Agent trained to play Blackjack

Officer-Students R. Callens & J. Weemaes

Abstract
This project implements a Q-learning agent to play the game of blackjack using the OpenAI Gym environment. The agent learns optimal strategies through trial and error, aiming to maximize its total reward. The rewards initially set as +1.2 for a win and -0.8 for a loss were later adjusted to +1 and -1, yielding better results and reflecting actual game rewards. The agent's performance was evaluated using specific hyperparameters: a learning rate of 0.01, a discount factor of 0.2, and a greedy exploration rate of 0.41. The agent achieved a win rate of 47.46%, surpassing random chance and outperforming fixed strategies, though not yet reaching a consistently winning strategy. Further improvements would require additional resources.

Keywords
Reinforcement learning, Q-learning, Blackjack, OpenAI Gym, Hyperparameters, Win rate.

Department MWMW, Royal Military Academy, Brussels, Belgium

Contents

1. Introduction

Reinforcement learning is a dynamic field of machine learning where agents learn to make optimal decisions by interacting with their environment and receiving rewards. In this project, reinforcement learning techniques are applied to the game of blackjack, a strategic card game with uncertain outcomes. The objective is to develop an agent capable of learning effective strategies for playing blackjack and evaluate its performance against different strategies.

Using the Q-learning algorithm, the agent estimates future rewards for each action in a given state and selects optimal actions accordingly. The state space is defined to represent the game state, the action space to encompass possible moves, and a reward function has to be found that incentivizes winning and discourages losses.

Through training, the algorithm's hyperparameters are fine-tuned to optimize the learning process. Next to the training, the agent's performance will be looked after by comparing its win rate to random actions and fixed strategies based on the player's cards. Additionally, the convergence of the algorithm will be defined by studying the relationship between training episodes and the agent's win rate.

This project offers practical experience in reinforcement learning and showcases understanding of key concepts and techniques.

2. Background

This section provides a brief overview of the game of blackjack, including its rules and objectives, as well as an introduction to the concepts of reinforcement learning and different types of reinforcement learning algorithms.

2.1 Rules of Blackjack

Blackjack is a card game that is played with one or more standard decks of 52 cards. The game revolves around the objective of achieving a hand total that is closer to 21 than the dealer's hand, without exceeding 21. In the following part the different parts of the game are explained.

First, there are different card Values. Each card in blackjack has a specific value assigned to it. Number cards (2-10) retain their face value, while face cards (Jack, Queen, and King) are worth 10 points each. The Ace card can be counted as either 1 or 11 points. The Ace card will always be worth 11 points, unless the total sum of your cards exceeds 21 then the Ace card will be worth 1 point. If the Ace is worth 1 point, it will be called a ”usable” Ace.

After explaining the card values, the game can be played. At the beginning of a round, the player and the dealer are each dealt two cards. The player's cards are usually face-up, while the dealer receives one face-up card and one face-down card. Once the initial deal is complete, it is the player's turn to make decisions to improve their hand total. The player has several options available:

- **Hit:** The player can request an additional card from the dealer by choosing to ”hit.” This action increases the player's hand total.
- **Stand:** If the player is satisfied with their current hand, they can choose to ”stand.” This action indicates that the player does not want any more cards and wants to keep their current hand total.
- **Double Down:** In some variations of blackjack, the player may have the option to ”double down” after receiving their initial two cards. This action allows the player to double their initial bet and receive only one additional card.
- **Split:** If the player's initial two cards are of the same rank, they may choose to ”split” the cards into two separate hands. This action requires an additional bet equal to the initial bet and allows the player to play each hand independently.

Once the player completes their turn, it is the dealer's turn to play. The dealer follows a predetermined set of rules: if the dealer's hand total is 16 or less, they must take another card (“hit”). If their hand total is 17 or more, they must ”stand” and not take any more cards. However, in some variations, the dealer is required to hit on a ”soft 17” (a hand that contains an Ace counted as 11 points), while in others, the dealer may be required to stand on a soft 17. In this project the “soft 17” won't be taken into account so the dealer will just stand if he has a hand total of 17 or more.

After the dealer completes their turn, the hands are compared to determine the winner. The following outcomes are possible:

- Player Wins: The player wins if their hand total is closer to 21 than the dealer's, without exceeding 21. In this case, the player is paid out at a 1:1 ratio (or as specified by the table rules).
- Dealer Wins: The dealer wins if their hand total is closer to 21 than the player's, without exceeding 21. If the player's hand total exceeds 21, they automatically lose. In this case the player isn't paid out anything and lost his or her money.
- Push/Tie: If the player and the dealer have the same hand total, it results in a tie or ”push.” In this case, the player's bet is returned without any additional winnings or losses.

