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

TD (Temporal Difference) learning is a type of reinforcement learning method that uses a combination of Monte Carlo and dynamic programming to predict the future value of a state.

In our case we use the notion of SARSA to compute the desired metrics. SARSA (State-Action-Reward-State-Action) is an on-policy reinforcement learning algorithm that helps an agent to learn an optimal policy by interacting with its environment. The agent explores its environment, takes actions, receives feedback and continuously updates its behavior to maximize long-term rewards.

The real difference between the Monte Carlo simulation method and SARSA comes down to the moment where each of one of them performs the update logic. Monte Carlo tracks the entire trajectory of the episode and only updates at the end of it while SARSA updates every time it selects an action and receives the instant reward.

Additionally, the mean-squared error, as represented in Equation 3, was computed every 1000 episodes to later compare the results for different λ values.

$$\sum_{s,a}(Q(s,a) - Q^*(s,a))^2 \tag{3}$$

4 Results

4.1 Monte Carlo

Heat Map and 3D Map of the Value Function:

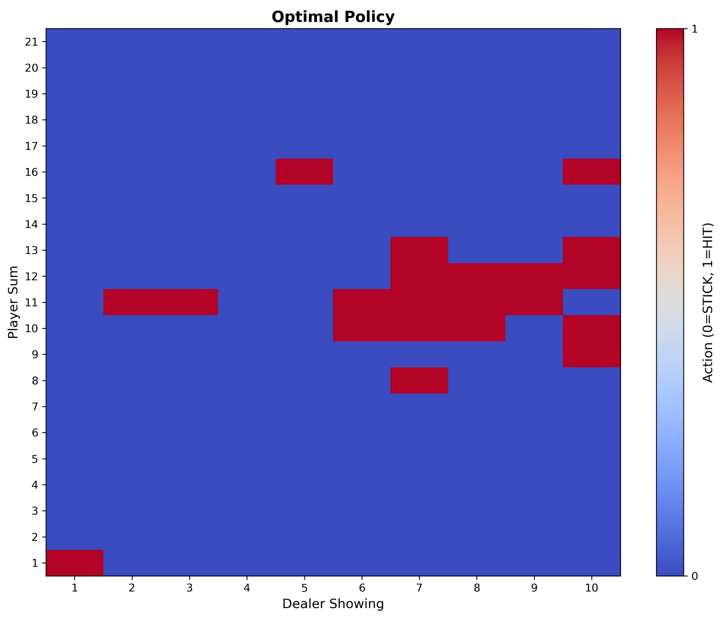

Figure 1: Monte Carlo - Heat Map.

4.2 TD Learning

Heat Map and 3D Map of the Value Function:

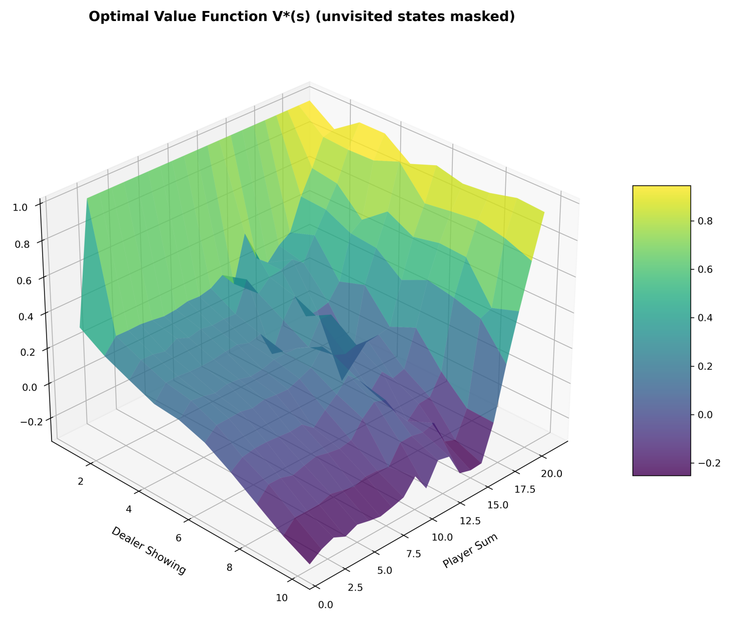

Figure 2: Monte Carlo - Value function represented using a 3D chart.

