Project Report: Reinforcement Learning and Decision Making Under Uncertainty

Authors: Matteo Biner, Benno Thalmann

1. Introduction

The goal of this project is to develop an optimal strategy for playing Blackjack using Reinforcement Learning (RL). We focused on a multiplayer setting against the dealer, implementing and evaluating a Q-learning approach. This report outlines our methodology, experiment design, results, and analysis.

2. Domain Description and Goals

Blackjack is a popular card game where the objective is to have a hand value as close to 21 as possible without exceeding it. Players compete against the dealer and make decisions such as hitting (taking another card) or standing (keeping their current hand).

Our goal is to develop a strategy using RL techniques, particularly Q-learning, to optimize the player's actions to maximize long-term rewards. We have taken an existing implementation (https://gist.github.com/jukujala/eb86e6fcd1570340d1b4171b440e3ca5) and adapted it so that several players and a Q-learning agent can play at the same time against one dealer. The rules are as close as possible to the official rules, certain side rules such as insurance, splitting or doubling down have not been taken into account. Additionally, because there is no fixed rule for the number of decks used, we decided to use 6 decks by default. According to (https://betandbeat.com/blackjack/blog/how-many-decks-of-cards-in-blackjack/) this is a reasonable number of decks. We were not able to find an improvement for a smaller deck size.

3. Methodology

3.1 Markov Decision Process (MDP)

We formalized the game of Blackjack as a Markov Decision Process (MDP), where:

- **States** represent the combinations of the player's hand, the dealer's visible card, and whether the player has usable aces.
- **Actions** are the available moves: hitting or standing.
- **Rewards** are the outcomes of the game: +1 for a win, -1 for a loss, and 0 for a draw.

3.2 Q-learning

Q-learning was used to learn the optimal policy for playing Blackjack. The Q-value function $Q(s, a)$ estimates the expected utility of taking action a in state s and following the optimal policy thereafter. The update rule is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

where:

- α is the learning rate.
- γ is the discount factor.
- r is the reward received after taking action a in state s.
- s' is the next state.

We were inspired by https://gymnasium.farama.org/tutorials/training_agents/blackjack_tutorial/ in our implementation, but since we used another implementation of the Blackjack simulator, we adapted the procedure to our environment.

Since we have a multiplayer game, the state s initially consisted of a tuple [player_sum, dealer_hand, usable_aces, other_players_sum], taking into account the sum of all visible cards of the other players. However, this made the Q-Table extremely bloated and the agent was unable to deliver good performance despite long training episodes. We have therefore deleted the other_players_sum from the tuple.

3.3 Implementation Details

The implementation includes four main components:

1. **Blackjack Simulator**: Simulates the game environment.
2. **Strategies**: Various predefined strategies for comparison.
3. **Agent**: The RL agent using Q-learning to learn the optimal policy.
4. **Training and Evaluation**: Training the agent and evaluating its performance.

The code was divided into four files:

- **python_blackjack_simulator.py**
- **strategies.py**
- **train_and_evaluate.py**
- **BlackJackAgentV2.py**

As seen in the list of the files, the Blackjack agent is the second version. We started with a more basic version, which did not perform that well considering the performance of the gained rewards. A key error of the first version was, that it took only into account the final reward during one game, which would punish earlier actions if the last one was badly chosen. Also, the agent's learning process was modified to improve its

performance by adding an additional reward during training based on the *expected value strategy* we implemented earlier to outperform a naïve *"hit until 17"-strategy*.

---

4. Experiment Design

4.1 Training

The agent was trained using the Blackjack simulator, with parameters:

- Learning rate (α): 0.01
- Initial exploration rate (ϵ): 1
- Exploration decay: $2/\text{repetitions}$
- Final exploration rate: 0.1
- Discount factor: 0.95

We trained the agent for various numbers of episodes (as seen in *5. Results*), gradually reducing exploration to exploit learned policies. The hyperparameters were chosen based on different tutorials and guides we found online. We have not done any precise cross-validation hyperparameter tuning, but the hyperparameters stated above are the result of countless adjustments and optimizations made during the work process.

4.2 Evaluation

The trained agent was evaluated over different amounts of episodes. Performance metrics included average rewards and the average number of actions per game. These metrics were then compared to the performance of our other strategies. Those were:

- Hit until bust, which would always get another card,
- A random action, which would get another card with a 50% chance,
- Hit until 17, which would always get a card if its current score was below 17,
- Expected value strategy, which would calculate the expected score when taking another card. It would then only take a card if its current score plus the expected score was below 21 and if the score including the next card is closer to 21 (important if the player is holding a usable ace).

4.3 Performance Metrics

- **Average Reward**: The average reward over a large number of games, indicating the effectiveness of the strategy.
- **Average Length**: The average number of actions per game, indicating how quickly the game reaches a conclusion. This is useful to compare the agent's aggressiveness to other strategies.

