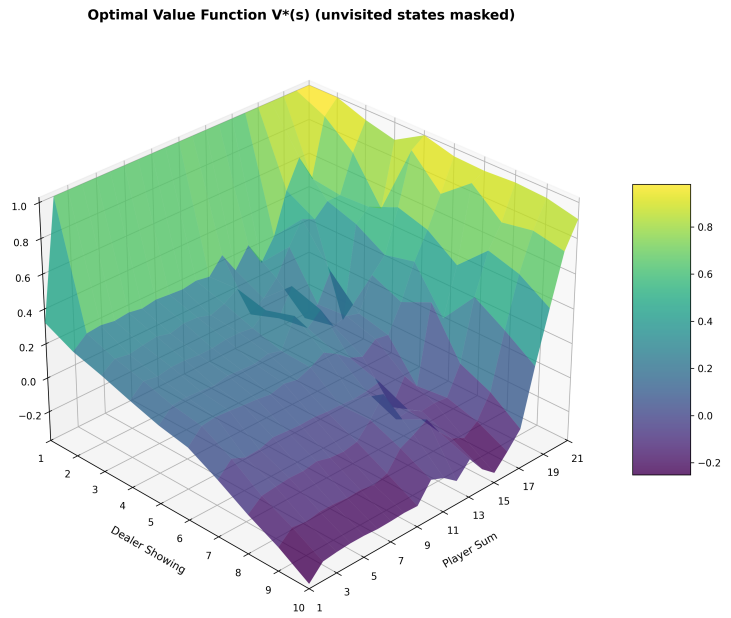

Figure 2: Monte Carlo - Value function represented using a 3D chart.

4.2.1 MSE vs Lambda Analysis

Table 2 presents the final MSE values for each λ configuration. The results show that $\lambda = 0.1$ achieved the best performance with an MSE of 0.204343, while $\lambda = 1.0$ had the worst performance with an MSE of 0.482572.

λ	MSE
0.0	0.237316
0.1	**0.204343**
0.2	0.226315
0.3	0.220161
0.4	0.220093
0.5	0.247632
0.6	0.266126
0.7	0.277020
0.8	0.301613
0.9	0.371927
1.0	0.482572

Table 2: Mean-squared error for different λ values after 1000 episodes.

Figure 3 visualizes the relationship between λ and MSE. The plot reveals a clear trend: performance degrades as λ increases beyond 0.1. This suggests that with only 1000 episodes, eligibility traces that propagate too far backward (higher λ values) introduce more variance than benefit. Lower λ values, which focus more on immediate temporal differences, converge faster with limited training data.

4.2.2 Learning Curves Analysis

Figure 4 shows the evolution of MSE over 1000 episodes for $\lambda = 0$ (pure Sarsa) and $\lambda = 1$ (Monte Carlo-like). The $\lambda = 0$ curve demonstrates more stable convergence, while $\lambda = 1$

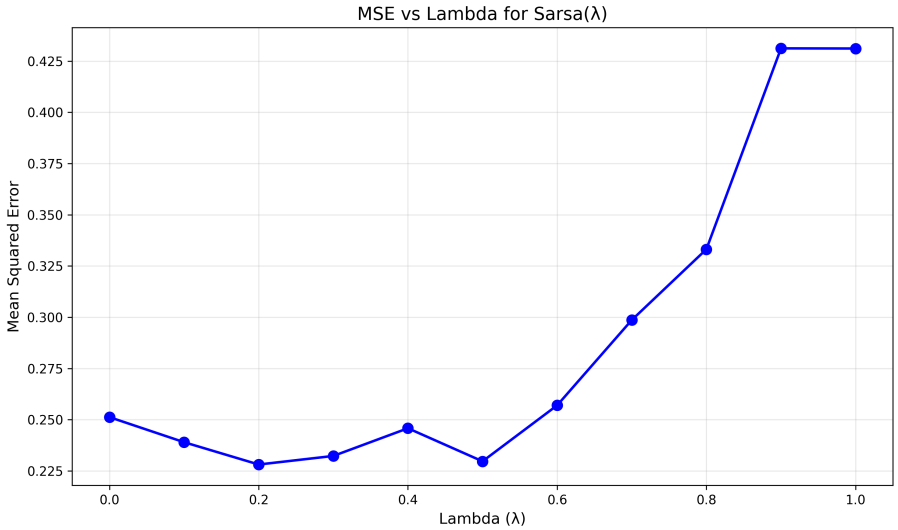

Figure 3: Mean-squared error as a function of λ. The minimum occurs at $\lambda = 0.1$, demonstrating that a small amount of eligibility trace provides the best balance between bias and variance for this task with 1000 episodes.

exhibits higher variance throughout training, which explains its poorer final performance.