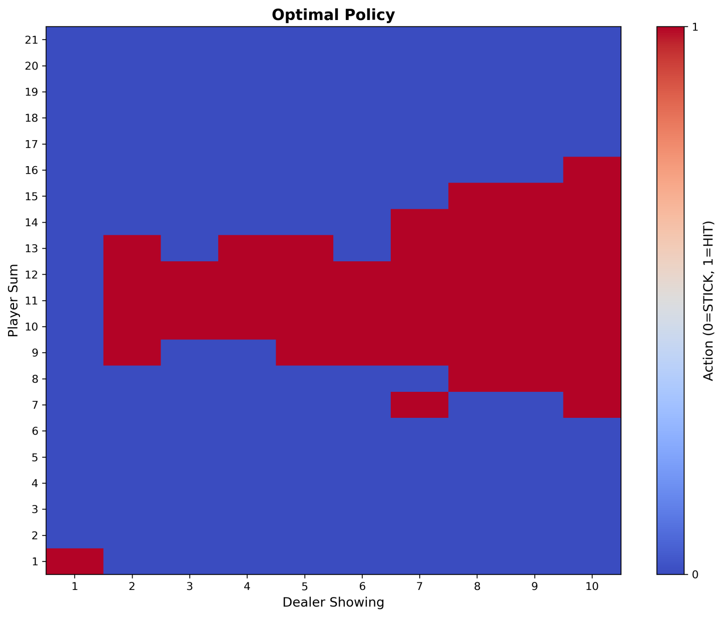

Figure 3: TD Learning - Heat Map.

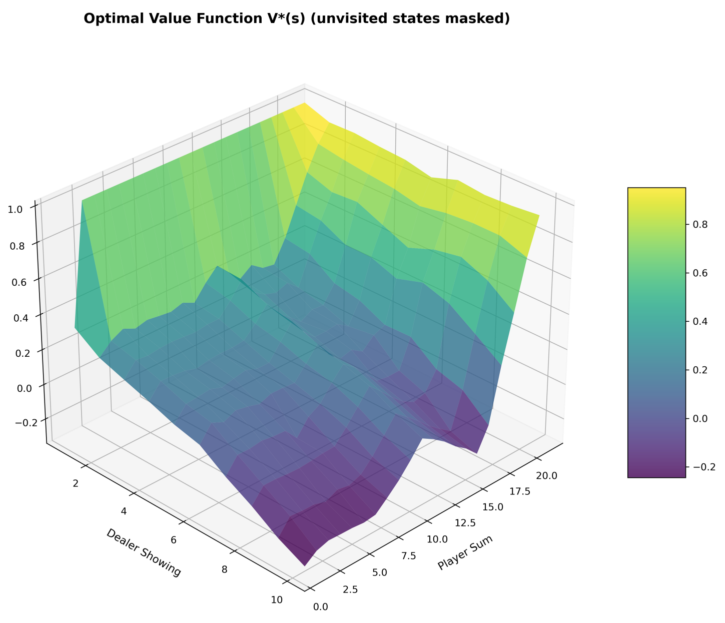

Figure 4: TD Learning - Value function represented using a 3D chart.

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 (TD) learning with SARSA. 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.

The TD learning implementation with SARSA provided an alternative approach through step-by-step updates rather than episode-end batch updates. This online learning characteristic allowed the agent to adapt its policy during each episode, potentially offering faster convergence for certain state-action pairs. The mean-squared error metric computed every 1000 episodes provided quantitative assessment of learning progress and enabled comparison with the Monte Carlo baseline.

Both methods successfully learned viable blackjack strategies, as evidenced by the value function visualizations presented in Figures 1, 2, 3, and 4. 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. The comparison between Monte Carlo and TD methods highlighted the trade-offs between episode-based and incremental learning approaches in the context of episodic tasks with terminal rewards.

6 Annexes

```
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

```
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,
            alpha: float = _config.ALPHA,
            gamma: float = _config.GAMMA,
            epsilon: float = _config.EPSILON,
            epsilon_decay_factor: float = _config.EPSILON_DECAY_FACTOR,
            epsilon_min: float = _config.EPSILON_MIN
        ):

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

        self.actions            = actions
        self.num_actions        = num_actions

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

        # Training
        self.epsilon: float        = epsilon
        self.epsilon_decay_factor  = epsilon_decay_factor
        self.epsilon_min: float    = epsilon_min
        self.lr: float             = alpha
        self.gamma: float          = gamma
```

Listing 2: Agent Class Initialization

```
# Scenario
SCENARIO_OBSERVATIONS: tuple[int]     = (0, 0)
SCENARIO_OBSERVATIONS_NUM: tuple[int] = (10+1, 10+1) #(21+1, 21+1)
           # +1 is the 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
EPSILON: float               = 1.0
EPSILON_DECAY_FACTOR: float  = 0.99
EPSILON_MIN: float           = 0.1
ALPHA: float                 = 0.1    # Learning Rate
GAMMA: float                 = 0.99   # Discount Factor

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

Listing 3: Configuration File