

Artificial Intelligence & Machine Learning

Machine Learning Project: BlackJack



Implementation of a Reinforcement Learning Agent using
Tabular Q-Learning and Deep Q-Network (DQN)

Professor:

Fabio Patrizi

Students:

Santavicca Federico 2003442

Pedicillo Marco 1983285

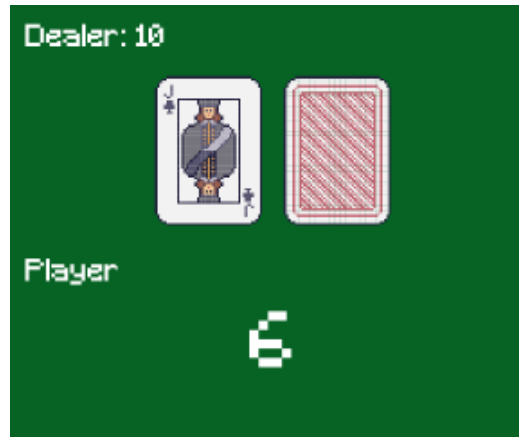
Academic Year 2024/2025

Index

1. Introduction	2
2. Tabular Q-Learning	5
2.1 Our implementation	5
2.2 Results	9
3. Deep Q-Network	15
3.1 Our implementation	15
3.2 Results	19
4. Comparison	22

1. Introduction

The Game



Blackjack is a widely known card game in which the player's goal is to beat the dealer by obtaining a hand value as close as possible to 21, without exceeding it. The game is modeled with an infinite deck (i.e., cards are drawn with replacement), and involves two participants: the player and the dealer.

The card values are defined as follows:

- **Face cards** (Jack, Queen, King) are worth 10 points;
- **Numerical cards** (2–10) have their respective numeric values;
- **Aces** can count as either 1 or 11, depending on which is more advantageous to the player. If counting an ace as 11 would cause the player to go over 21, it is automatically treated as 1. An ace is considered *usable* if it is currently counted as 11.

The game begins with the player receiving two face-up cards and the dealer receiving one face-up and one face-down card. The player may choose to:

- **Hit (1)**: draw another card;
- **Stick (0)**: keep their current hand.

If the player's hand exceeds 21, they immediately lose (bust). Otherwise, once the player decides to stick, the dealer reveals their hidden card and draws additional cards until reaching a hand value of 17 or higher. The winner is whoever has the hand closest to 21 without busting. If both hands have the same value, the result is a draw.

The Problem of Reinforcement Learning

Blackjack serves as a classic example for studying Reinforcement Learning (RL), as it naturally maps to a Markov Decision Process (MDP). Reinforcement Learning is a framework in which an agent learns to make decisions by interacting with an environment, aiming to maximize cumulative rewards over time. An MDP provides the mathematical foundation for this framework, describing the environment as a set of states, actions, transition dynamics, and rewards. In this project, we use the **Blackjack-v1** environment provided by Gymnasium, a widely used library for simulating RL environments.

The environment is formally defined as follows:

- Observation space: a 3-tuple consisting of:
 - The player's current sum (from 4 to 21),
 - The dealer's showing card (1 to 10, where 1 represents an ace),
 - A binary flag indicating whether the player has a usable ace (0 or 1).

```
Observation Space = Tuple(Discrete(32),  
Discrete(11), Discrete(2))
```

- Action space:
 - 0: Stick (stop drawing cards)
 - 1: Hit (draw another card)

```
Action Space = Discrete(2)
```

- Reward function:
 - Win: +1
 - Lose: -1
 - Draw: 0
 - Win with natural blackjack: +1.5 if enabled via natural=True
- Episode termination:
 - The player busts by exceeding 21;
 - The player sticks and the dealer completes their turn.

Project Objective

The objective of this project is to train an agent capable of learning an optimal strategy for playing Blackjack through trial-and-error interaction with the environment. To this end, two fundamental Reinforcement Learning approaches are implemented and compared:

1. **Tabular Q-learning**: a model-free algorithm that uses a Q-table to estimate the action-value function $Q(s,a)$ for each discrete state-action pair. It is effective when the state space is small and fully enumerable.
2. **Deep Q-Network (DQN)**: an extension of Q-learning that uses a neural network to approximate the Q-function. This approach generalizes better to large or continuous state spaces.

Both approaches are trained and evaluated using the [Blackjack-v1](#) environment to assess their learning performance and ability to converge to a winning strategy.

2. Tabular Q-Learning

Tabular Q-learning is a fundamental algorithm in Reinforcement Learning that enables an agent to learn optimal decision-making policies through interaction with a discrete environment. The core idea is to maintain a **Q-table**, where each entry $Q(s,a)$ estimates the expected return of taking action a in state s , and subsequently following the best possible policy.

During learning, the agent updates the Q-values based on its experiences, gradually refining its understanding of which actions lead to higher rewards. Since the algorithm is **model-free** and **off-policy**, it does not require a model of the environment and can learn optimal behavior even while exploring.