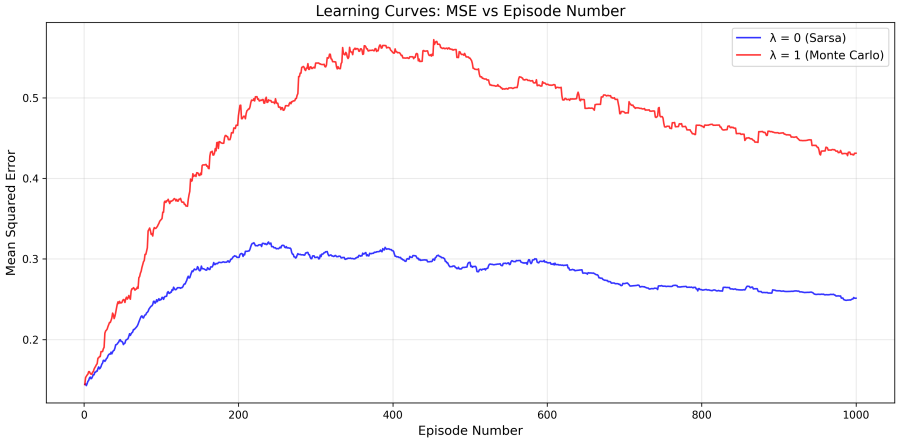

Figure 4: Learning curves comparing $\lambda = 0$ (Sarsa) and $\lambda = 1$ (Monte Carlo-like). The Sarsa method ($\lambda = 0$) shows faster and more stable convergence compared to the Monte Carlo approach with limited episodes.

The key observation is that $\lambda = 0$ (one-step Sarsa) outperforms $\lambda = 1$ (full episode backup) when training is limited to 1000 episodes. This is expected because Monte Carlo methods require more samples to achieve low variance estimates, while TD methods can learn effectively from fewer samples by bootstrapping from current estimates.

5 Conclusions

This project successfully implemented and compared two fundamental reinforcement learning approaches for solving the Easy21 card game: Monte Carlo simulation and Temporal Difference learning with Sarsa(λ). The custom environment and modular agent architecture developed from scratch demonstrated the practical application of action-value methods in a stochastic decision-making scenario.

The Monte Carlo approach, using first-visit updates with episode-based learning, converged to a stable value function after 1,000,000 episodes. The adaptive learning rate $\alpha_t = 1/N(s_t, a_t)$ and epsilon-greedy exploration strategy $\epsilon_t = N_0/(N_0 + N(s_t))$ with $N_0 = 100$ enabled effective balance between exploration and exploitation. The resulting policy heatmap and 3D value function visualization clearly illustrated the agent's learned decision boundaries for different combinations of player and dealer cards. This optimal Q* matrix served as the ground truth for evaluating the TD learning experiments.

The Sarsa(λ) implementation with eligibility traces provided a comprehensive study of the bias-variance tradeoff in TD learning. Experiments across 11 different λ values revealed that $\lambda = 0.1$ achieved the lowest MSE (0.204343) after 1000 episodes, outperforming both pure one-step Sarsa ($\lambda = 0$, MSE = 0.237316) and Monte Carlo-like methods ($\lambda = 1$, MSE = 0.482572). This demonstrates that a small amount of eligibility trace provides the optimal balance between immediate TD updates and full episode backups when training with limited samples.