2.2 Concepts of reinforcement learning

Reinforcement learning is a subfield of machine learning that focuses on training agents to make sequential decisions in dynamic environments. It is inspired by the way humans learn through trial and error, where the agent interacts with an environment, takes actions, receives feedback, and adjusts its behavior accordingly. Before diving in the implementation of the agent, the environment, etc. in a later section, these concepts will be explained.
The agent is the learner or decision-maker that interacts with the environment. It is typically represented as a software program or an algorithm. The goal of the agent is to maximize a cumulative measure of performance, known as the ”reward,” by selecting appropriate actions in different states of the environment.

On the other hand, the environment is the external context in which the agent operates usually modeled as a Markov Decision Process (MDP). It can be a physical system, a virtual simulation, or any problem domain that the agent interacts with. The environment provides feedback to the agent in response to its actions and determines the state transitions.

The actions represent the decisions made by the agent to interact with the environment. At each time step, the agent selects an action from a set of available actions (the action space) based on its current state and the action policy. The action can have immediate consequences, leading to a change in the state of the environment.

The policy is the strategy or rule that the agent employs to determine its actions based on the current state. It maps states

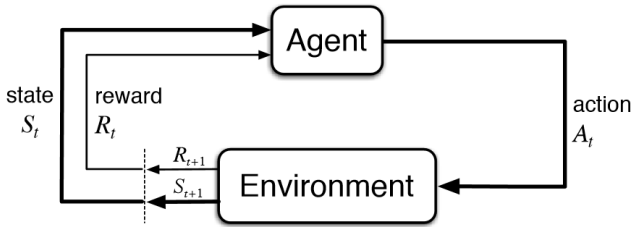

Figure 1. Concepts of reinforcement learning [1]

to actions and guides the agent's decision-making process. Policies can be deterministic, where each state is associated with a single action, or stochastic, where the policy defines a probability distribution over actions for each state.

After the agent takes an action, there will be a feedback signal provided by the environment to the agent, namely the rewards. The rewards indicate the desirability or quality of the agent's actions. The objective of the agent is to maximize the cumulative reward over time. Rewards can be positive, negative, or zero, depending on the outcome of the action taken by the agent.

The idea is for the agent to learn an optimal policy that maximizes the expected cumulative reward over its lifetime. This is achieved through a process of trial and error, where the agent explores different actions and observes their consequences in the environment. By evaluating the received rewards and updating its policy, the agent gradually improves its decision-making abilities and learns to make more effective choices in different states of the environment. The whole feedback and learning process can be visualised using figure 1.

2.3 Different types of reinforcement learning algorithms

Reinforcement learning offers a range of algorithms commonly used in game playing. Each algorithm has its own unique characteristics, advantages, and disadvantages. By exploring these different algorithms, we can assess their suitability for implementing the agent in our game. Considering their explanations and evaluating their pros and cons, we can ultimately select the most appropriate algorithm to meet our requirements.

Before diving into the different algorithms it is important to know that reinforcement learning algorithms can be categorized into two main types namely, model-based and model-free algorithms. Model-based algorithms aim to learn an explicit model of the environment and use it for decision-making. They offer efficiency and insights into the environment dynamics but require accurate modeling. Model-free algorithms, on the other hand, directly learn policies or value functions through trial and error. They are simpler to implement, handle unknown dynamics, but may require more interactions with the environment. [2]

In this paper a model-free algorithm is preferred. This way the agent learns optimal policies or value functions through trial and error without explicitly modeling the environment. This approach is well-suited because of the well-defined rules of blackjack. On the other side it is also easier to implement, then inventing a blackjack model.

Using this information, three, model-free, algorithms will be compared: Q-learning, Monte Carlo methods, and Temporal Difference (TD) learning. Each algorithm has its own advantages and disadvantages.

The first algorithm is Q-learning, a value-based algorithm. Its advantage lies in its simplicity and suitability for problems with discrete state and action spaces. Q-learning iteratively updates the Q-values enabling efficient exploration. It aims to learn the optimal action-value function, known as the Q-function, which maps state-action pairs to their expected cumulative rewards. Q-learning uses an iterative process to update the Q-values based on the observed rewards and the maximum Q-value of the next state. The update equation is:

$$Q(s,a) \longleftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)] \quad (1)$$

with:

- $Q(s,a)$ is the Q-value of the state-action pair (s,a);
- r the immediate reward;
- α the learning rate;
- γ the discount factor;
- s' the next state;
- a' the possible actions in the next state s'.

The action selection in Q-learning follows an ε-greedy policy, where the agent chooses the action with the highest Q-value with a probability of (1 - ε) and selects a random action with a probability of epsilon. This epsilon-greedy approach ensures a balance between exploitation (selecting the best action according to current Q-values) and exploration (selecting random actions to gather more information). [3]

Next to Q-learning we have Monte Carlo methods, they estimate value functions by averaging observed returns across episodes. They handle stochastic environments well and provide unbiased value estimates. The value function is updated by averaging the observed returns for each state.

