Reinforcement Learning for Blackjack Strategy

Placido Pellegriti

07/04/2025

Contents

1 Introduction

This report investigates reinforcement learning (RL) techniques to derive an optimal hit-stand policy for blackjack. We implement two RL approaches—a traditional Q-table and a Deep Q-Network (DQN)—and evaluate their performance both against each other and against two benchmarks: a random agent and a known optimal strategy. The analysis compares their efficiency (computational speed and convergence time) and policy quality (win rates and decision accuracy). Finally, we detail the technical implementation of each method and discuss their relative strengths and limitations. Some simple code examples, provided in Python, are included to illustrate key implementation steps.

1.1 Blackjack introduction and rules

Blackjack is a popular card game played between a dealer and one or more players. The goal of the game is to obtain a hand value as close as possible to 21 without exceeding it. The value of a hand is calculated as the sum of its card values, where:

- **Number cards (2-10)** are worth their face value.
- **Face cards (Jack, Queen, King)** are worth 10 points each.
- **Aces** can be worth either 1 or 11 points, depending on which value benefits the player most.

Each player is initially dealt two cards and can choose to:

- **Hit**: Take an additional card.
- **Stand**: Keep their current hand.
- **Double Down**: Double their bet and receive exactly one additional card.
- **Split**: If the first two cards are of the same rank, they can be split into two separate hands.

The dealer follows a fixed strategy, typically hitting until their hand value reaches at least 17.

1.1.1 Soft and Hard Hands

A hand in blackjack can be classified as either soft or hard:

- **Soft Hand**: A hand containing an Ace counted as 11. For example, an Ace and a 6 (denoted as (A,6)) is a soft 17.
- **Hard Hand**: A hand that either does not contain an Ace or contains an Ace that must be counted as 1 to avoid busting. For example, a (10,7) or an (A,6,10) (where the Ace must be 1) is a hard 17.

Soft hands provide more flexibility since the Ace can switch between 1 and 11, reducing the risk of busting when hitting.

1.2 Gymnasium

Gymnasium is a toolkit designed for developing and evaluating reinforcement learning algorithms. It provides a standardized interface for a wide range of environments, enabling researchers and practitioners to test different reinforcement learning methods under consistent conditions.

Besides allowing users to create custom environments, Gymnasium includes a variety of preexisting ones, such as classic control tasks, Atari games, and toy examples like the Blackjack environment, where the goal is to train an agent to make optimal hit or stand decisions.

The interaction with an environment follows a simple structure: the agent takes an action, the environment updates its state, and a reward is provided based on the effectiveness of the action. This process allows agents to learn policies in various ways, depending on the approach used.

1.3 Blackjack Environment

The "Blackjack" environment is part of the predefined Gymnasium environments and serves as a simplified simulation of the popular card game. The objective is for the player to obtain a hand value closer to 21 than the dealer's hand without exceeding 21. The game mechanics align with the version described in Example 5.1 of [2] by Sutton and Barto.

At the start of the game, the dealer has one face-up and one face-down card, while the player is dealt two face-up cards. Cards are drawn from an infinite deck, meaning that they are replaced after each draw. The player may continue drawing cards (hit) or choose to stop (stand). If the player's sum exceeds 21, they lose immediately (bust). Once the player stands, the dealer reveals their hidden card and continues drawing until they reach at least 17. The outcome of the game is determined by comparing sums, with the closest value to 21 winning.

1.3.1 Observation Space

The observation space in this environment is represented as a tuple containing three values:

- **The player's current hand sum**: ranging from 4 to 21.
- **The dealer's visible card value**: ranging from 1 to 10, where 1 represents an Ace.
- **A binary flag indicating whether the player has a usable Ace**: 1 if usable, 0 otherwise.

This compact representation will allow agents to make decisions based on key aspects of the game state.

An example state of the environment can be seen in Figure 1.