The learning curve analysis comparing $\lambda = 0$ and $\lambda = 1$ highlighted a fundamental insight: TD methods converge faster with fewer episodes by bootstrapping from current estimates, while Monte Carlo methods require substantially more samples to achieve low-variance Q-value estimates. The progressive degradation of performance as λ increases beyond 0.1 indicates that excessive credit assignment to distant state-action pairs introduces more noise than signal in the early stages of learning.

The implementation's modular design, with configuration-driven parameters and separation of concerns between environment, agent, and visualization components, facilitated systematic experimentation and reproducible results. The project demonstrated that reinforcement learning can effectively solve card game scenarios without explicit knowledge of optimal strategies, relying solely on trial-and-error interaction with the environment.

In conclusion, the comparison between Monte Carlo and Sarsa(λ) methods highlighted important trade-offs: Monte Carlo provides unbiased estimates but requires many episodes to converge, while TD methods with appropriate λ values offer faster learning but may introduce bias. For the Easy21 task with limited training budget, Sarsa with small eligibility traces ($\lambda \approx 0.1$) represents the most sample-efficient approach.

6 Annexes

```python
class Easy21:
    def __init__(self):

        # Observations
        self.cards_dealer: list[int] = []    # First card: 1-10
        self.cards_player: list[int] = []    # Total sum: 1-21

        # Actions
        self.actions: list[str] = [STICK, HIT]    # Stick, Hit

        # Probabilities
        self.draw_values: tuple[int] = (1, 10) # If we draw a red card
           we have (-10, -1) instead
        self.prob_red: float    = 1/3
        self.prob_black: float  = 2/3
```

Listing 1: Easy21 Environment Class Initialization

```python
class Agent:
    def __init__(
            self,
            observation: tuple[int] = _config.SCENARIO_OBSERVATIONS,
            num_observations: tuple[int] = _config.
               SCENARIO_OBSERVATIONS_NUM,
            actions: tuple[int] = _config.SCENARIO_ACTIONS,
            num_actions: int = _config.SCENARIO_ACTIONS_NUM,
            gamma: float = _config.GAMMA,
            N_0: float = _config.N_0,
            monte_carlo: bool = _config.MONTE_CARLO,
            lambda_param: float = 0.0
        ):

        # Info
        self.prev_observation  = observation
        self.num_observations  = num_observations
        self.actions           = actions
        self.num_actions       = num_actions

        # Q-matrix shape: (num_actions, dealer_card, player_sum)
        self.q_matrix          = np.zeros((self.num_actions, *self.
           num_observations))
        self.is_training: bool = False

        # Training parameters
        self.monte_carlo: bool    = monte_carlo
        self.gamma: float         = gamma
        self.N_0: float           = N_0
        self.lambda_param: float  = lambda_param

        # Visit counters
        self.state_action_count   = np.zeros((self.num_actions, *self.
           num_observations))
        self.state_count          = np.zeros(self.num_observations)
```

```
        # Episode trajectory for Monte Carlo
        self.episode_trajectory: list[tuple[tuple[int], int, float]] =
            []

        # Eligibility traces for Sarsa(lambda)
        self.eligibility_traces   = np.zeros((self.num_actions, *self.
            num_observations))
        self.prev_action: int = 0
```

Listing 2: Agent Class Initialization with Sarsa(λ) Support

```
# Scenario
SCENARIO_OBSERVATIONS: tuple[int]     = (0, 0)
SCENARIO_OBSERVATIONS_NUM: tuple[int] = (10+1, 21+1)  # +1 offset since
    array starts at [0]
SCENARIO_ACTIONS: tuple[int]          = (0, 1)
SCENARIO_ACTIONS_NUM: int             = len(SCENARIO_ACTIONS)

# Agent
AGENT_TRAIN: bool = True

# Episode
if AGENT_TRAIN:
    NUM_EPISODES: int = 1_000_000
else:
    NUM_EPISODES: int = 10_000

# Training
MONTE_CARLO: bool  = True   # True: Monte Carlo, False: TD Learning
GAMMA: float       = 1.0    # Discount Factors
N_0: float         = 100.0  # Epsilon constant for exploration

# Path
PATH_SAVE_IMAGES: str   = "model/images/"
PATH_SAVE_Q_MATRIX: str = "model/q_matrix/"
```

Listing 3: Configuration File