Exploring Optimal Strategies for Blackjack Using Reinforcement Learning

Yernar Kubegenov, Frank Deinzer
Faculty of Computer Science - University of Applied Science Würzburg-Schweinfurt
Wurzburg, Germany
yernar.kubegenov@study.thws.de

***Abstract*—Blackjack is a popular most played casino game. The goal of the game is to keep card values less then 21 and have total sum of cards higher than dealers hand value. This study presents an approaches to uncovering optimal strategies for the card game Blackjack using a Reinforcement Learning (RL) framework. Using the fundamental simulated Blackjack environment as a test-bed, an implementation of Q-learning agent is trained to play numerous rounds of Blackjack and learn from its experiences. The results demonstrate that Reinforcement Learning has significant potential to discover effective strategies for complex decision-making tasks, such as playing Blackjack.**

***Index Terms*—Reinforcement Learning, Q-learning, Blackjack, Decision-making, Artificial Intelligence, Machine Learning**

I. Introduction

Blackjack is a popular card game that requires complex decision-making skills, making it an ideal subject for the application of Artificial Intelligence techniques. This study utilizes a form of Reinforcement Learning (RL) called Q-learning, where in an agent learns to choose actions that maximize its expected future rewards by interacting with its environment.

A. Black Jack Challenge

The game assigns point values to each card, with 2 through 10 being worth their face value, Jacks, Queens, and Kings worth 10 points, and Aces worth either 1 or 11 points. It is important to note the terms ”soft” and ”hard” hands. If a player’s Ace is counted as 1, they are holding a soft hand. Otherwise, they have a hard hand. Each player places a bet before receiving two cards face up, while the dealer receives one face down and one face up card. Players must decide whether to hit (receive another card), stand (end their turn), split (play two cards of the same value as separate hands), or double down (double their bet and receive one additional card).[1] Blackjack presents a challenge for machine learning algorithms due to its stochastic nature, with reinforcement learning algorithms being particularly well suited to approximate an optimal strategy. An optimal blackjack strategy should maximize financial return over the long run and cut the dealers per cent edge.

B. Reinforcement Learning

RL techniques use numerical reward signals as feedback to guide the agent in developing its policy. The environment is typically modeled as a Markov Decision Process (MDP), which is defined by a set of states, actions, transition probabilities, and expected rewards.[2] Each action has a probability of being selected and an associated value, which corresponds to the expected reward of taking the action.

This paper is structured as follows. The introduction section briefly describes the problem and introduces the basic rules of Blackjack and RL. In the Methodology section, It is present a simulated Blackjack environment and provide a short description of how the Q-learning algorithm interacts with. In the Results and Discussion section, the results are evaluated of the experiment in accordance with the agent’s behavior. Finally, the conclusion section provides a brief summary of the paper and presents questions for future research.

II. Methodology

The following section describes the methodology for the study, focusing on the environment used to facilitate interactions between the Q-Learning agent and the game of Blackjack.

A. Environment

I have developed a self-implemented Blackjack environment that serves as a learning playground for the Q-learning agent. This environment defines the game’s states, actions, and rewards, providing a simulation for the agent to interact with. The Blackjack environment is constructed to follow the traditional rules of the game and is encapsulated in the BlackjackEnvironment class. However, it is worth noting that the environment does not include a betting function in the scenario. Instead, it assigns rewards for each game based on the current strategy the agent is learning.

At the start of the game, a standard one deck of cards is used, including four suits: hearts, diamonds, clubs, and spades. Each suit contains the numbers 2 through 10 as well as the faces ’jack’, ’queen’, ’king’, and ’ace’. The deck is randomly shuffled at the beginning of the game and one single deck is played until it runs out of cards.

The BlackjackEnvironment class contains various methods to simulate the game of Blackjack and facilitate the agent’s interaction with it, such as ’draw_card’, ’hand_value’, ’dealer_play’, ’get_states’ and etc. The game also considering using a the Hi-Lo card counting system, specifically the complete-point card-counting system mentioned in book when it needed. [1] In this system, high cards (10 and ace) are assigned a negative count and low cards [2,3,4,5,6] a positive count. The total Hi-lo points are then divided by the total number of unseen cards in the deck. The final taken values is called by term ”low_high_index” similar as it mentioned in the book. This environment creates an arena where the Q-learning agent can interact with the game of Blackjack, take actions based on its learned policy, and receive rewards that reflect the success or failure of those actions, driving the learning process.

In the next subsection, we discuss the implementation of the Q-Learning agent and how it interacts with this environment.