---

5. Results

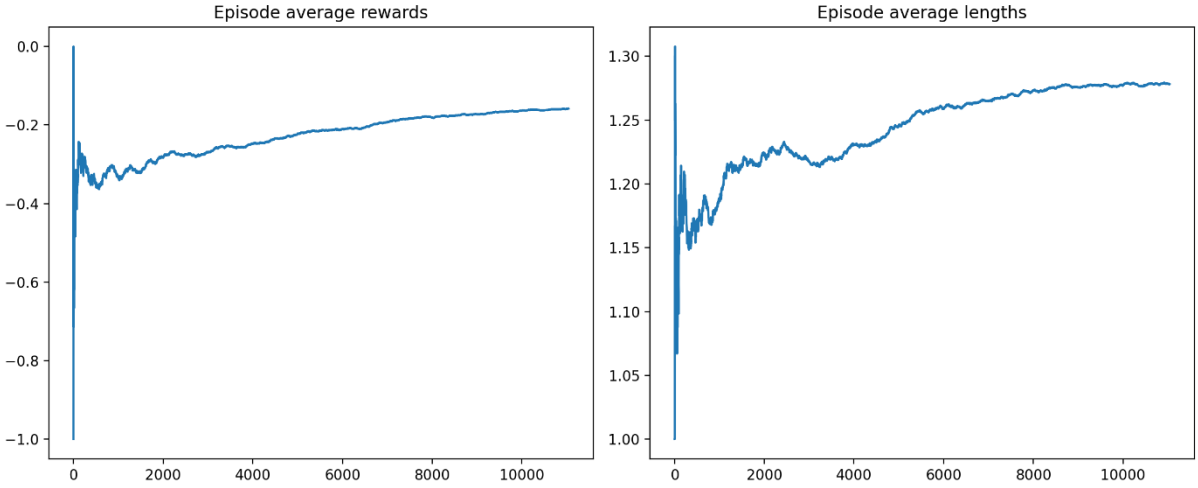
The baseline resulting average rewards for our other strategies after 10’000 games were as follows:

- Hit until bust: -0.6612
- Random action: -0.3072
- Hit until 17: -0.0808
- Expected value strategy: -0.0582

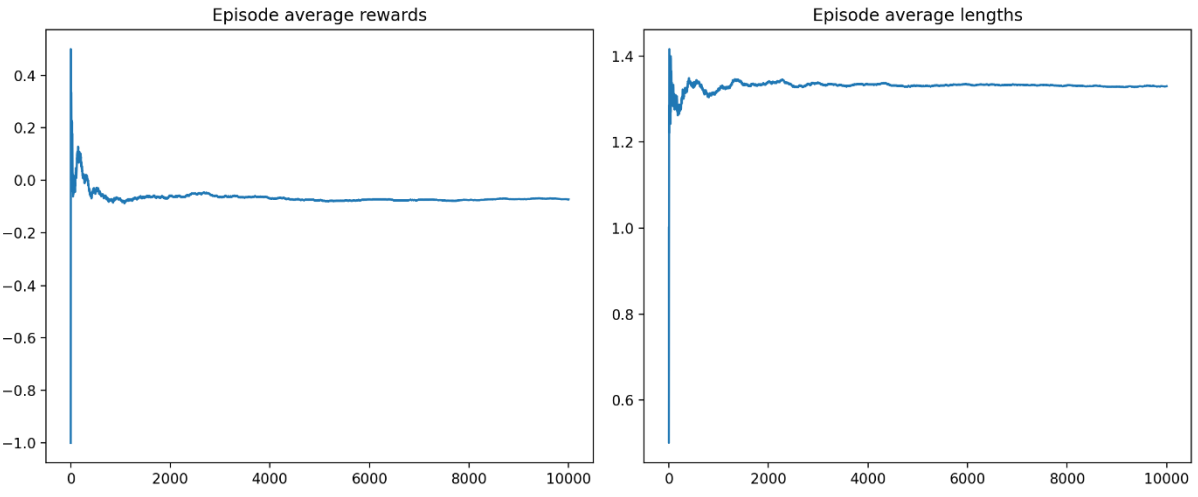
The first version of our agent only outperformed the two worst strategies with a resulting average reward of -0.1717.