A key aspect of Q-learning is balancing **exploration** and **exploitation**. Early in training, the agent must explore the environment sufficiently to discover which actions yield good long-term rewards. This is commonly achieved using an **ϵ -greedy strategy**, where the agent selects a random action with probability ϵ , and chooses the action with the highest estimated Q-value with probability $1-\epsilon$. Over time, the value of ϵ is typically decayed to favor exploitation, allowing the agent to increasingly rely on its learned knowledge.

2.1 Our Implementation

Imports and Environment Setup

We begin by importing the required libraries:

- **gymnasium**: to access and manage the Blackjack environment;
- **defaultdict**: to initialize a flexible Q-table structure;
- **numpy**: for numerical operations;
- **matplotlib** and **seaborn**: for visualizations;
- **tqdm**: to provide progress bars during training.

The environment is created using:

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

Setting `sab=True` ensures that the version of the game matches the one described in the *Sutton & Barto* textbook.

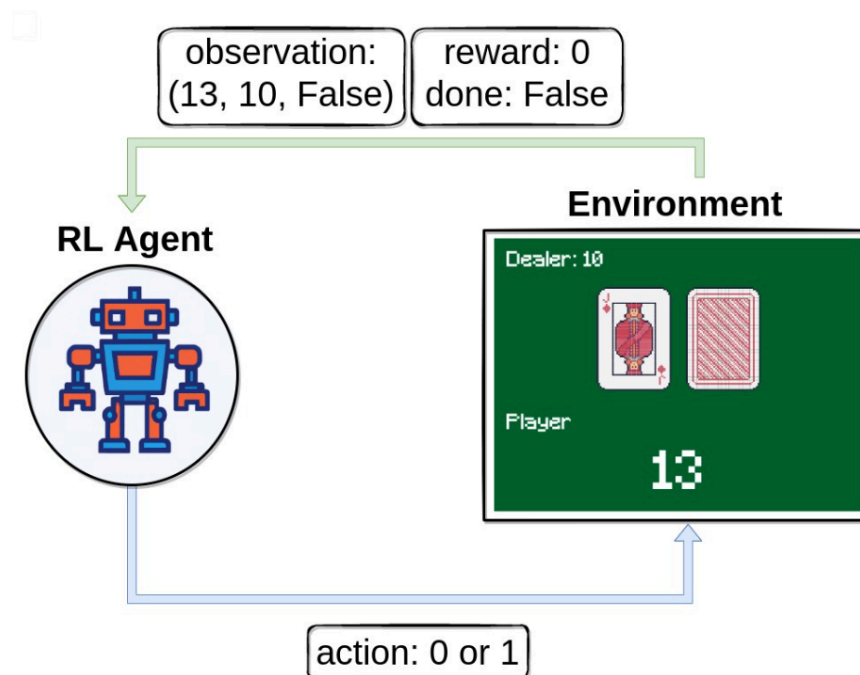
In this environment, the state is a tuple:

```
(player_sum, dealer_card, usable_ace)
```

and actions are -> 0: stick and 1: hit

Agent Definition

We define a class `BlackjackAgent` that encapsulates all the logic for learning and acting in the environment.



The agent uses a `defaultdict` to store Q-values, automatically initializing unseen states with a zero-valued array for both possible actions:

```
self.q_values=defaultdict(lambda:np.zeros(env.action_space.n))
```

The agent is initialized with the following hyperparameters:

- `learning_rate` (α): step size for Q-value updates.
- `discount_factor` (γ): weight of future rewards.
- `initial_epsilon`, `final_epsilon`, `epsilon_decay`: control the ϵ -greedy exploration schedule.

The `get_action()` method implements the **ϵ -greedy policy**. At each decision point, the agent chooses:

- a **random action** with probability ϵ (exploration),
- the **action with the highest Q-value** for the current state with probability $1-\epsilon$ (exploitation).

This encourages the agent to explore the environment in early episodes and exploit its knowledge in later episodes:

```
if np.random.random() < self.epsilon:
    return env.action_space.sample()
else:
    return int(np.argmax(self.q_values[obs]))
```

The core learning logic is implemented in the `update()` method. For each transition (s,a,r,s') , the Q-value is updated as follows:

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

In the code:

```
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.lr *
temporal_difference
```

If the episode has terminated, the future reward is assumed to be 0.

A record of the temporal-difference error is maintained in `self.training_error`, which can be useful for analysis and convergence tracking.

After each episode, the value of epsilon is decreased linearly:

```
self.epsilon = max(self.final_epsilon, self.epsilon -
self.epsilon_decay)
```

This ensures that the agent explores widely in early episodes and gradually focuses on exploiting its learned policy.

Training Procedure

The training process begins with the configuration of key hyperparameters that govern the learning dynamics of the agent:

- `learning_rate = 0.01`: controls how much the Q-value is updated at each step (i.e., the step size α).
- `n_episodes = 600,000`: defines the total number of episodes the agent will experience during training.
- `start_epsilon = 1.0`: sets the initial probability of choosing a random action, ensuring full exploration at the start.
- `epsilon_decay = start_epsilon / (n_episodes / 2)`: linearly reduces the exploration rate so that it reaches the `final_epsilon` value halfway through training.
- `final_epsilon = 0.05`: sets the lower bound on exploration to prevent the agent from becoming entirely greedy.