However, Monte Carlo methods can be computationally expensive as they require complete episodes and may have high variance in value estimation. They estimate the value function using the equation:

$$V(s) \longleftarrow V(s) + \alpha(G - V(s)) \tag{2}$$

with

- V(s) the value estimate of the state s;
- α the learning rate;
- G the observed return.

Lastly, the Temporal Difference (TD) learning algorithm. TD learning strikes a balance between Q-learning and Monte Carlo methods but can be slower to converge and have higher variance in value estimates. It uses bootstrapping to update value estimates based on subsequent time steps. It strikes a balance between Q-learning and Monte Carlo methods, but can be slower to converge and have higher variance in value estimates. The update equation is:

$$V(s) \longleftarrow V(s) + \alpha(r + \gamma V(s') - V(s)) \tag{3}$$

with:

- V(s) the value estimate of state s;
- α the learning rate;
- r the immediate reward;
- γ the discount factor;
- s' the next state.

In conclusion, Q-learning is selected as the algorithm of choice due to its simplicity, efficiency, online learning capability, and compatibility with the discrete nature of blackjack. Its ability to learn and update policies incrementally, make Q-learning the ideal algorithm for developing an effective and optimal blackjack agent.

3. Designing the Blackjack Agent

In this section, we will focus on designing the blackjack agent, which involves defining the representation of the blackjack environment and determining the available actions for the agent.

3.1 Implementing a Blackjack environment

To design an agent that can play Blackjack, it needs to be able to play the game. Rather than implementing a Blackjack environment ourselves, an existing environment has been used. The environment is called "Gym" and is developed by OpenAI. It is a python library that contains many game environments, including Blackjack. The environment has the following assumptions:

- The game is played with an infinite deck of cards.
- The player does not have two individual cards, but a sum of them. This greatly reduces the state space.

When the environment is initialized, it returns a "state" tuple containing the sum of the players cards, the dealers up card and a boolean indicating if the player has a usable ace. The agent can interact with the environment by hitting or sticking, which will, besides the updated state tuple, return a boolean indicating if the game has ended or not and the reward for the last action (+1 if the last action won the game, -1 if the last action lost the game and 0 in all other cases). When the game has ended, the environment can be manually reset to play a new round.
The state-space consists of all the combinations of the sum of the players cards and the dealers face-up card. Since the sum of the players cards stays between 4 and 21, and the dealers up-card can be between 1 (ace) and 10. Finally, the player having a usable ace or not is described by a boolean, meaning the state-space is of total size 360. The action-space consists of sticking or hitting, and is thus of size 2.

3.2 Implementing the agent

The agent will be trained using the Q-learning algorithm with an epsilon-greedy exploration strategy. This algorithm aims to calculate the quality, known as the Q-value, of different state-action combinations. The Q-values are stored in a Q-table, which represents the agent's knowledge of the environment. The implementation of the Q-learning algorithm with epsilon-greedy strategy can be summarized as follows:

1. Initialize the Q-table ($Q(s,a)$) with arbitrary values.
2. For a certain number of episodes:
   (a) Choose an action a while in state s based on an epsilon-greedy policy derived from the Q-table. With probability ε, select a random action, and with probability $1-\varepsilon$, select the action with the highest Q-value.
   (b) Take the chosen action a, observe the resulting new state s' and the associated reward r.
   (c) Update the Q-value of the state-action pair using the Q-learning equation:

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

Here, ε represents the exploration rate, which determines the probability of taking a random action. A higher epsilon encourages more exploration, allowing the agent to discover new states and potentially better actions. The epsilon greediness is

thus a balance between exploration and exploitation.

By iteratively updating the Q-values based on observed rewards, the agent gradually learns the optimal actions to take in each state. This learning process allows the agent to develop a policy that maximizes its cumulative rewards over time.

During the implementation of Q-learning, the selection of appropriate reward functions is crucial. These reward functions serve as a feedback mechanism for the agent, assigning positive or negative values based on the outcomes of its actions. Well-designed reward functions provide incentives for the agent to make desirable moves, in this case winning, while discouraging unfavorable choices, losing or burning his hand.

The selection of appropriate reward functions significantly shapes the agent's learning process and directly influences its decision-making capabilities. If the reward function is not properly designed, the agent may fail to learn the desired behavior or exhibit suboptimal performance.

After selecting an appropriate reward function, it is common to tune hyperparameters to optimize the learning process and obtain the best possible model. Hyperparameters, such as the learning rate and discount factor, affect the rate of convergence and the trade-off between immediate rewards and long-term goals. Fine-tuning these hyperparameters can improve the efficiency and effectiveness of the Q-learning algorithm.