The following plots show the performance of the trained agent:

Average reward during training 10k repetitions: -0.1584

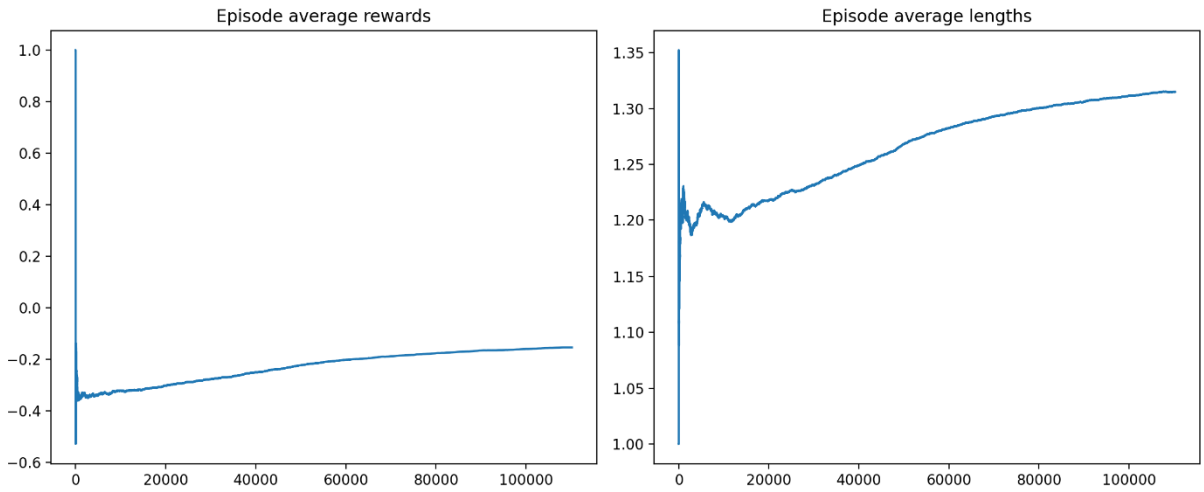


Average Reward with trained agent 10k repetitions: -0.0733

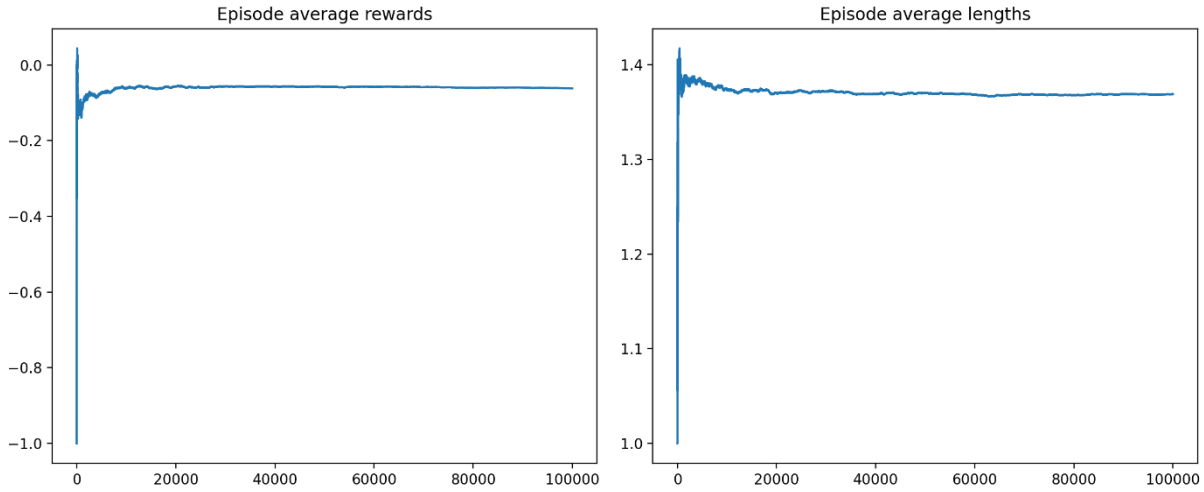


Because there were still minor improvements after 10’000 repetitions, we trained the agent for 100’000 repetitions and later for 1’000’000 repetitions. After which we were able to beat the expected value strategy.