The agent is then instantiated with these parameters and the Blackjack environment. To collect statistics during training (such as cumulative reward or episode length), the environment is wrapped using Gymnasium's `RecordEpisodeStatistics` wrapper with a buffer length equal to the number of episodes.

In addition to the agent, three counters (`n_wins`, `n_losses`, `n_draws`) are initialized to keep track of the outcomes of each episode. A list called `epsilon_history` is also used to store the value of epsilon after each episode, which will later be used to visualize the exploration decay over time.

The main training loop iterates over the specified number of episodes (`n_episodes`). For each episode:

1. The environment is reset and the initial observation (game state) is retrieved.
2. The episode continues until it ends (either by the player busting or choosing to stick).

3. In each time step:

- The agent selects an action using the ϵ -greedy policy via `agent.get_action(env, obs)`.
- The environment executes the action and returns the next state, reward, and termination flags.
- The agent updates its Q-table using the transition (s,a,r,s') via `agent.update(...)`.
- The state is updated for the next step.

Once the episode ends, the final outcome is determined based on the reward:

- `+1` indicates a win,
- `-1` indicates a loss,
- `0` indicates a draw.

The corresponding counter is incremented accordingly.

At the end of each episode, the agent's epsilon value is updated using the `decay_epsilon()` method, which decreases it by a fixed step while ensuring it never falls below `final_epsilon`. The current value of epsilon is appended to `epsilon_history` for later analysis.

2.2 Results

In this section, we present the results of our Tabular Q-learning agent, divided into two parts:

- **Training results** show how the agent improved over time, using metrics such as rewards, episode lengths, epsilon decay, and the learned policy.
- **Testing results** evaluate the final policy on 10,000 new episodes using a greedy strategy (no exploration), to measure real-world performance.

Training results

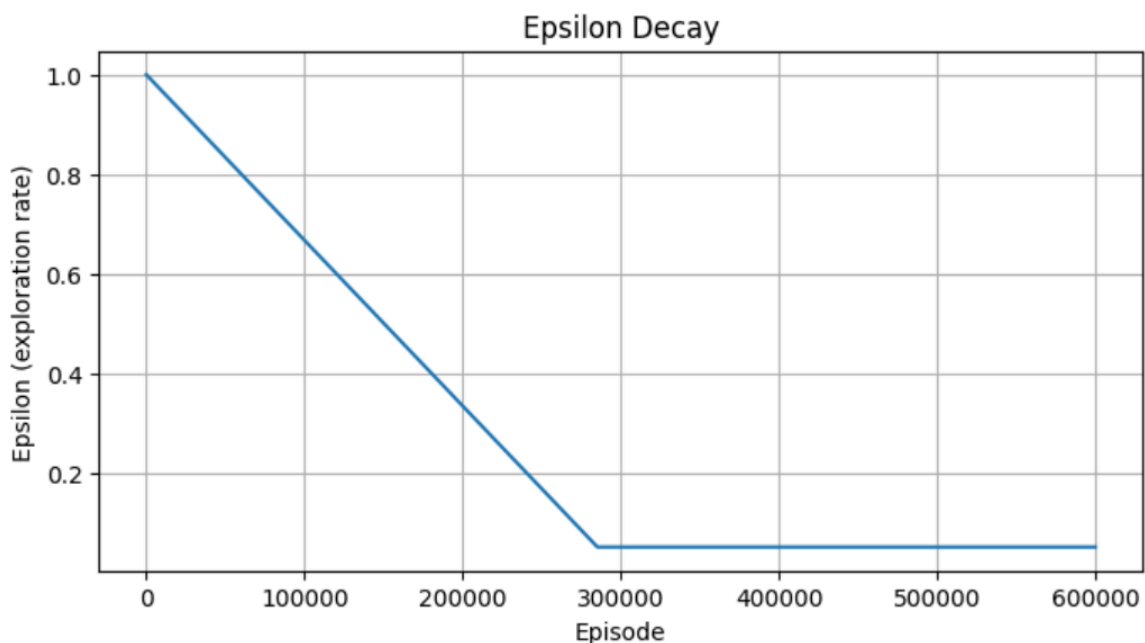
The performance of the Tabular Q-learning agent was evaluated over 600,000 training episodes. Throughout the training process, several metrics were collected to monitor the learning progress, including the number of wins, losses, and draws, the evolution of the exploration rate (ϵ), episode rewards, episode lengths, and the agent's temporal-difference (TD) error.

At the end of training, the total number of episode outcomes was as follows:

```
--- Final results after training ---  
Total wins:      234030 ✓  
Total losses:    320013 ✗  
Total draws:     45957 🟡
```

A key component of the training procedure was the use of an ϵ -greedy policy to balance exploration and exploitation. To monitor this, the value of ϵ was recorded at each episode and plotted.