B. *Q-Learning Agent and its Interaction with the Environment*

Q-learning was used, a type of RL, to learn how to play Blackjack. The agent uses a Q-table to keep track of the expected rewards for each possible action in each possible state. It uses this table to determine which action to take at any given state, following an epsilon-greedy policy. The Q-table is updated iteratively based on the rewards the agent receives for its actions. The core of our approach relies on a Q-table, a data structure used by the agent to keep track of the expected rewards for each action in each state. This table serves as the agent’s memory, guiding it towards actions that will maximize future rewards. The Q-table gets updated iteratively using the Q-learning update rule, which gradually converges to the optimal Q-values.

Mainly, the agent follows an epsilon-greedy policy for action selection, a balance of exploration and exploitation. Unless, there certain rules is depicted according to the Book[1]. To simplify the action-state variations agent can only choose action between ’hit’ and ’stand’. During the rule variations the action size could increase up to 4 including ’doubling down’ and ’pair split’. The study begins with little knowledge of the environment, mostly exploring by choosing random decision between ’hit’ and ’stand’ and improves its exploitation by continually interacting with it. For that purpose, the agent adheres to an epsilon value at highest value at the beginning, then reducing it over time by given decaying coefficient. See Figure 1 to get impression of how manipulation epsilon value at the beginning of the iteration look like.

According the ”Basic Strategy” rules the agent would need to take into account three following factors when deciding its actions:

1) The player’s current hand value
2) The dealer’s face-up card value
3) Whether the player’s hand is hard or soft (i.e., whether it contains an Ace that is being counted as 11)

That means only for ”Basic Strategy” Q-learning agent’s hand value can range from 4 to 21, resulting in 18 unique values, while the dealer’s up card can range from 2 to 11, resulting in 10 unique values. The ”soft” hand status can be True or False, resulting in 2 unique values. Also, considering two action variations, the total state-action space could reach up to 720 pairs. This is number of possible states agent has to investigate.

The core of our approach relies on a Q-table, a data structure used by the agent to keep track of the expected rewards for each action in each state. This table serves as the agent’s memory, guiding it towards actions that will maximize future rewards. The Q-table gets updated iteratively using the Q-learning update rule, which gradually converges to the optimal Q-values.

C. *Q-learning agent implementation*

The Q-Learning Agent, instantiated by the *QLearningAgent* class, uses a parameterized Q-Learning algorithm to interact with the Blackjack environment. The agent is initialized with various parameters including:

- $alpha$, the learning rate, which determines to what extent the newly acquired information overrides the old information.
-$gamma$, the discount factor, used to balance immediate and future reward.

-$epsilon$, the exploration rate, initially set to *initial epsilon* and will decay over time to *final epsilon* at a rate of *epsilon decay*. Epsilon is a parameter for epsilon-greedy policy, controlling the balance between exploration and exploitation.

num episodes, the total number of episodes for the agent to train on.

Furthermore, the agent maintains a Q-table, a dictionary mapping *(state, action)* pairs to Q-values, to guide its actions. To track the training progress, it keeps a list *training error* to store the temporal differences after each update, and *epsilon list* to record the value of epsilon over time.

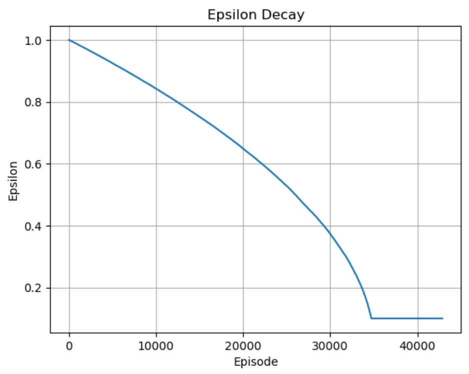
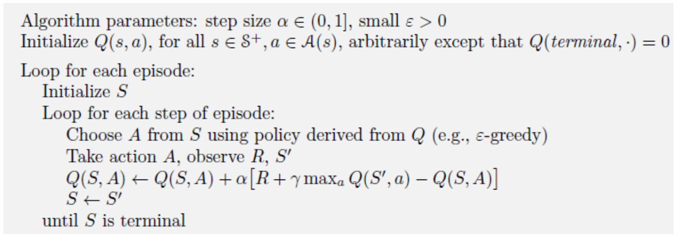


Fig. 1. The plot displays the epsilon decay from its initial value to its final value at a certain rate. This showcases 100,000 runs, where epsilon was decaying in the first 30,000 runs and only 10% exploration was carried out in the remaining 70,000 runs.