4. Training the Blackjack Agent

The training process for the Blackjack agent involves filling the Q-table, which serves as the basis for the agent's decision-making during gameplay. Initially, the Q-table is filled with '0'. The agent then interacts with the environment over a large number of episodes, following the Q-learning algorithm. The training process proceeds as follows:

In each episode, while the game is ongoing:

1. The agent selects an action based on an exploration-exploitation trade-off. A random number is generated, and if it is lower than a predefined exploration rate (ε), the agent chooses a random action. Otherwise, it selects the action with the highest Q-value for the current state.
2. The agent takes the selected action in the environment and observes the resulting new state, reward, and whether the game is finished or not.
3. The Q-value of the chosen action is updated using the Q-learning update equation (see equation 1), incorporating the reward and the maximum Q-value among possible actions in the next state.
4. The current state is updated to the new state, and the process continues until the game is done.

After that this learning process is finished, the parameter that is used to determine if it was a good strategy or not is the win rate of the agent playing the game blackjack. During this process 100.000 games are played. The actions of the agent are based on the Q-table calculated in the learning phase. The amount the agent wins, loses, or has a draw is saved. At the end, the total win rate for that agent is calculated using the following formula:

$$winrate = \frac{\#wins}{\#played\_games - \#draws} \tag{4}$$

So the rewards r and the different hyperparameters used in equation 1 can be reviewed and even optimized. This is what will be done in the following paragraphs.

4.1 Reward Function

The training process for the Blackjack agent involved a sequential approach to achieve optimal performance. Initially, a set of hyperparameters was chosen, determining the aspects of the agent's learning and decision-making process. With the hyperparameters in place, the focus shifted to choosing the reward function, which played a crucial role in guiding the agent's learning trajectory. By fine-tuning the rewards, desirable outcomes were incentivized, while discouraging undesirable actions. This fine-tuning was done using a grid-search. The reward that the agent got for a draw was kept at a constant 0. While draws are not desirable, they also aren't inherently undesirable as they can be strategically advantageous in certain losing scenarios. By assigning a neutral reward for draws, the agent was encouraged to consider the option strategically, potentially improving its overall decision-making capabilities and adaptive gameplay.

The rewards for winning ranged from 0.2 to 2.0 in steps of 0.1. The rewards given to the agent for losing the game ranges from -1.5 to -0.3 in steps of 0.1. The grid search was done using the following hyperparameters: $\alpha = 0.05$; $\gamma = 0.99$; $\varepsilon = 0.1$. After the Q-tables were made, each agent was tested by playing 10.000 games. The outcome of this grid-search can be found on figure 2. The size of the win rates is shown using the colorbar.
The rewardparameters that obtained the highest win rate of 44.87% are:

1. Reward to win a game: 1.2
2. Reward to lose a game: -0.8
3. Reward in case of a draw: 0

A general pattern can be noticed. For lower win and loss rates the overall win rate is rather small. This can be the consequence of the agent becoming overly risk-averse and prioritize avoiding losses over pursuing wins. This cautious approach can lead to conservative decision-making, where a draw is almost as good as a win. The agent may opt for safer

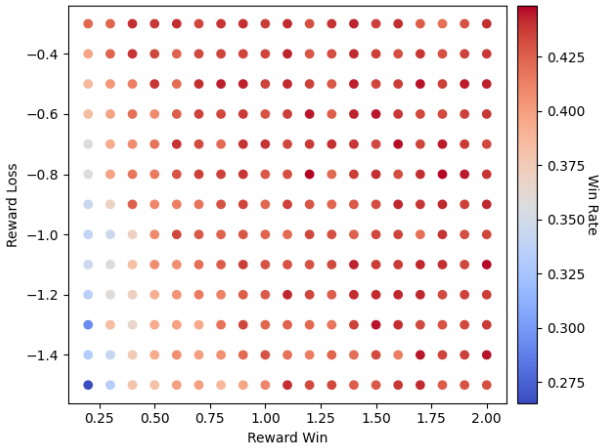

Figure 2. Test reward function parameters $\alpha = 0.05$; $\gamma = 0.99$; $\varepsilon = 0.1$

actions that minimize the chances of losing but potentially limit opportunities for achieving higher win rates. At low loss rewards, this trend is visible until the win reward is around one or bigger. So it can be assumed that the agent may not be incentivized to actively seek for winning opportunities but rather on avoiding losses, which can lead to a lower win rate overall.

It is interesting to observe that the overall win rate is generally not affected by the magnitude of the loss reward when the win reward is set to 1 or bigger. This finding suggests that the agent's learning and decision-making process may be more sensitive to positive reinforcement (win rewards) than negative reinforcement (loss rewards).
When the win reward is substantial, the agent is motivated to maximize its opportunities for winning. This positive reinforcement will encourage the agent to take actions that lead to favorable outcomes, rather than simply avoiding losing. Consequently, the agent will not nearly be as satisfied with a draw as with a win. The overall win rate thus tends to be influenced primarily by the rewards associated with winning, rather than the rewards associated with losing.

However, this is just based on figure 2, and of course one should keep in mind that a reward value of 0 is assigned in case of a draw, so the dealer and the agent have the same number and not higher than 21. By assigning a reward of 0 for draws, the agent is encouraged to consider drawing as a preferable outcome when facing potential losses. This can lead the agent to prioritize drawing rather than taking actions that may result in an outright loss (such as sticking with a low hand, or hitting with a strong hand). As a result, the agent may exhibit a more risk-averse strategy, aiming to secure a draw rather than taking the riskier path of potentially losing.