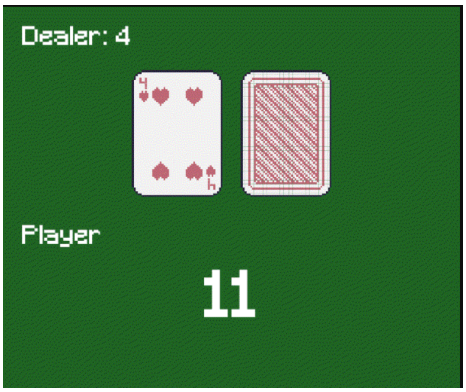

Figure 1: Example environment state: (11, 4, 0)

1.4 Action Space

The action space consists of two discrete actions:

- **Hit (1)**: The player requests an additional card.
- **Stand (0)**: The player stops drawing and finalizes their hand.

These actions define the primary decision-making process in blackjack, where the agent must balance risk and reward when choosing whether to draw more cards.

Notice that this simplified environment does not allow the user to split or double down, as opposed to the typical casino rules, but only to hit or stand.

1.4.1 Reward Structure

The reward system in the Blackjack environment follows these rules:

- **Winning** the game results in a reward of $+1$.
- **Losing** the game results in a reward of -1.
- A **draw** results in a reward of 0.

An episode ends when the player either goes bust, chooses to stand, or the dealer completes their turn. Aces are treated as 11 unless doing so would cause the player to exceed 21.

1.4.2 Environment Configuration

Throughout this paper, the Blackjack environment is initialized with the `sab=True` configuration:

```
env = gym.make("Blackjack-v1", sab=True)
```

This setting ensures that the environment follows the exact rules described in [2]. Specifically, if the player achieves a natural blackjack and the dealer does not, the player wins (the reverse rule does not apply).

2 Solutions adopted

To compare the performances of the various betting policies, several different agents have been created. The first two are rule-based agents, while the other two will try to learn an optimal hit-stay policy through reinforcement learning.

2.1 Random Agent

A random agent selects actions uniformly at random. While this strategy is inherently ineffective—as it disregards game state and logical decision-making—it serves as a useful baseline for comparing the performance of more sophisticated strategies in blackjack.

The following implementation defines a simple RandomBlackjackAgent class. It interacts with the environment by choosing actions (hit or stand) with equal probability, regardless of the current state:

```python
import numpy as np
import gym

class RandomBlackjackAgent:

    def __init__(self, env: gym.Env):
        self.env = env

    def get_action(self, obs: tuple[int, int, bool]) -> int:
        # 50% chance of hitting or staying
        return np.random.choice([0, 1])
```

2.2 Optimal Agent

The optimal basic strategy for Blackjack, including decisions about hitting, standing, splitting, and doubling down, was first mathematically derived in [1] using probability theory and expected value maximization. Although [1] addressed the standard game rules, the actions available in our simplified scenario differ. Thus, the strategy has been adapted to our action space (hit/stand only) and rule set (no splitting). The best *hit-stand* policy can therefore be transformed in the following conditions:

```python
        if player_total <= 11:
            return 1
```

```
3
4           if useable_ace:
5               if player_total <= 17:
6                   return 1
7               elif player_total == 18 and dealer_card >= 9:
8                   return 1
9
10          else:
11              if player_total <= 16 and dealer_card >= 7:
12                  return 1
13              elif player_total == 12 and dealer_card <= 3:
14                  return 1
15          return 0
```

This agent will be used to make comparisons between the reinforcement-learning agents and the best policy.

2.3 Tabular Q-learning

Q-learning is reinforcement learning algorithm that enables an agent to learn the optimal policy for decision making in an environment through trial and error. The primary goal of Q-learning is to find the best action to take in any given state to maximize the long-term cumulative reward, without requiring a model of the environment's dynamics. This makes it particularly useful in situations where the agent has no prior knowledge of the environment and must learn only from interaction. By using a process of exploration and exploitation, Q-learning gradually improves its decision-making strategy over time. One of the ways Q-Learning can be implemented is with a Q-Table, which helps the agent store and update the knowledge it gathers about the environment.

2.3.1 Q-Table

The Q-Table technique is a core component of Q-learning. The goal is to maximize the agent's cumulative reward over time by learning the best actions to take in each state. This method is especially useful for problems where the environment has a finite number of states and actions, and both action and observation space are discrete. The Q-table, a matrix that stores the expected rewards for state-action pairs, is updated as the agent learns through experience.