As expected, ϵ decreased linearly from 1.0 to 0.05 over the course of 300,000 episodes, and remained constant thereafter. This behavior ensures that the agent explores extensively in the first half of training and exploits its learned policy in the second half.



To further analyze learning dynamics, three moving average plots were generated:

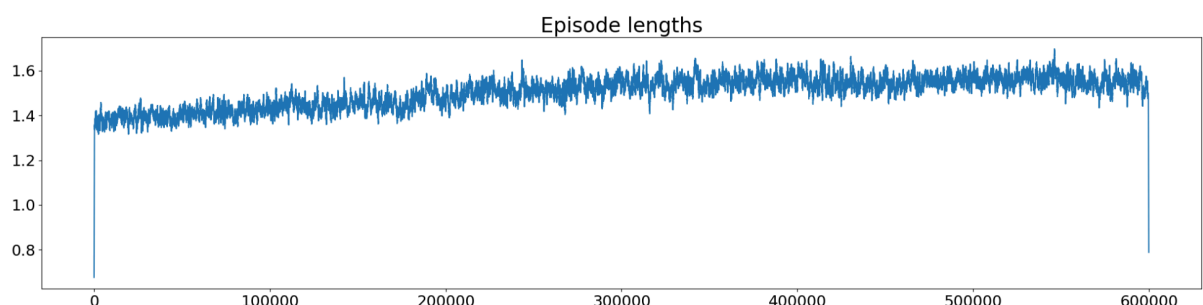
1. Episode Rewards:

It shows the moving average of episode rewards throughout training. Initially, the agent consistently receives negative rewards, reflecting frequent losses and suboptimal actions. Over time, the average reward increases, approaching zero and even exceeding it in some intervals. This trend indicates that the agent is learning to make better decisions, leading to fewer losses and more wins as training progresses.



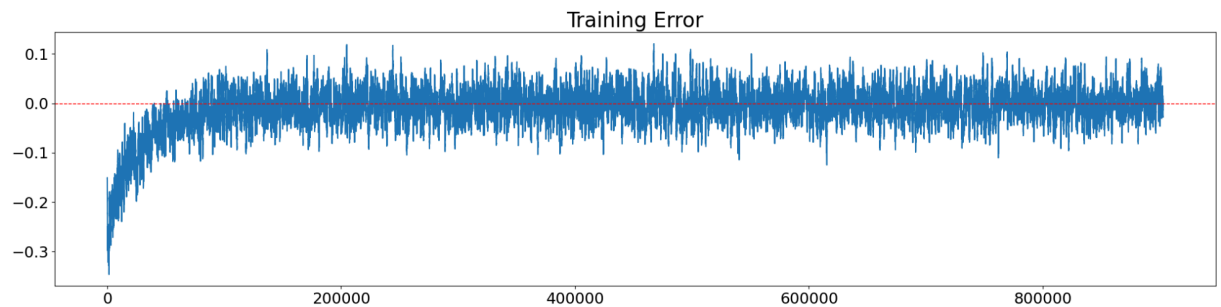
2. Episode Lengths:

This plot shows the moving average of episode lengths over time, representing the average number of steps taken before each episode ends. At the beginning of training, episode lengths are relatively short, as the agent is still learning and tends to make poor decisions, which often lead to early termination. As training progresses, the agent improves its strategy, playing more cautiously, resulting in longer and more stable episodes. The sharp drops at the beginning and end of the curve are artifacts of the moving average calculation. Specifically, at the boundaries of the data, there are fewer available samples to compute a reliable average, which causes these temporary distortions.



3. Temporal-Difference (TD) Error:

It shows the error over time, which measures how much the agent adjusts its Q-values at each step. In early training, the TD error fluctuates significantly, reflecting high uncertainty and frequent updates to the value estimates. As training continues and the agent's policy stabilizes, the TD error decreases and becomes less volatile. This convergence suggests that the agent is increasingly confident in its value function and making smaller corrections.



To evaluate the quality and interpretability of the learned policy, we generated value and policy plots based on the final Q-table. These plots are computed separately for the two distinct scenarios in Blackjack:

- **With a usable ace** (Ace counts as 11)
- **Without a usable ace** (Ace counts as 1)

For each case, two visualizations are created:

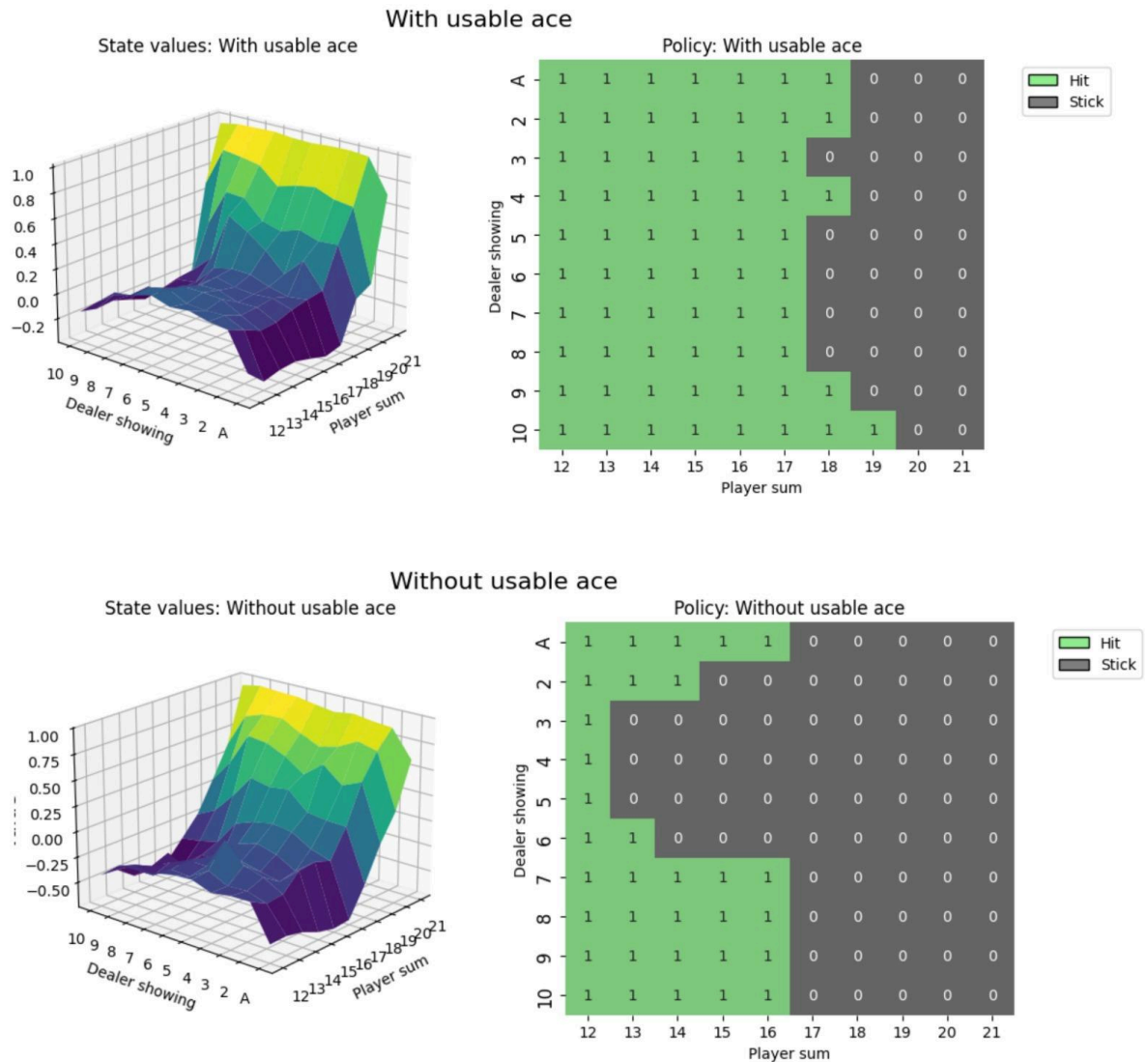
1. State-Value Surface Plot (3D):

Shows the maximum Q-value for each state (player_sum, dealer_card). Peaks in this surface reflect high-value states (typically close to 21), and valleys represent unfavorable situations.

2. Policy Heatmap (2D):

Shows which action, *hit* or *stick*, the agent chooses in each state. This plot reveals the agent's learned strategy and can be compared to the known optimal policy.

- Green squares represent **hit**.
- Grey squares represent **stick**.



Testing results

After training, the agent was evaluated on 10,000 new episodes using a fully greedy policy ($\epsilon = 0$), meaning it always selected the action with the highest estimated Q-value. The purpose of this test was to assess the agent's real-world performance without further exploration.

While the agent lost slightly more games than it won, the results demonstrate that it has learned a strategy that performs significantly better than random behavior. The win rate, along with the moderate number of draws, indicates that the agent consistently makes reasonable decisions.

```

--- Test results ---
Wins:    4378  ✓
Losses:  4730  ✗
Draws:   892   🤝
Win rate: 43.78% 🏆
  
```

3. Deep Q-Network

Deep Q-Networks (DQN) extend the idea of Q-learning by replacing the Q-table with a **neural network** that approximates the action-value function $Q(s,a)$. This approach allows reinforcement learning to scale to environments with large or continuous state spaces, where storing a separate Q-value for every state-action pair is infeasible. In DQN, the network takes the current state as input and outputs a Q-value for each possible action, enabling the agent to generalize across similar states.

Training is performed using **mini-batches of past experiences** sampled from a **replay buffer**, which helps break correlations between consecutive updates and improves stability. The Q-values are updated by minimizing the difference between the predicted value and a target computed using the Bellman equation, similar to traditional Q-learning.

In the context of Blackjack, DQN provides an alternative to tabular methods by allowing the agent to learn directly from the raw state representation without explicitly storing values for every possible state.