D. *Q-Table Update*

Once an action is selected and executed, the Q-table is updated based on the new state and reward. The update method implements the Q-Learning update rule. Specifically, for a given state-action pair, it computes the difference between the current Q-value and the newly observed reward plus the discounted future reward (estimated by the maximum Q-value in the next state). This difference, also known as the temporal difference, is then scaled by the learning rate *alpha* and added to the current Q-value to update the Q-table. The agent also logs this temporal difference in the *training error* list. The general procedural form can be shown in figure below.



```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Fig. 2. Q-learning (off-policy TD control). [2]

Over the course of specified episodes, the agent repeatedly interacts with the environment. For each episode, it first resets the environment and gets the initial state. Then it enters a loop where it chooses an action based on the current state, executes the action in the environment, observes the next state and reward, and updates the Q-table accordingly. This loop continues until the episode ends, marked by the done flag. After each episode, the agent decays its epsilon value to gradually shift from exploration towards exploitation as it gains more knowledge.

Overall, the Q-Learning agent learns by continually interacting with the Blackjack environment, updating its knowledge (Q-table), and refining its policy to make better decisions over time. In the next section, we will discuss how we evaluate the performance of the agent and the training process. Also, we check weather the agent could learn optimal polices mentioned in book strategy. [1]

III. Experiment

The experiment aims to evaluate the performance of the approach presented in the methodology section. Four agents interact with the environment under different scenarios. The first agent makes random decisions between 'hit' and 'stand' actions, trying to maximize reward without any prior knowledge. The rewarding rule is simple: the agent gets 1 point for winning, -1 point for losing, and no reward for a tie. The second agent, our main approach from the methodology, tries to balance exploration and exploitation based on the same rewarding system. The third agent imitates a player who follows the "Basic strategy" rules for drawing and standing from a book[1], with the same rewarding system. The last player agent investigates the "Complete-card counting" strategy from the same book[1]. Unlike the other agents, the rewarding system is adjusted here. Depending on the complete card count, the reward system gives extra reward or punishes if the card counting is smaller than depicted thresholds. At the beginning, the Random agent (first), Epsilon-Greedy (second), and Card-Counting (fourth) agents have the same random policy, but over time, each will start to converge to an optimal policy based on given scenarios. It is worth mentioning that the Epsilon-Greedy action is expected to converge to the same policy as Basic Strategy will learn.

IV. Results and Discussion

The table I shows the performance of each agent player evaluated by win, lose rate and tie. The number of runs stays the same for each agent N = 100,000 times. Because, the experiment has been shown that after depicted iterations the overall result will stay the same and oscillate around the optimal policy without ever converging.

TABLE I
Agent's accuracy for playing Black Jack using trained optimal policy for different scenarios for number of episodes 100,000

N = 100,000	Random Decision	Epsilon Greedy	Basic Strategy	Card Counting
Win rate	29%	40%	44%	43%
Lose rate	67%	52%	47%	49%
Tie	4%	8%	9%	8%

The most obvious conclusion that can be made is that all agents are losing the game, regardless of which rules they follow. This means that they are also losing their money. Even if a player is counting cards, it does not converge to a winning policy, although it could be helpful for betting strategy, which is not examined in this work.

The second clearest point is that RL agents are performing significantly better than the Random Player and have significantly better winning performance. Although the random player and Q-learning agent are exploring the game at the same rate, we can say that the agent has learned useful knowledge about how to play by the end.