2.3.2 Temporal Difference Learning Mechanism

The Q-values are updated through temporal difference (TD) learning, which integrates immediate rewards with bootstrapped estimates of future returns. The update rule is defined as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \delta_t$$

where the TD error, δ_t, is given by:

$$\delta_t = r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$$

Key components:

- **TD Error (δ_t)**: This is the difference between the received reward plus the discounted maximum future Q-value and the current Q-value.
- **Bootstrapping**: The method uses the current Q-value estimates to approximate future rewards.
- **Learning Rate (α)**: This parameter controls how much new information should override the existing Q-values. A high α means that the agent adjusts its Q-values quickly in response to new experiences, making the learning process faster. However, if α is too high, the agent may "overreact" to recent rewards and fail to properly balance older knowledge with new information.
- **Discount Factor (γ)**: This factor scales future rewards, ensuring that immediate rewards have a greater influence on the current update.

The corresponding code snippet demonstrates this mechanism by computing the future Q-value only if the next state is non-terminal, calculating the temporal difference, and updating the Q-table accordingly:

```
future_q_value = (not terminated) * np.max(self.q_values[next_obs])
temporal_difference = (
    reward + self.discount_factor * future_q_value - self.q_values[obs][action]
)
self.q_values[obs][action] = (
    self.q_values[obs][action] + self.lr * temporal_difference
)
```

This incremental updating approach allows for efficient learning by immediately incorporating each new experience into the Q-table via the computed TD error.

2.3.3 Implementation

The Q-learning agent was implemented using a tabular approach with the following key design decisions. A *defaultdict* initialized with zero-filled NumPy arrays served as the Q-table, where each state tuple (player sum, dealer card, usable Ace) maps to an array of action values. The agent employed an ε-greedy exploration strategy with linear decay, starting at $\varepsilon = 1.0$ (complete randomness) and decaying to $\varepsilon = 0.01$ over 5 million episodes according to:

$$\varepsilon_{\text{decay}} = \frac{\varepsilon_{\text{initial}}}{n_{\text{episodes}}/2} = \frac{1.0}{5,000,000} = 2 \times 10^{-7}$$

Here, ε represents the exploration rate—the probability that the agent will choose a random action rather than the action with the highest estimated Q-value. This strategy balances exploration (trying new actions to discover their effects) and exploitation (leveraging known high-reward actions). The linear decay schedule ensures the agent explores extensively early in training while gradually shifting toward exploitation as it gains experience.

This maintains a baseline exploration rate to prevent premature convergence to suboptimal policies. The learning rate $\alpha = 0.001$ provided conservative Q-value updates while the discount factor $\gamma = 0.95$ emphasized near-term rewards.

During each of the 10 million training episodes, the agent:

1. Selected actions using ε-greedy exploration
2. Updated Q-values via temporal difference learning
3. Decayed ε linearly while maintaining a 0.1 minimum
4. Tracked temporal difference errors

The state space consisted of 360 discrete states (18 possible sums [4-21] $\times$ 10 dealer cards $\times$ 2 Ace states), enabling comprehensive exploration within the 10 million episode budget. Figure 2 illustrates the convergence pattern through temporal difference errors, where each point represents a moving average of 50,000 consecutive training steps. The sustained plateau in TD error after approximately 1 million episodes indicates policy stabilization, demonstrating that the agent required less than half of the total allocated episodes to converge to its final strategy.

2.3.4 Convergence recognition

The convergence of the Q-table training was assessed by analyzing the agent's training error using a moving average approach. Specifically, the training error was first smoothed using a convolution with a window of 50,000 steps to mitigate short-term fluctuations. Subsequently, rolling statistics (mean and standard deviation) were computed over a window of 100 smoothed averages, which corresponds to 5 million training steps per window. Convergence was identified when the standard deviation of the rolling window fell below a threshold of 0.0001 and the absolute value of the rolling mean was less than 0.001, indicating that the training error was both stable and close to zero. Moreover, the process required these conditions to be met for at least 50 consecutive windows (approximately 2.5 million steps) to ensure that the convergence was sustained rather than transient.