3.1 Our Implementation

Imports and Environment Setup

Again, we begin by importing the required libraries:

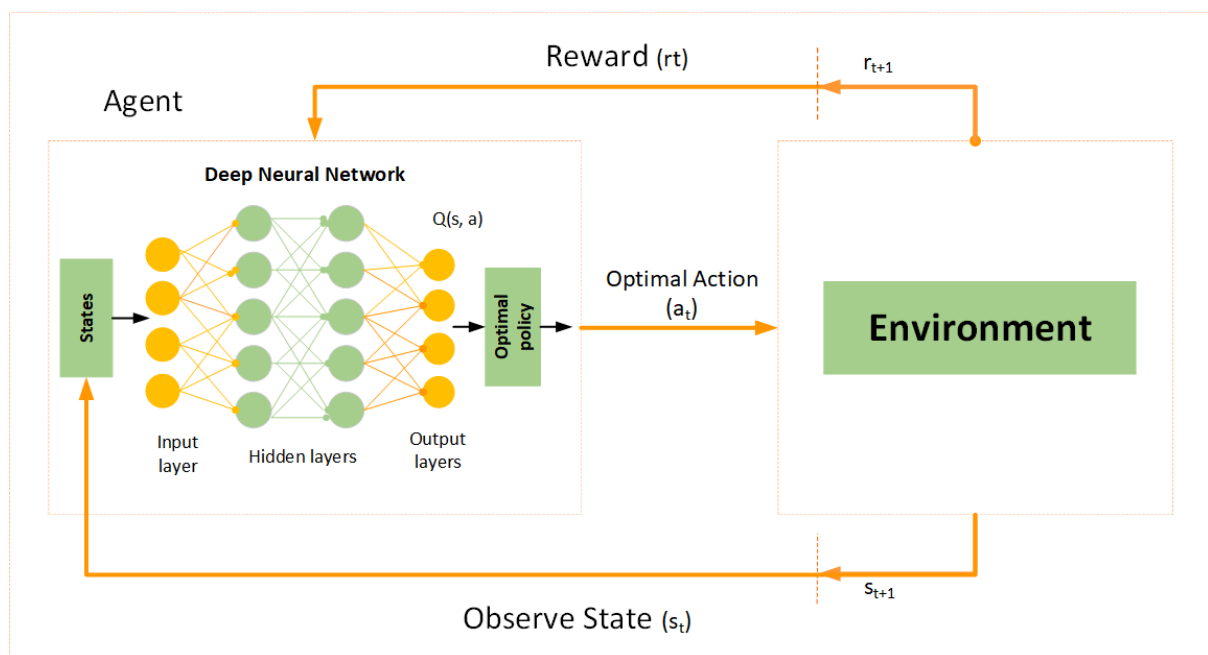
- **torch.nn**: to define the architecture of the neural network;
- **torch.optim**: to handle the optimization process during training;
- **deque** from **collections**: to implement the replay buffer for storing past experiences;

The environment is the same as before:

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

DQN Setup

In our Deep Q-Network implementation, we begin by defining the key hyperparameters that control the learning dynamics and exploration strategy of the agent. The training is carried out over 600,000 episodes, with a discount factor $\gamma = 0.99$ that prioritizes long-term rewards. The exploration-exploitation trade-off is handled using an ϵ -greedy policy, where the exploration rate ϵ starts at 1.0 (pure exploration) and decays exponentially over time toward a minimum of 0.05. The rate of decay is controlled by a constant $\epsilon_decay = 80000$, and the epsilon value at any given step is computed using an exponential decay function. This ensures that the agent explores extensively in the early stages of training and gradually shifts to exploiting its learned policy as learning progresses.



The DQN model itself is defined as a simple feedforward neural network using PyTorch. The input to the network is a three-dimensional state vector representing the player's sum, the dealer's visible card, and the presence of a usable ace. The network architecture consists of two hidden layers with 64 units each, and an output layer with two units corresponding to the estimated Q-values for the available actions:

- 0: stick
- 1: hit

This design allows the network to learn a mapping from state inputs to action values, enabling the agent to approximate the optimal Q-function.


```

class DQN(nn.Module):

    def __init__(self):
        super().__init__()

        self.fc = nn.Sequential(

            nn.Linear(3, 64),

            nn.ReLU(),

            nn.Linear(64, 64),

            nn.ReLU(),

            nn.Linear(64, 2)

        )

    def forward(self, x):

        return self.fc(x)

```

To enable efficient and stable training, we incorporate an experience replay mechanism via a **ReplayBuffer** class. The buffer is implemented using a fixed-size double-ended queue ([deque](#)) that stores individual transitions of the form (s,a,r,s',done). Each time the agent interacts with the environment, a new transition is added to the buffer. Once a minimum number of experiences (2,000) is accumulated, the training process begins using mini-batches of 64 transitions sampled at random from the buffer. When sampled, each component of the transition (states, actions, rewards, next states, and done flags) is converted into PyTorch tensors and moved to the appropriate computation device, either GPU or CPU, depending on hardware availability.

```

class ReplayBuffer:

    def __init__(self, capacity):

        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state,
done):

        self.buffer.append((state, action, reward,
next_state, done))

    def sample(self, batch_size):

