AI 3603 Artificial Intelligence: Principles and Techniques

By: 陈路轩 (523030910014)

HW#: 2

2025 年 11 月 14 日

I. INTRODUCTION

A. Problem background

This assignment consists of three core tasks, designed to progressively build and analyze capabilities from basic tabular RL to Deep Reinforcement Learning (DRL):

- **Cliff-walking:** A classic grid-world problem. The agent must navigate a 12×4 grid from a start point to a goal point. The grid includes a "cliff" region, which provides a large negative reward and returns the agent to the start. The agent must learn to find an efficient path while avoiding the cliff.
- **Lunar Lander:** A more complex control problem. The agent must control a lander to touch down safely on a landing pad between two flags in a simulated lunar gravity environment. The state space is continuous (8 dimensions) and the action space is discrete (4 actions). This requires the agent to learn not just where to go, but how to control its thrusters for a smooth landing.

B. Task Objectives

This assignment has three core tasks, designed to progressively build a comprehensive path-planning system:

- **Task 1**: To implement the Sarsa, Q-Learning, and Dyna-Q algorithms. Then plot the episode reward curves, ϵ-decay curves, and visualize the final paths learned by the agents. Compare and analyze the performance of the three algorithms.
- **Task 2**: To read and understand the provided `dqn.py` code, adding comments to key sections. Then train and tune the DQN agent on the Lunar Lander environment, plotting the reward and ϵ-decay curves. Finally, visualize the trained agent's landing behavior.
- **Task 3**: Find and learn an exploration strategy other than ϵ-greedy.

II. ALGORITHM DESIGN AND IMPLEMENTATION

A. Task 1: Reinforcement Learning in Cliff-walking Environment

1. *Description*

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns by trial and error through interactions with an environment. The goal is to acquire a policy to get as high as possible scores in the game. In this task, we implement RL agents based on Sarsa, Q-Learning, and Dyna-Q algorithms to find a safe path to the goal in a grid-shaped maze. The environment is a 12×4 grid map, and the agent is restricted to moving only upward, downward, leftward, and rightward.

2. *Formulation*

The core of these RL algorithms lies in their components: states, actions, and a reward function. The agent learns an action-value function, $Q(s, a)$, to determine the expected return of taking an action in a given state.

1. **State (s_t):** This value is an integer representing the agent's current coordinate (x, y) .
2. **Action (a_t):** This value is $a_t \in \{0, 1, 2, 3\}$, where the four integers represent the four moving directions (up, down, left, right) respectively.
3. **Reward (r):** This value represents the feedback from the environment. In this task, every step costs -1. Falling into the cliff gives a punishment of -100 and returns the agent to the starting point.

Q-value Update (Sarsa): The agent updates its Q-value estimation based on experience. The update rule for Sarsa (on-policy) is:

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

Q-value Update (Q-Learning): The update rule for Q-Learning (off-policy) is:

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

Model Learning and Planning (Dyna-Q): Dyna-Q integrates model-free interaction with model-based planning. In addition to the direct Q-Learning update from real experience (Direct RL), it involves:

1. **Model Learning:** The agent uses the experience (s, a, r, s') to update a model, $Model(s, a) \leftarrow (r, s')$.
2. **Planning:** The agent performs n planning steps. For each step, it randomly samples a previously observed state-action pair (s_{plan}, a_{plan}), uses the model to get the simulated next state s'_{plan} and reward r_{plan}, and then applies the Q-Learning update rule:

$$Q(s_{plan}, a_{plan}) \leftarrow Q(s_{plan}, a_{plan}) + \alpha[r_{plan} + \gamma \max_{a'} Q(s'_{plan}, a') - Q(s_{plan}, a_{plan})]$$

In these formulas, s is the current state, a is the current action, r is the reward received, s' is the next state, and a' is the next action. α is the learning rate, and γ is the reward decay.

a. **Core Data Structures:**

  (a) **Agent Class:** Separate classes are defined for each algorithm (SarsaAgent, QLearningAgent, Dyna_QAgent) in `agent.py`. Each instance stores hyperparameters like alpha, gamma, and epsilon.

  (b) **Q-Table:** A data structure (typically a 2D numpy array) within each agent, mapping state-action pairs to their estimated Q-values. It is initialized with the state dimension and number of actions.

  (c) **Model (only for Dyna-Q):** A data structure specific to `Dyna_QAgent` used to store experiences (s, a, r, s'). This model is used for planning by simulating interactions.

b. **Algorithm Execution Flow:**

  (a) **Initialization:** Create the Gym environment (`gym.make`). Then, construct the agent with its hyperparameters.

  (b) **Main Training Loop:** The agent is trained over a fixed number of episodes (e.g., 1000).