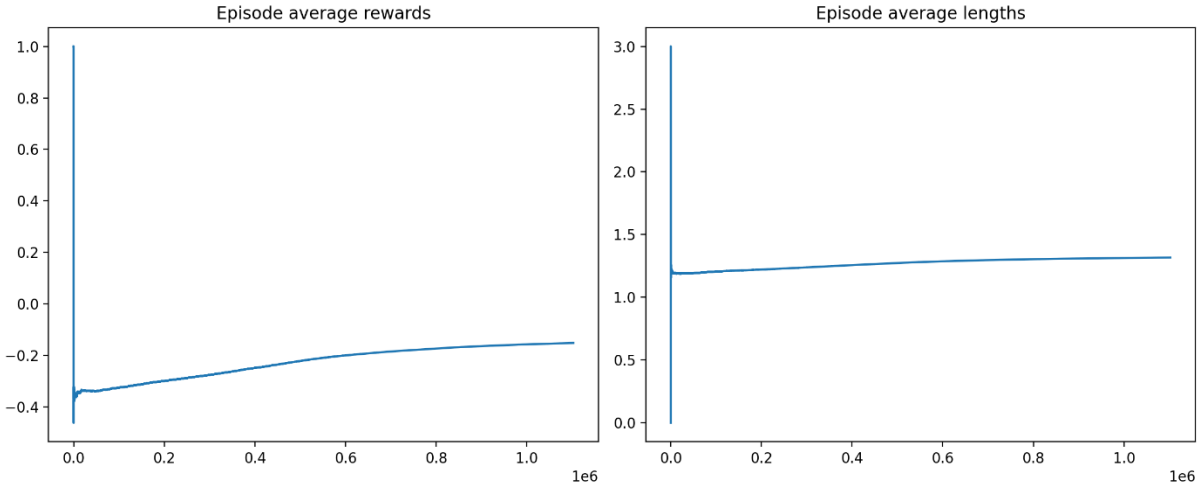
Average Reward during training 100k repetitions: -0.15368



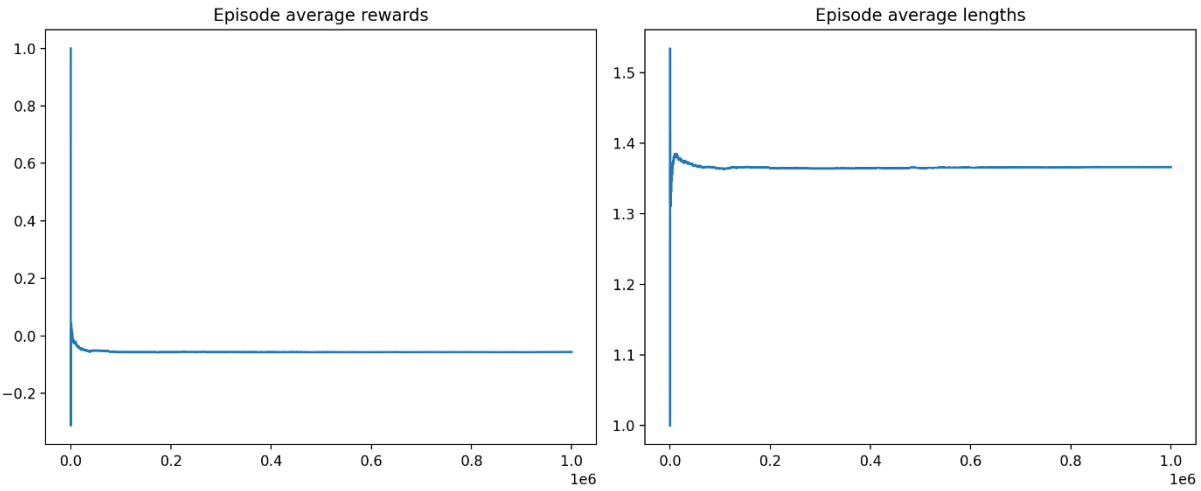
Average Reward with trained agent 100k repetitions: -0.06195



Average Reward during training 1M repetitions: -0.151343



Average Reward with trained agent 1M repetitions: -0.056122



---

6. Analysis and Discussion

6.1 Technical Correctness

Our implementation of the Blackjack simulator and Q-learning algorithm is technically sound, which can be found in https://github.com/MatteoRBiner/ReinforcementLearning. The agent's learning process was modified from the standard Q-learning update rules to improve its performance as mentioned above.

6.2 Model Assumptions

We assumed the standard rules of Blackjack without additional complexities like Insurance or Splitting. This simplified the state space, making the learning process more tractable.

6.3 Optimality and Robustness

Our Q-learning agent learned a strategy that performed better than the baseline strategy on average, indicating successful learning. However, the inherent house edge in Blackjack means that long-term returns are still negative, as expected. Although AgentV2 is noticeably better than AgentV1, the expected value strategy could only be beaten after 1’000’000 games of training.

6.4 Independent Thinking

We explored the impact of additional state information (such as the sum of other players' hands) on the agent's performance, contributing to our understanding of state representation in RL for Blackjack. We thought that the more information for the agent, the better his performance would be. This was a mistake, because the state space became too large and despite an increase in learning episodes, the agent could not keep up with the bloated Q-Table.

6.5 Limitations and Future Work

The main limitation is the convergence speed of Q-learning. Future work could explore other RL algorithms like Deep Q-Networks (DQN) or policy gradient methods to improve learning efficiency and performance.

---

7. Conclusion

This project successfully applied Q-learning to develop a Blackjack strategy that outperformed a baseline strategy. The agent learned to make decisions that maximized long-term rewards, demonstrating the potential of RL in game strategy optimization. Further research could enhance the strategy by incorporating more complex game rules and advanced RL algorithms.