To provide a comprehensive comparison between different reward functions, a total of 1023 average win rates (over 100 000 simulated games each) are calculated using the Q-tables generated with a fixed set of hyperparameters ($\alpha = 0.05$; $\gamma = 0.99$; $\varepsilon = 0.1$). This number of simulations was chosen to ensure statistically significant results and to minimize the impact of random fluctuations.

In figure 3, the outcomes of these simulations of the two reward functions are plotted. The first reward function assigned rewards for winning and losing in an inverse manner, namely a reward of 1 for winning and a reward of -1 for losing. The second reward function was the one based on the previous computations, so 1.2 for winning and -0.8 for losing.

This analysis revealed that the average win rate achieved using the first reward function was lower than that obtained with the computation of the second reward function, the average win rates were respectively 42.43 % and 44.85 %. Next to the average win rate, the variances of the two reward functions were assessed. The variances for the first and second reward function were respectively 2.788e-6 and 2.7727e-6. These were calculated using the following formula [4]:

$$\begin{aligned} S^2 &= \frac{n}{n-1}\tilde{S}_Y^2 \\ &= \frac{n}{n-1}\left[\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \tilde{Y}\right)^2\right] \\ &= \frac{1}{n-1}\sum_{i=1}^{n}\left(Y_i - \tilde{Y}\right)^2 \end{aligned} \tag{5}$$

In this equation, n is the sample size (in this case 1023), Y_i is the $i-th$ item of the sample and $\tilde{Y}$ is the sample mean. This formula calculates the sample variance S^2, based on the differences between each observation Y_i and the mean $\tilde{Y}$ of the observations. The summation term represents the sum of the squared differences.

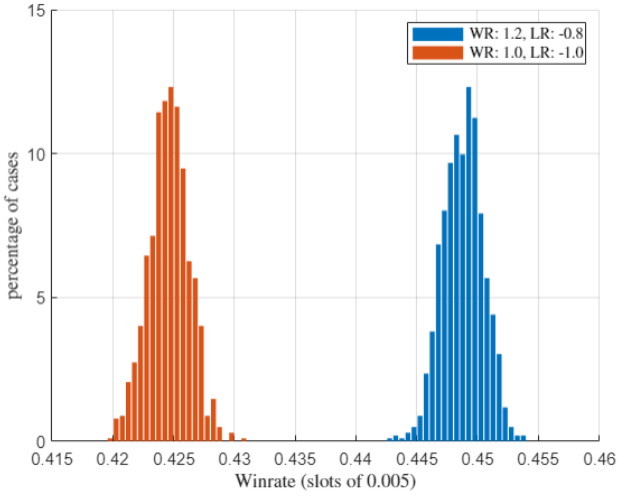

Figure 3. Distributions of average win rates (100k games) for different reward functions $\alpha = 0.05$; $\gamma = 0.99$; $\varepsilon = 0.1$

These variances are rather low, which suggests that 100 000 games are enough to compute a reliable average win rate. The similarity in variances, on the other hand, implies that the effects of the reward functions on the performance were consistent during the simulation runs.

Remark During this analysis a grid search was used to optimize the agent. Afterwards, we noticed that while a grid search can be effective for optimizing hyperparameters, it shouldn't be used to calculate the reward function. The reward function is typically best chosen based on domain knowledge or predefined criteria, rather than using a grid search. The previously obtained reward functions are thus optimal for the hyperparameters chosen at the time, but are not necessarily "the best" in general. Intuitively, the rewards +1 and -1 for a win and loss respectively should have been kept. With this reward function, the model is trained knowing a win generates the same reward a loss discards (like in real life), optimizing its decisions to maximize rewards in the long run.

Since this realization happened too late, we did not have time to compute everything for the better reward function (winreward of 1 and losereward of -1). A simple grid search for its optimal hyperparameters was done by chance at the start of the project, but there are no plots for it. This and the following section will describe the work we did with the sub-optimal reward function, as we would have done with the optimal one had we discovered it earlier. In the next section, "evaluating the Blackjack Agent" the performance of the agent will be compared to one with the better reward function and his respective optimal hyperparameters. This way, our most performant version of the agent will still be shown.

4.2 Hyperparameters

After obtaining the rewards calculated in the previous section (win reward of 1.2, loss reward of -0.8, and draw reward of 0), the hyperparameters were refined using a grid search. The grid search technique allowed to explore different combinations of hyperparameter values and evaluate the agent's performance for each configuration. A range of values for each hyperparameter was defined, such as learning rate (α), discount factor (γ), and exploration rate (ε). For each set of hyperparameters, the learning algorithm was executed for 100 000 games. Then using the different Q-tables, one for every set of hyperparameters, the win rate was calculated by letting the agent play 100 000 games without learning. The win rates are graphically shown on figure 4. The grid search allowed to systematically explore the hyperparameter space, so that the agents learning and decision-making process could be fine-tuned. Through this approach, the hyperparameter values that maximized the agent's performance in the game of Blackjack could be identified.