2.3.5 Reward Evolution

Although the model appears to converge early in training, the reward only stabilizes after approximately 5 million episodes. This delay is primarily due to the epsilon decay schedule: the epsilon gradually decreases to a fixed value

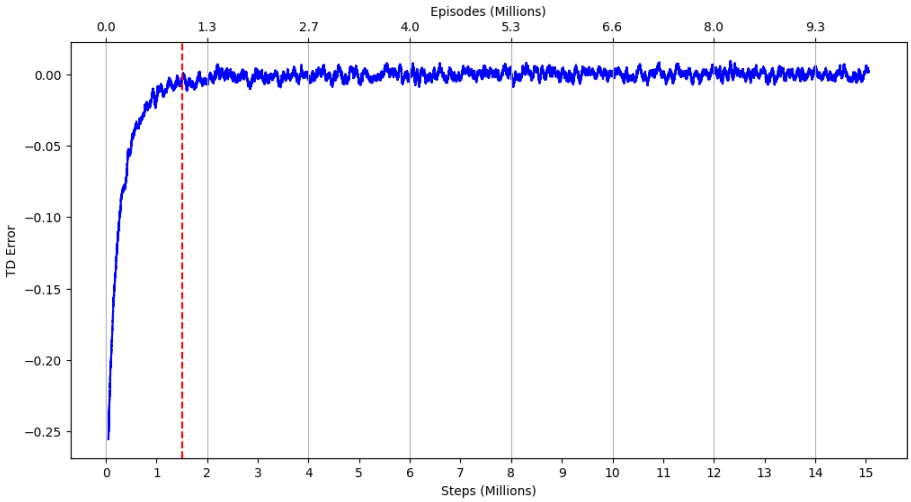

Figure 2: Q-Table TD error evolution over training

of 0.01 by the fifth million episodes and remains constant thereafter. As a result, the randomness in the exploration diminishes significantly at this point, leading to more consistent policy updates and reward stabilization. This effect is visually evident in Figure 3, where the reward curve exhibits greater stability once epsilon reaches its minimum value.

2.3.6 Results

The Q-Table based approach, which predictably fits well to the Blackjack environment due to the discrete action and observation spaces, has come up with a policy for deciding when to hit and when to stand. The policy can be summarized in the following rules:

- Hit if the player has less than 12.
- If the player has an Ace:
  - Hit if the player has less than 18.
  - Hit if the player has exactly 18 and the dealer has 9 or 10.
- If the player has no Ace:
  - Hit if the player has less than 18 and the dealer has more than 6.
  - Hit if the player has exactly 12 and the dealer has less than 5.
  - Hit if the player has exactly 13 and the dealer has a 2 or 3.
- Else stand

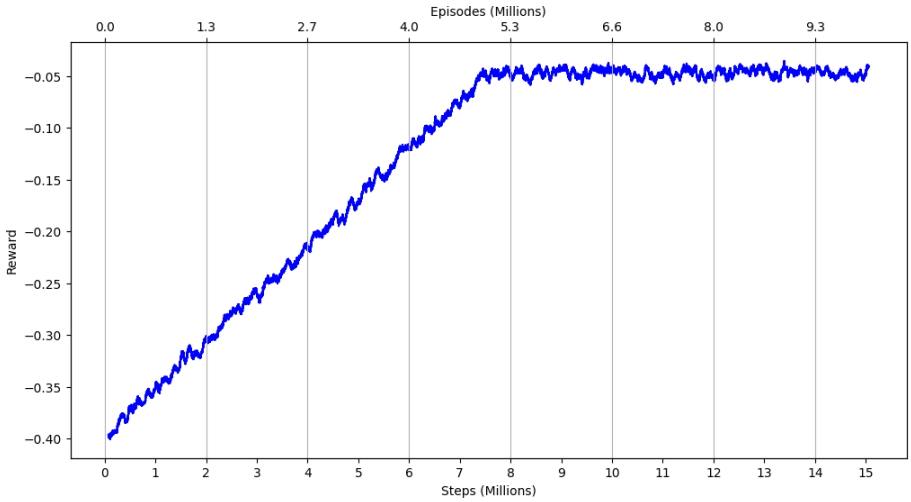

Figure 3: Q-Table TD error evolution over training

Also, it is worth noticing that the generation of the Q-Table with 10 million steps took about 30 minutes, meaning this approach is able to find the optimal policy with a very high efficiency.