From Figure 3, one can notice that Q-learning agents are exploring the states, and the average reward is increasing almost linearly over time towards the cumulative positive rewards. However, at the same time, the average reward rate is highly oscillating for the random player, which shows the inability to learn anything.

The Basic Strategy Follower player has the maximum reward value overall and also fluctuates around it, which overall has good stability knowledge of how to play the game. The Card Counting Q-learning graph shows its potential to learn more useful strategies further.

After analyzing the performance of the basic strategy player and the q-learning RL player, it was found that they did not intersect. This means that the learning agent did not converge to the optimal policy, but was rather to a near-optimal policy. It is important to note that this finding highlights the need to further explore and analyze the factors that led to the learning agent's inability to converge to the optimal policy. The figures below 4 and 5 show the learned suboptimal policy grid explicitly for hard and soft hands.

Lets brake some concepts and discuss the advantages and drawbacks of learning algorithms.

A. *Q-learning agent*

Q-Learning is a way for a computer program to learn how to make good choices. It does this by watching what happens when it picks different things to do, and then figuring out

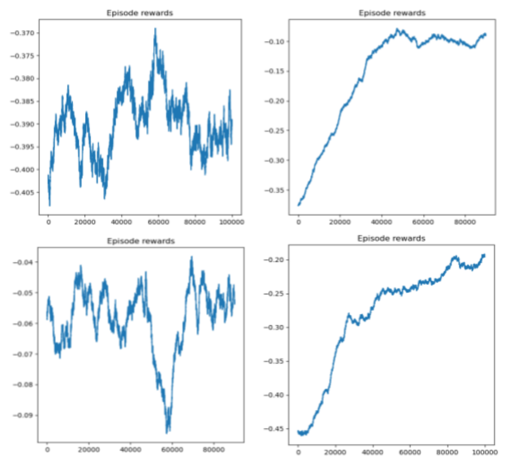


Fig. 3. Players' average reward per 10,000 games: top left is the random player, top right is the epsilon-greedy player, bottom left is the basic strategy player, and bottom right is the card counting player.

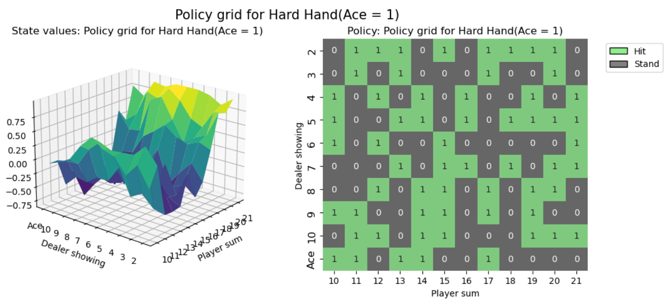


Fig. 4. Epsilon greedy optimal policy grid with hard hands. Agent is trying to find best strategy which aligns to "Basic strategy" scenario [1]. Number of episodes 100,000. Win rate of the policy in this scenario was 40 percentage on average.

which choices are better. This method is called "off-policy learning."

One advantage of Q-Learning is that it can work with different ways of rewarding good decisions. This can help encourage the program to find winning strategies.

However, Q-Learning also has some disadvantages; it can be slow to learn, especially in situations where there are many different choices to make, like in a complete card counting scenario. If the program only gets rewarded rarely, like in a game of Blackjack, it can take a long time to figure out which choices are good and which are bad. To help with this, the program needs to be careful to try different things without wasting too much time. Also, Q-Learning doesn't always work well in complicated situations, where understanding how things change from one moment to the next is important.

B. *Complete-card counting*

This subsection discusses the pros and cons of using card counting, a technique borrowed from book [1], to help make decisions in the game of Blackjack. Card counting has benefits, such as better decision-making and understanding the probability distribution used in games like Blackjack where only some information is available. However, this technique heavily relies on the agent's memory and ability to keep track of cards, which can make it too complex to use with Q-Learning. Additionally, incorporating it in a way that affects the rewards structure could be complicated. Choosing the appropriate threshold for awarding extra rewards and punishing the agent could be quite tricky to balance the reward strategy.

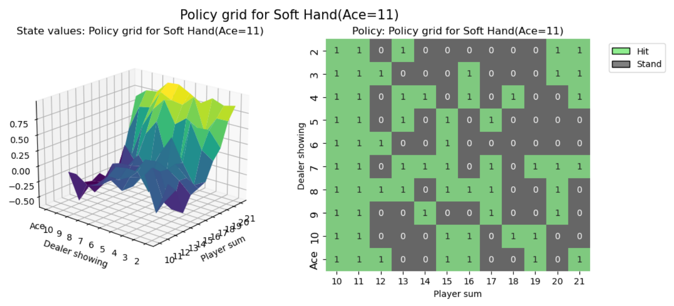


Fig. 5. Epsilon greedy optimal policy grid with soft hands. Agent is trying to find best strategy which aligns to "Basic strategy" as mentioned in book [1]. Number of episodes 100,000. Win rate of the policy in this scenario was around 40 percentage on average.

C. *State-action space*

The Blackjack Q-learning implementation's state-action space is determined by the considered states. The player's hand value can range from 4 to 21, resulting in 18 unique values, while the dealer's up card can range from 2 to 11, resulting in 10 unique values. The "soft" hand status can be True or False, resulting in 2 unique values. The high low index can theoretically range from -100 to 100, but this is a continuous variable, which is discretized into increments of 1 to keep the state space manageable, resulting in 201 unique values.