The optimal alpha was found to be 0.01, gamma was 0.5, and epsilon was 0.25. These values were associated with the highest win rate.

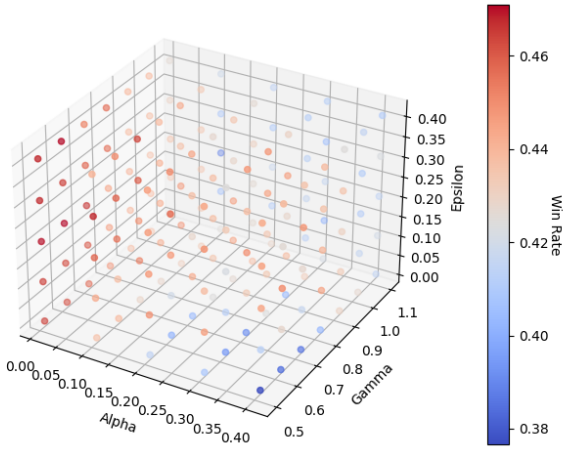

Figure 4. Grid-search hyperparameters

Some trends can be found in these results. A first impression shows that a higher learning rate (α) resulted in significantly lower win rates. This suggests that a higher learning rate might lead to sub-optimal learning and decision-making by the agent, because the impact of a single win or loss on the Q-value is too large. While this may potentially expedite the learning process, it can also introduce instability. The agent may become more reactive to immediate rewards without adequately considering long-term strategies, ultimately leading to lower win rates. Conversely, a lower learning rate allows for more gradual updates to the Q-values, reducing the impact of fluctuations and providing a more thoughtful and cautious learning process. The negative side is of course that more learning episodes are required before achieving a steady outcome of the agent. A balance between performance and time needed for the convergence is of essence.

Secondly, a relatively higher exploration factor (epsilon) was found to be associated with higher win rates in general. This means that when the agent has a greater tendency to explore new actions rather than exploiting its current knowledge, it tends to achieve higher win rates. By allowing more exploration, the agent can discover potentially advantageous strategies and actions that it might not have tried yet, or try out and discard bad state-action pairs. This flexibility in exploring different options can lead to higher win rates as the agent learns from its experiences and refines its decision-making process. However, it's important to note that there is a limit to the benefits of increased exploration. When epsilon becomes

too high, the agent may excessively prioritize exploration over exploitation, resulting in slower convergence. Therefore, finding an appropriate balance is crucial to ensure the highest possible win rate.

After an initial grid-search with a "roughly" grained grid, a second, more in-depth grid-search was performed. This time, narrower ranges were used. Interestingly, the optimal values obtained from the initial grid search remained the same in this more detailed search, indicating their robustness and effectiveness in maximizing the win rate. However, it is important to note that lowering the learning rate (alpha) significantly may impact the convergence speed and accuracy of the Q-table. A lower learning rate implies slower updates to the Q-values, which may require more episodes to properly converge. Therefore, it is essential to consider the convergence of the Q-table when adjusting the learning rate. This is also the reason why the learning rate never went below 0.01. This value was chosen as minimum value for the learning rate, otherwise the time needed to let the results converge was too high.

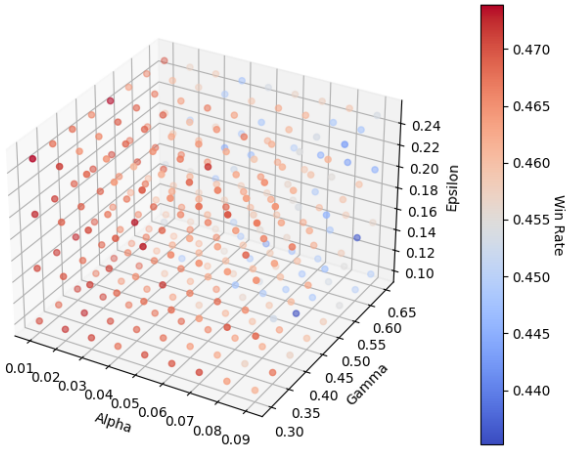

Figure 5. Fine-grained grid search hyperparameters

The optimal hyperparameters, for rewards of 1.2 and -0.8, obtain a winrate of 47.39% and have the following values:

1. α: the learning rate: 0.01
2. γ: the discount factor: 0.5
3. ε: the greediness factor: 0.25

4.3 Convergence

In the following paragraph, we will examine the convergence of the algorithm to assess whether the chosen number of episodes allow for the appropriate calculation of the Q-table.