2.4 Deep Q-Network (DQN)

Deep Q-Networks are a class of reinforcement learning algorithms that integrate Q-learning with deep neural networks to approximate the optimal action-value function. Unlike traditional Q-learning, which relies on a tabular approach, DQNs can handle high-dimensional state spaces by leveraging the representational power of neural networks. In a single DQN implementation, the network is directly updated based on the observed transitions.

2.4.1 Replay Mechanism

The replay mechanism, commonly referred to as experience replay, is a critical component in DQN training. It involves storing past experiences—each defined as a tuple of state, action, reward, and next state—in a memory buffer. During training, mini-batches of these experiences are randomly sampled to update the network. This approach breaks the temporal correlations between consecutive samples and improves data efficiency by allowing the same experience to be used in multiple updates. As a result, experience replay contributes significantly to the stability and convergence of the learning process.

2.4.2 Network Structure

The network architecture implemented is a fully connected neural network designed to serve as the function approximator for the DQN. It is built using PyTorch, a widely used deep learning framework that provides dynamic computation graphs and efficient tensor operations. The `torch.nn` module, which contains various pre-defined layers and activation functions, is used to define the network's architecture. The structure is as follows:

```
import torch
import torch.nn as nn

class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, output_dim)
        )
```

The `nn.Linear` layers define fully connected layers, where each neuron is connected to all neurons in the previous layer. The `nn.ReLU` activation function introduces non-linearity, allowing the network to learn complex representations. The model is encapsulated within `nn.Sequential`, which simplifies the execution of layers in a sequential manner.

Additionally, PyTorch's autograd engine automatically computes gradients during backpropagation, enabling efficient model training.

2.5 Training parameters

The agent is trained with a set of hyperparameters that balance exploration, learning stability and computational efficiency. The following parameters are key to the training process:

- **Exploration Strategy:**
  - **Initial Epsilon**: The training begins with full exploration, where actions are chosen randomly.
  - **Epsilon Decay**: Epsilon decreases linearly over the first half of the total episodes, gradually shifting the balance from exploration to exploitation.
  - **Final Epsilon**: A minimum exploration rate of 0.01 is maintained to ensure that the agent continues to explore new actions even during later stages of training.

- **Experience Replay Buffer**: The replay buffer stores 10,000 number of past experiences, which are sampled during training to break correlations between sequential data and improve learning stability.
- **Target Network Update Frequency**: The target network is updated every 100 steps by copying the weights from the policy network. This helps stabilize learning by providing a fixed target for a period of time.
- **Learning Rate**: The learning rate is set to 3×10^{-4}. This high rate allows for faster convergence, albeit with a careful balance to avoid instability.
- **Batch Size**: During each update, a batch of 128 experiences is sampled from the replay buffer.
- **Discount Factor**: The discount factor determines the importance of future rewards. Setting it to 0.99 places greater emphasis on long-term rewards, encouraging more farsighted strategies.

2.5.1 Results

The TD error decreased from 0.51 to a steady value of 0.47, suggesting that the network's Q-value updates have become minimal. In comparison, when using Q-Tables, the TD error would have been close to 0, reflecting near-perfect value estimation. However, the TD error in DQN stabilizing at 0.47 is expected given the nature of deep reinforcement learning and the function approximation involved. Unlike Q-Tables, where the state-action values are stored explicitly in a table and updated directly, DQN relies on a neural network to approximate the Q-function. This approximation inherently introduces error due to the limited capacity of the network to represent all state-action pairs with perfect precision, leading to higher TD errors even as the model stabilizes. Furthermore, the use of experience replay, while helpful in reducing variance and stabilizing learning, does not eliminate the inherent bias and error associated with function approximation. Therefore, while the TD error is higher than in Q-Tables, this is not necessarily an indicator of a "worse" agent, but rather a consequence of the chosen method and the trade-offs inherent in using neural networks for Q-function approximation.

Additionally, the reward stabilized between the 700K–800K episodes, with an average reward of -0.053 over the last 200K episodes. This consistent performance, as shown in Figures 4 and 5, indicates that the learning process has reached a plateau.