```

```

        batch = random.sample(self.buffer, batch_size)

        states, actions, rewards, next_states, dones =
zip(*batch)

        return (

            torch.FloatTensor(states).to(device),

            torch.LongTensor(actions).to(device),

            torch.FloatTensor(rewards).to(device),

            torch.FloatTensor(next_states).to(device),

            torch.FloatTensor(dones).to(device)

        )

```

Training Procedure

The training phase begins by setting up the core components of the learning system:

- **Q-Network:** We instantiate the DQN neural network and move it to the appropriate computation device (cuda if available, else cpu). This network is responsible for approximating the Q-function $Q(s,a)$, mapping each input state to a vector of Q-values, one for each possible action.
- **Optimizer and Loss Function:** The network's parameters are optimized using the Adam optimizer with a learning rate defined by `lr`. The Mean Squared Error (MSE) loss function is used to compute the difference between predicted Q-values and the target values based on the Bellman equation.
- **Replay Buffer:** A `ReplayBuffer` is created to store transitions experienced by the agent in the form (state, action, reward, next_state, done). This mechanism enables experience replay, allowing the model to learn from a diverse set of past experiences.
- **Tracking Variables:** Additional variables like `episode_rewards` are initialized to record the cumulative reward for each episode, and `step_count` is used to control the epsilon decay over time.

The agent is trained over a fixed number of episodes. For each episode:

1. **Environment Reset:** The Blackjack environment is reset and the initial state is obtained.
2. **Exploration vs. Exploitation (ϵ -greedy Policy):**
 - The agent computes the current value of ϵ using an exponential decay function, gradually reducing exploration in favor of exploitation.
 - A random number is drawn: if it is lower than ϵ , the agent explores by selecting a random action; otherwise, it exploits the neural network by choosing the action with the highest Q-value.
3. **Transition Collection:**
 - The chosen action is executed in the environment.
 - The resulting next state, reward, and termination signal are recorded.
 - This transition is added to the replay buffer for later sampling.

After the buffer has reached a minimum size (`min_buffer_size`), learning begins:

1. **Batch Sampling:** A mini-batch of transitions is sampled randomly from the buffer.
2. **Q-value Computation:**
 - The current Q-values for the selected actions are extracted from the network predictions using `gather`.
 - The Q-values for the next states are computed to estimate the future reward.
3. **Target Calculation:**
 - The target Q-value is calculated using the Bellman formula:
$$\text{target} = r + \gamma \cdot \max_{a'} Q(s', a') \cdot (1 - \text{done})$$
 - The future reward is masked if the episode has terminated.

4. Loss and Backpropagation:

- The MSE loss between predicted and target Q-values is computed.
- The network is updated using backpropagation and the optimizer step.

This process continues for every step within the episode.

3.2 Results

Training results

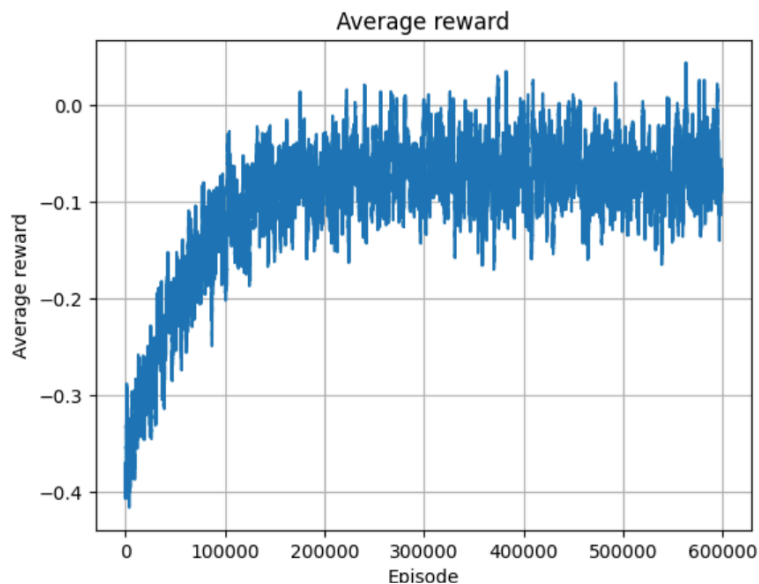
After each episode, the total reward is appended to `episode_rewards`. Every 5000 episodes, the average reward over the last 5000 episodes is printed along with the current value of ϵ , providing insight into the training progression.

```
...
Episode 265000, Average Reward: -0.073, Epsilon: 0.057
Episode 270000, Average Reward: -0.040, Epsilon: 0.056
Episode 275000, Average Reward: -0.058, Epsilon: 0.056
Episode 280000, Average Reward: -0.077, Epsilon: 0.055
Episode 285000, Average Reward: -0.050, Epsilon: 0.055
Episode 290000, Average Reward: -0.067, Epsilon: 0.054
Episode 295000, Average Reward: -0.074, Epsilon: 0.054
Episode 300000, Average Reward: -0.085, Epsilon: 0.053
Episode 305000, Average Reward: -0.071, Epsilon: 0.053
Episode 310000, Average Reward: -0.066, Epsilon: 0.053
Episode 315000, Average Reward: -0.056, Epsilon: 0.053
Episode 320000, Average Reward: -0.048, Epsilon: 0.052
Episode 325000, Average Reward: -0.079, Epsilon: 0.052
Episode 330000, Average Reward: -0.035, Epsilon: 0.052
Episode 335000, Average Reward: -0.094, Epsilon: 0.052
Episode 340000, Average Reward: -0.062, Epsilon: 0.052
Episode 345000, Average Reward: -0.048, Epsilon: 0.051
Episode 350000, Average Reward: -0.079, Epsilon: 0.051
Episode 355000, Average Reward: -0.067, Epsilon: 0.051
Episode 360000, Average Reward: -0.099, Epsilon: 0.051
Episode 365000, Average Reward: -0.081, Epsilon: 0.051
Episode 370000, Average Reward: -0.057, Epsilon: 0.051
Episode 375000, Average Reward: -0.066, Epsilon: 0.051
Episode 380000, Average Reward: -0.078, Epsilon: 0.051
Episode 385000, Average Reward: -0.052, Epsilon: 0.051
Episode 390000, Average Reward: -0.068, Epsilon: 0.051
Episode 395000, Average Reward: -0.075, Epsilon: 0.051
Episode 400000, Average Reward: -0.039, Epsilon: 0.050
...
```