To evaluate the convergence, a series of simulations was conducted with varying numbers of episodes. The hyperparameters used in these simulations were alpha = 0.01, gamma = 0.5, and epsilon = 0.25, while the reward function consisted of a draw reward of 0, a win reward of 1.2, and a loss reward of -0.8.

The number of games that was played to determine the win rate with the different Q-tables was kept constant at 100.000 games. To assess convergence more effectively, a sliding window approach was utilized. Average win rates were calculated using a window size of 13 adjacent points. This smoothing technique provided a clearer indication of the algorithm's convergence.

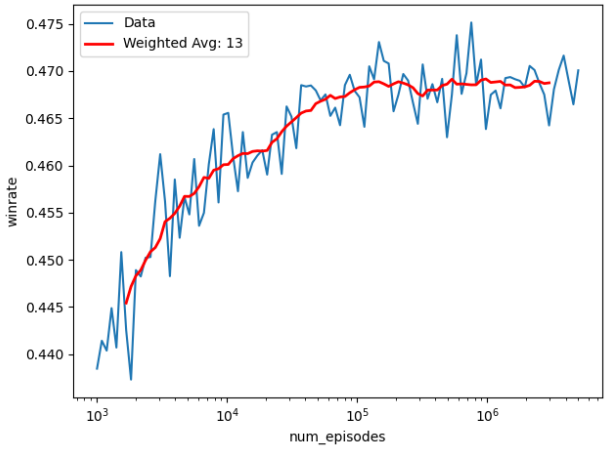

Figure 6. Q-table convergence plot

As can bee seen on figure 6, the win rate stabilizes around 100 000 episodes. While technically the red curve (average) at 100 000 episodes is very slightly lower than the part of the curve to its right, being an average it is affected by the lower win rates to its left. This figure shows as well that if we would have taken a lower learning rate (lower than 0.01) during the analysis of the hyperparameters, the constant number of 100.000 episodes wasn't enough anymore to let the algorithm converge properly.

5. Evaluating the Blackjack Agent

In this section, we will compare the performance of our agent to a few other strategies commonly used in blackjack. To conduct the comparison, 1023 sets of 100,000 games were played for each policy, and the average win rates were calculated for each set. The win rates were then categorized into bins of 0.05% to visualize the distribution of average win rates for each policy. The results are presented in figure 7 and table 1, where the blue bars represent the win rates of the agent using the reward function with a win value of +1.2 and a loss value of -0.8, while the orange bars represent the win rates of the agent using the reward function with a win value of +1.0 and a loss value of -1.0. Both agents were trained using their own optimal set of hyperparameters.

As mentioned earlier, the best performance was achieved when using the reward function with win: +1/-1, in combination with the following hyperparameters:

1. Learning rate α: 0.01
2. Discounting factor γ: 0.2
3. Greedy parameter ε: 0.41

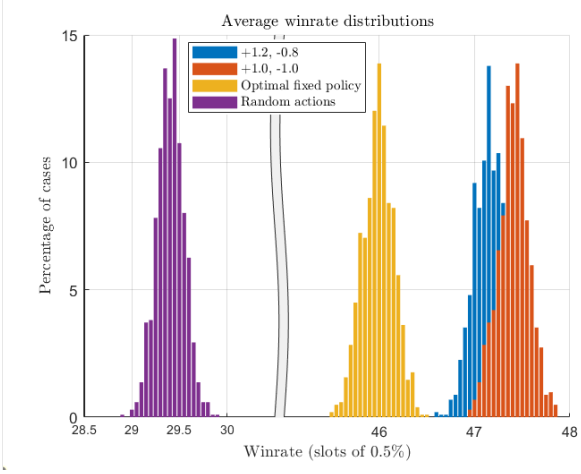

Figure 7. Performance comparison

Strategy	Sample mean	Sample variance
WR +1.2 & LR -0.8	47.21%	2.83e-6
WR +1.0 & LR -1.0	47.46%	2.63e-6
Fixed policy	46.04%	2.72e-6
Random	29.46%	2.05e-6

Table 1. Performance Comparison of Different Blackjack Strategies

Figure 7 shows taking random actions is (obviously) a very bad strategy. However, having a fixed strategy (sticking as from 16) based solely on the players cards makes sense. In most cases, it is hard to evaluate the impact of the dealers up-card on the chances of winning of the player, thus inciting to make choices only based on the players cards. Despite this, both agent versions are more performant than the highest performing fixed strategy. This demonstrates the agent's ability to effectively utilize the information provided by the dealer's up-card, which is indeed remarkable.

The comparison presented in this section provides valuable insights into the performance of our agent relative to other common strategies employed in blackjack. It underscores the agent's superiority in exploiting the available information and making strategic decisions that lead to higher win rates than normal standard strategies.

6. Discussion and Future Work

In the discussion and future work section, it is important to analyze the strengths and weaknesses of the developed blackjack agent and explore potential improvements or extensions to its training or architecture.