The resulting policy diverged from previous agents by exhibiting more specific decision rules:

- Without an Ace:
  - Hit if the player's sum is less than 12
  - Hit if the dealer shows more than 8 and the player's sum is less than 15.

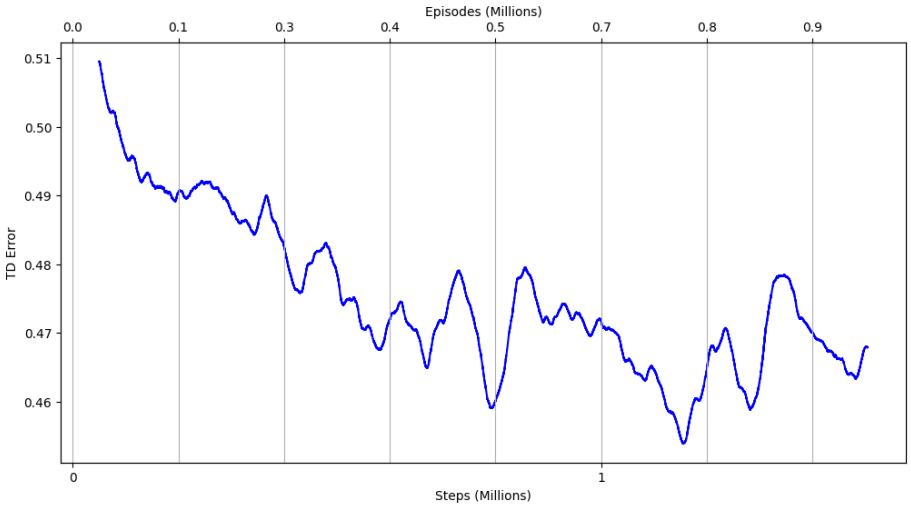

Figure 4: DQN TD error evolution over training

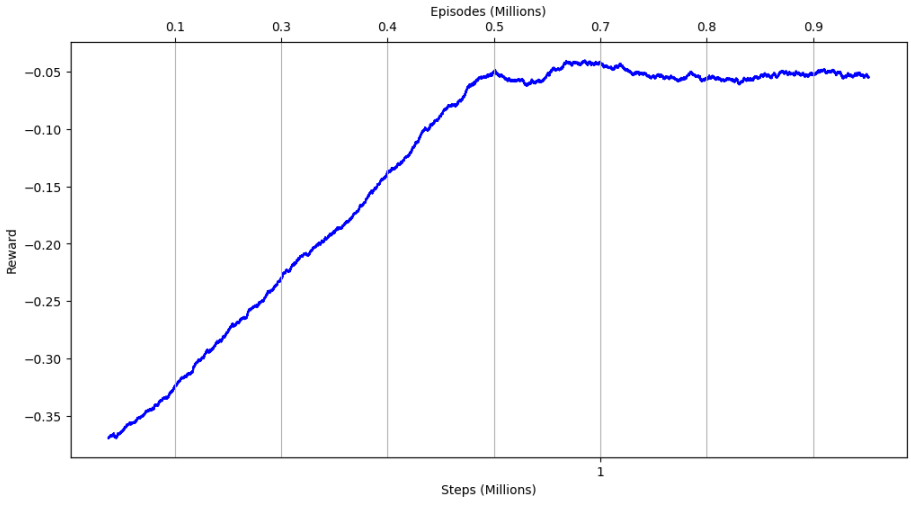

Figure 5: DQN reward evolution over training

  - Hit if the dealer shows a 3 and the player's sum is 15.
  - Hit if the dealer shows less or more than 5 and the player's sum is 12.
  - Hit if the dealer shows a 6 and the player's sum is exactly 13.

- With an Ace:
  - Always hit if the player's sum is less than 16.
  - Hit if the player's sum is 16 and the dealer does not have a 5 or 6.
  - Hit if the player's sum is 17 and the dealer shows more than 7.
  - Hit if the player's sum is 18 and the agent holds either a 10 or an ace.

These specific rules indicate that the function approximation, given the chosen network architecture, has captured a policy that is more rigid than expected.