Taking into account the actions ('h' or 's'), the total state-action space reaches an impressive 144,360 possible state-action pairs. However, Q-learning assumes that every state-action pair is visited an infinite number of times to achieve stable Q-value estimates. In practice, this is often not feasible within a reasonable time-frame, particularly in environments with large state-action spaces. Therefore, the algorithm's effectiveness depends on its ability to generalize from the visited state-action pairs to those it has not encountered.

It is worth noting that the discretization of high low index into increments of 1 can be modified to achieve a more refined state space representation, which would increase the number of state-action pairs even further. Furthermore, a larger state space can lead to more accurate Q-value estimates but also requires more computational resources. Therefore, a trade-off between accuracy and computational complexity may need to be considered when implementing the algorithm in practice.

D. *Impact of rules changes*

In this Blackjack game, the policy is the strategy the learning agent uses to decide which actions to take based on its current state. The policy changes when the environment's rules change. For example, if a new rule is introduced where the dealer must stand on a 'soft 17' (a hand including an ace and totaling 7 or 17), the player's optimal policy would likely change. The agent's policy adapts to reflect these changes.

If the agent was trained under the standard rule where the dealer must hit on a 'soft 17', it may have learned a policy that takes more risks. But when the rule changes, the agent's policy adjusts, becoming less or more aggressive depending on the situation.

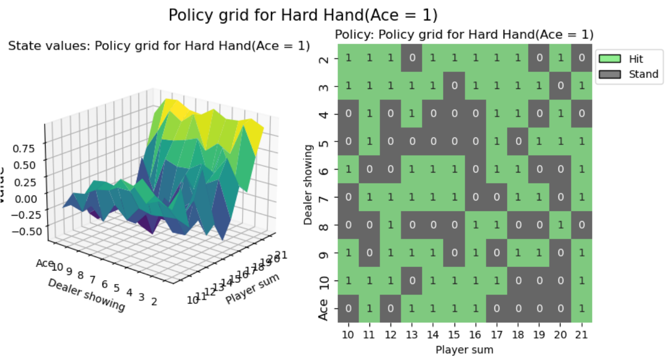


Fig. 6. Optimal policy grid under complete-point card counting scenarios for hard hands. The number of episodes is 100,000.

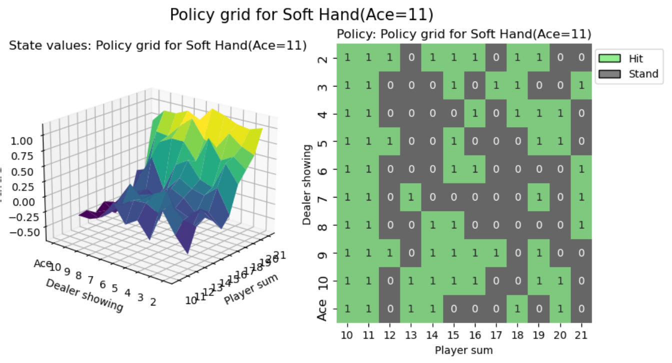


Fig. 7. Optimal policy grid under complete-point card counting scenarios for soft hands. The number of episodes is 100,000.

Another example of a change to the Blackjack environment is when the reward system was modified to be based on complete-point card counting. As a result, the potential reward for the agent increased, which could influence the agent's policy towards taking more risks in order to try and hit Blackjack, since the payout for this outcome is now significantly higher. This change can be seen in the newly obtained optimal policy grid, which is illustrated in Figures 6 and 7, as compared to the previous Figures 4 and 5. It is interesting to note how the change in the reward system led to a shift in the agent's behavior, as it sought to maximize its potential payout. These findings are important for understanding how changes to reward systems can affect an agent's behavior in complex decision-making environments.

In sum, any changes in the rules of the environment can potentially alter the Q-values and therefore the policy that the agent learns.

V. Conclusion

RL is learning method that has been shown to be effective in discovering optimal strategies in complex scenarios. This approach has been demonstrated by our Q-learning agent's performance in the Blackjack environment, which was able to learn and apply a winning strategy. However, while the results of this study are promising, there is still much to be done to beat the dealer. Specifically, it would be interesting to test the RL agent's behavior on specific rules, such as doubling-down, pair split used by world casinos to gain a winning edge in real-world scenarios. This will help discover the true potential of RL in solving real-world problems. In future, an exploration of further applications of more sophisticated RL methods to other complex decision-making tasks, such as developing a betting policy under different scenarios. RL is making great progress and can help society revolutionize how to approach complex decisions and solve important problems.

References

[1] Thorp, E. (1966). Beat the Dealer: A Winning Strategy for the Game of Twenty-One. Vintage Books.

[2] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[3] Brockman, Greg, et al. "Openai gym." arXiv preprint arXiv:1606.01540 (2016).

[4] Watkins, Christopher JCH, and Peter Dayan. "Q-learning." Machine learning 8.3-4 (1992): 279-292.