Two plots are generated after training to analyze the agent's performance:

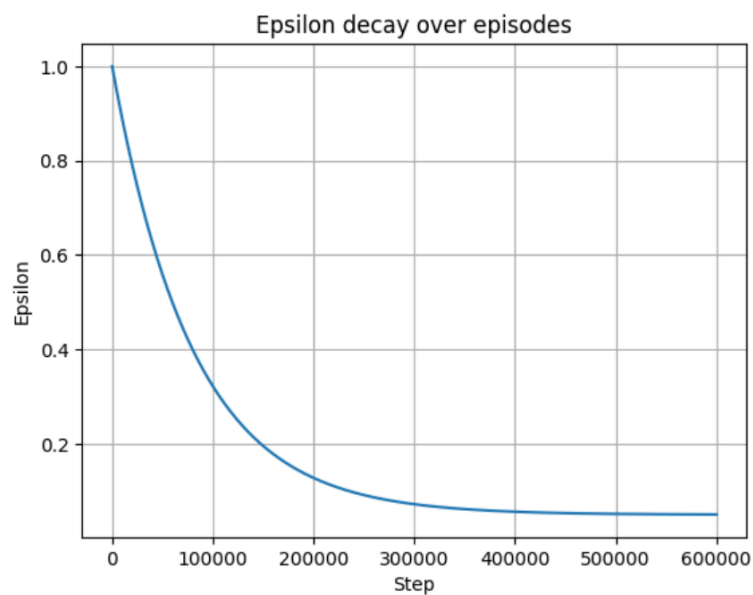
1. Average Reward Plot:

- Shows the moving average of total episode rewards.
- Useful for observing convergence and stability over time.



2. Epsilon Decay Plot:





- Illustrates how ϵ decreased throughout training.
- Demonstrates the shift from exploration to exploitation as training progressed.



Testing results

After completing the training phase, we evaluate the performance of the Deep Q-Network agent over 10,000 episodes. During this phase, exploration is fully disabled ($\epsilon = 0$), meaning the agent consistently selects the action with the highest Q-value for each state, fully exploiting the learned policy.

The agent achieves the following results:

```
Results over 10000 episodes:  
Wins: 4290 (42.90%)   
Losses: 4814 (48.14%)   
Draws: 896 (8.96%)   
Average reward: -0.0524 
```

The **average reward** being slightly negative indicates that, while the agent has learned a reasonable strategy, it still suffers more losses than wins overall. This is consistent with the inherent difficulty and stochastic nature of Blackjack, where the dealer typically has a slight advantage.

4. Comparison

Both the tabular Q-learning agent and the Deep Q-Network (DQN) demonstrate the ability to learn an effective strategy for Blackjack through interaction with the environment. In both cases, the average reward improves significantly during training. However, a key difference emerges in the later stages of learning:

The tabular agent occasionally surpasses the zero-reward threshold, indicating that its learned policy can yield a slight net gain over time. In contrast, the **DQN agent**, while still improving, **remains consistently just below zero**, suggesting a policy that performs reasonably well but does not reach the same level of refinement.

This gap can be attributed to the fact that tabular Q-learning explicitly stores and updates Q-values for each state-action pair in a fully observable and discrete environment, while DQN relies on function approximation, which introduces a degree of estimation error that may prevent the agent from fully optimizing its behavior.