From a computational point of view, updating the neural network was significantly more expensive than the tabular update seen with Q-Tables. In fact, the training of 1 million episodes took almost 10 hours.

3 Discussion and Conclusions

3.1 Policy Comparison

In this section, we compare the learned policies from the Q-Learning and DQN approach along with the optimal policy. While they share similarities, there are also differences that highlight the limitations of function approximation in DQN compared to the more explicit value storage in Q-Tables.

3.1.1 Q-Learning Policy

The policy learned through Tabular Q-Learning closely approximates the optimal strategy, with only a few notable deviations in specific situations:

- The Q-Learning agent tends to be more conservative when the dealer shows a 9 or 10 and the player holds a soft 18. While the optimal policy recommends hitting in these cases, the agent chooses to stand instead.
- The agent also introduces some deviations for particular hard hands:
  - choosing to hit with a hard 13 against a dealer's 2 or 3
  - choosing to hit with a hard 12 against a dealer's 4.

  In these situations, the optimal policy advises standing.

Despite this small differences, the Q-Learning policy can almost perfectly mimic the optimal strategy.

3.1.2 DQN Policy

The DQN policy, while sharing some patterns with the other policies, introduces more specific decision rules. The use of a neural network for function approximation introduces some constraints on how well the policy generalizes across the entire state space, resulting in a policy that is still somewhat far from the optimal strategy.

3.1.3 Summary

In conclusion, the Q-Learning policy is very close to the optimal policy, with only minor differences, while the DQN policy introduces more rigid rules due to the function approximation constraints. These differences are best seen in the matrix visualizations, where the "Hit" and "Stand" decisions for each sum and dealer card are clearly outlined (see Figure 6 and 7 for a comparative visualization of the policies).