6.1 Discussion

While no versions attained a win rate above 50% (long term positive rewards), the best implementation comes relatively close with an average win rate of 47.46%. If this algorithm was to be combined with Martingale's betting system[5], long term rewards can be achieved in most cases, even with a *relatively small* minimum-to-maximum bet range. If the maximum bet is at least 8 times the minimum bet, the probability of the algorithm having a long term (up to 4 games) positive reward is slightly more than 85%. To raise this probability to 98%, the maximum bet must at least be 8 times the minimum bet, which is totally possible at some low-stakes casino's that have a betting range from $1 to up to $200 dollars. And nothing forbids the player to change from a low-stakes to mid- or high-stakes table if the remaining 2% event happened. Achieving a significantly higher average win rate is not possible using this environment. The assumptions such as an infinite deck and the player having a sum of cards rather than two individual cards limit the knowledge of the agent and thus the information at its disposal to make choices. Using a custom made environment which is more close to reality could allow for increased win rates, but would also require drastically more computing power.

6.2 Future work

First, it is important to acknowledge that the specific rule regarding blackjack, where a player wins at a higher payout rate if their initial two cards form a total of 21 and the dealer does not have a blackjack, was not implemented in the current version of the reinforcement learning program for playing blackjack. This rule typically results in a 3:2 payout ratio for the player's blackjack hand.

Due to time constraints, we were unable to incorporate this specific rule into the program and obtain updated values reflecting its impact. Re-running the entire program to include this rule would have required a significant amount of time and resources.

However, it should be noted that adapting the program to include this rule is relatively straightforward. The rule can be easily integrated into the reward system of the reinforcement learning algorithm. By assigning a higher reward for achieving a blackjack, the program can be trained again to recognize and exploit this advantageous situation.

Therefore, as a recommendation for future work, we suggest modifying the reinforcement learning program to account for the blackjack rule. This adaptation is likely to lead to improved performance and a more accurate representation of the game of blackjack. It is possible that incorporating this reward will result in a positive outcome in terms of monetary gains, but the overall win rate is still expected to hover around 50% given the random nature of the game.

Taking this rule into consideration would enhance the agent's ability to leverage the strategic advantage of a blackjack hand, leading to potentially better results and a more realistic gameplay experience.

Secondly, as stated previously, designing a more realistic blackjack environment that accurately reflects the rules and dynamics of the real-life game would enhance the training process and improve the agent's ability to utilize its learned strategies in real-world scenarios. Now the agent was trained using the sum of cards of his hand, while in real life the values of the two separate cards are known. This implementation would make the state space significantly bigger, drastically increasing computing time for all training sessions. However, this environment could also allow the agent to "count cards" and gain a further edge, probably increasing its average win rate above 50%.

Implementing Blackjack insurance could also improve the performance of the agent. This rule allows the player to bet on the dealer having a Blackjack when they have an ace in order to break even if it is the case. While mathematically, this bet should never be taken[6], it can be used in very specific scenarios when counting cards to reduce losses.

Another potential improvement is to use parallel computing techniques, or use the cluster of the *Royal Military Academy* to reduce the training time required to achieve an optimal performance. The two, already mentioned, improvements both have as a negative side that it would increase the training time. A reduction of training time by using better hardware, or using the hardware more efficiently would lead to a better final model.

7. Conclusion

In this project, a Q-learning agent was implemented to play blackjack, utilizing the OpenAI Gym library as the environment. The Q-learning algorithm was employed, known for its ability to learn optimal strategies through trial and error.

The rewards in the implementation were initially set as +1.2 for a win and -0.8 for a loss. This approach incentivizes the agent to seek wins and avoid losses, aligning with typical reinforcement learning tasks. Later, it was realized simply using +1 and -1 respectively would provide better results, while also being more logical since it follows actual game rewards.

The agent's most performant version, using the +1 and -1 rewards, incorporated specific hyperparameters: a learning rate (α) of 0.01, a discount factor (γ) of 0.2, and a greedy (exploration rate) ε of 0.41. These hyperparameters were determined by performing grid searches, and afterwards these parameters will determine the agent's learning and exploration process during the training phase.

The agent's performance was evaluated, and it achieved a win rate of 47.46%. This win rate indicates that the agent performed significantly better than random chance and slightly better than any fixed strategy, but has not yet reached a consistently winning strategy. This aligns with the initial expectations for a casino game where the house has an inherent advantage and has to make money. Further improvements could be made to enhance its decision-making process, but it would require a new environment and drastically more computing power or time.

References

[1] Pranav Mahajan. Playing blackjack using model-free reinforcement learning in google colab!, Jun 2021.

[2] Elisha Odemakinde. Model-based and model-free reinforcement learning: Pytennis case study, Apr 2023.

[3] sami shawl. Epsilon-greedy algorithm in reinforcement learning, Jan 2023.

[4] Wikipedia Contributors. Variance, May 2023.

[5] Wikipedia Contributors. Martingale (betting system), March 2023.

[6] How does blackjack insurance work?, Feb 2022.