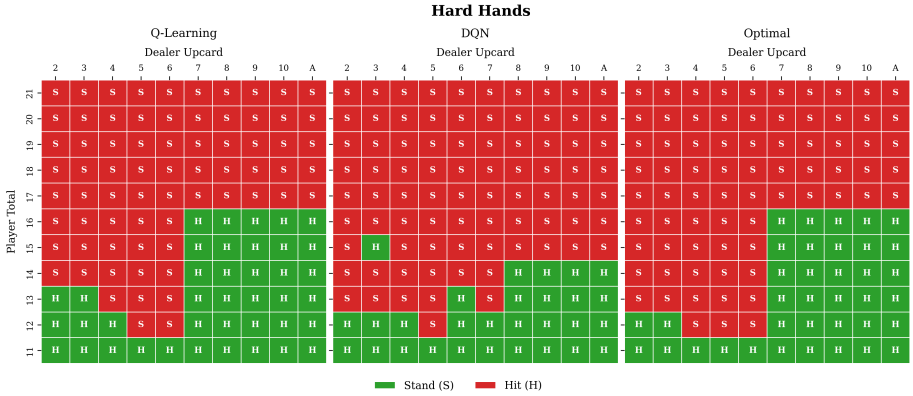

Figure 6: Comparison of the policies for hard hands

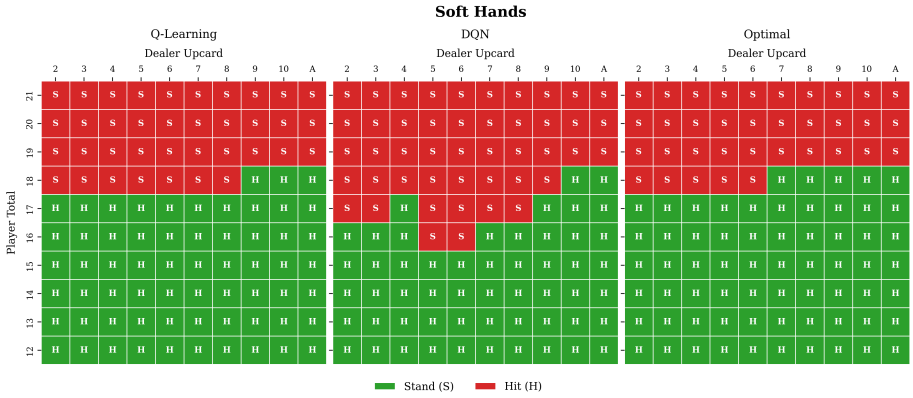

Figure 7: Comparison of the policies for soft hands

3.2 Win Rate Estimation

In this section, we compare the win rates of the four agents—DQN, Q-Learning, Optimal, and Random. To ensure statistical reliability, simulations have been performed over 960,400 episodes, a sample size chosen to achieve a 0.1% error margin with 95% confidence. For the sake of calculating the required number of episodes, we used an estimated win rate of 0.5. This choice was made because the exact win rate is unknown, and using 0.5 maximizes the variance in the binomial distribution, providing a conservative estimate for sample size. However, the actual win rate is lower, as draws are considered losses and the house has an inherent advantage.

To determine the number of episodes required to achieve the 0.1% error margin with 95% confidence, we rely on the properties of the binomial distribution. Since each episode represents a Bernoulli trial with a success probability p, the margin of error E for estimating the win rate is given by:

$$E = Z_{\alpha/2} \times \sqrt{\frac{p(1-p)}{n}} \tag{1}$$

Where:

- $E = 0.001$ (0.1% error margin),
- $Z_{\alpha/2} = 1.96$ (for 95% confidence, hence α=0.05),
- $p = 0.5$ (used for conservative estimation),
- n is the required number of episodes.

Rearranging to solve for n:

$$n = \left(\frac{Z_{\alpha/2}^2 \times p(1-p)}{E^2} \right) \tag{2}$$

Substituting values:

$$n = \left(\frac{1.96^2 \times 0.5(1-0.5)}{0.001^2} \right) \approx 960,400 \tag{3}$$

This ensures that the results are statistically robust with minimal uncertainty.

3.3 Conclusions

The win rates obtained for the four agents are reported in Table 1.

Agent	**Win Rate (%)**
Optimal	43.29
Tabular Q-Learning	43.20
DQN	42.79
Random	28.23

Table 1: Comparison of win rates among different agents in the Blackjack environment.

The results indicate that Tabular Q-Learning and DQN achieve win rates close to the optimal policy, showing that reinforcement learning can effectively learn strategies to counteract the house's advantage. However, Tabular Q-Learning proves to be the superior approach for this environment due to two key factors:

1. **Exploration Speed**: Q-Learning completed **10 million episodes in approximately 30 minutes**, whereas DQN took nearly 10 hours to train for 1 million episodes. Furthermore, Q-Learning achieved convergence after approximately 1.1 million episodes, meaning it had already learned

an effective policy well before reaching 10 million episodes. In contrast, DQN's training time and resource requirements were significantly higher.

2. **Performance in a Discrete Environment**: Since Gym Blackjack is a fully discrete environment (both in state and action space), Tabular Q-Learning is naturally well-suited for this problem. It directly maintains a Q-table, allowing for efficient updates and near-optimal decision-making. The results confirm this, as its win rate of 43.20% is highly competitive with the Optimal policy at 43.29%, outperforming DQN.

Moreover, as shown in the previous section, the policy obtained by Tabular Q-Learning is incredibly similar to the proven optimal policy. While there are minor differences in state-action mappings when comparing to the Optimal policy, these discrepancies do not significantly impact overall performance. This further supports the idea that Q-Learning effectively approximates the optimal strategy, leveraging its tabular structure efficiently in this discrete environment.

The Random policy serves as a baseline, with a significantly lower win rate of 28.23%, demonstrating the impact policies have on the outcome of each game.

While both DQN and Tabular Q-Learning are capable of learning effective Blackjack strategies, the latter proves to be the superior approach in this particular environment. Its faster convergence, lower computational cost, and strong performance in a discrete state-action space makes it the most effective reinforcement learning algorithm for this task. The fact that it closely matches the Optimal policy further solidifies its suitability. Meanwhile, DQN's significantly higher training time and slightly lower performance makes it a less practical choice for this problem.

References

[1] Roger R. Baldwin et al. "The Optimum Strategy in Blackjack". In: *Journal of the American Statistical Association* 51.275 (1956), pp. 429–439. DOI: 10.1080/01621459.1956.10501361.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.