  (c) **Episode Reset:** At the start of each episode, the environment is reset (`env.reset()`) to get the initial state `s`.

  (d) **Step Interaction Loop:** Within an episode, the agent interacts with the environment for a maximum number of steps or until the terminal state is reached (`isdone`).

  (e) **Action Selection:** The agent chooses an action `a` based on state `s` using an ϵ-greedy policy(Explore randomly with probability epsilon; otherwise, exploit the currently known best action.) (`agent.choose_action(s)`).

  (f) **Environment Step:** The chosen action `a` is sent to the environment (`env.step(a)`), which returns the next state `s_`, reward `r`, and done flag `isdone`.

  (g) **Learning (Q-Update):** The agent's `learn` function is called with the experience tuple.

    - **Sarsa:** Requires the next action `a_` chosen by the policy at `s_`. The update is `agent.learn(s, a, r, s_, a_)`.
    - **Q-Learning / Dyna-Q:** The update is based on the max Q-value at `s_`. The call is `agent.learn(s, a, r, s_)`. Dyna-Q also performs n planning steps internally during this call.

  (h) **State Transition:** The current state `s` is updated to `s_`. (For Sarsa, `a` is also updated to `a_`).

  (i) **Epsilon Decay:** After each episode, the `epsilon` value is reduced via `agent.decay_epsilon()` to shift from exploration to exploitation.

  (j) **Result Visualization:** After training, the epsilon decay curve and moving average of rewards are plotted. A final test run with $\epsilon = 0$ is performed, and the resulting path is visualized.

4. Parameter Tuning

After multiple rounds of experimentation and parameter tuning, the current set of values (**alpha** = 0.1, **gamma** = 0.9, **epsilon** = 1.0, **epsilon_decay** = 0.99, **epsilon_min** = 0.01, and **n_planning_steps** = 100) was determined to be a good solution for the Cliff-walking task. Training was conducted over 1000 episodes.

During the tuning process, we observe the impact of each hyperparameter. The **alpha** (learning rate) parameter dictates the convergence speed. An excessively large alpha caused Q-value estimates to oscillate and fail to converge, while a value that was too small resulted in slow learning. The **gamma** (discount factor) influences the agent's foresight. A value close to 1, such as 0.9, is crucial for this task, as it encourages the agent to value long-term rewards (reaching the exit) over short-term costs (the -1 for each step).

The **epsilon** decay schema manages the critical balance between exploration and exploitation. An initial **epsilon** of 1.0 ensures full exploration at the beginning of training. The **epsilon_decay** rate of 0.99 ensures a gradual transition, allowing the agent to exploit known good paths as training progresses. For Dyna-Q, the **n_planning_steps** parameter directly impacts training efficiency. A higher value allows the agent to learn more from each real experience, leading to much faster convergence in terms of episodes, at the cost of more computation per step.

The selected parameters achieve an effective balance between sufficient exploration to discover the optimal path, stable Q-value convergence, and an efficient learning process across all three implemented algorithms.

5. Result

Here are the Epsilon decay curves, average reward curves and final paths for the three algorithms:

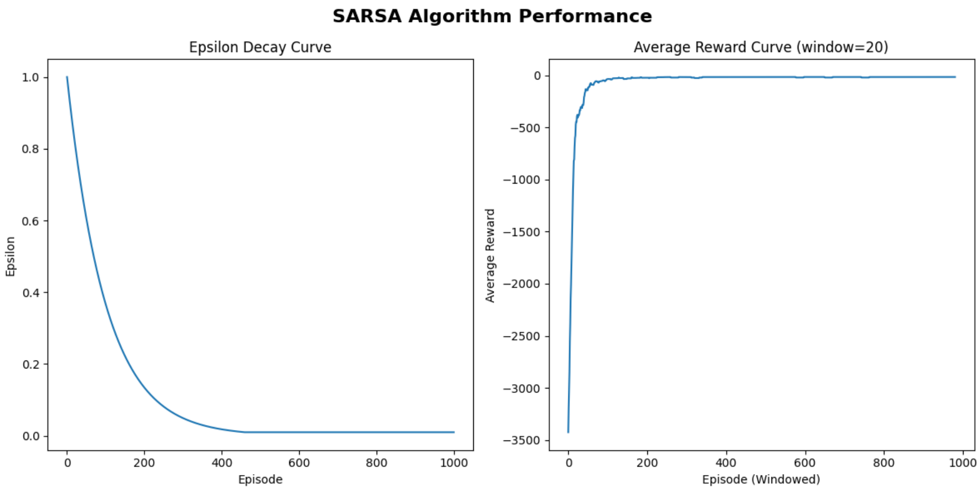

Figure 1: SARSA Performance

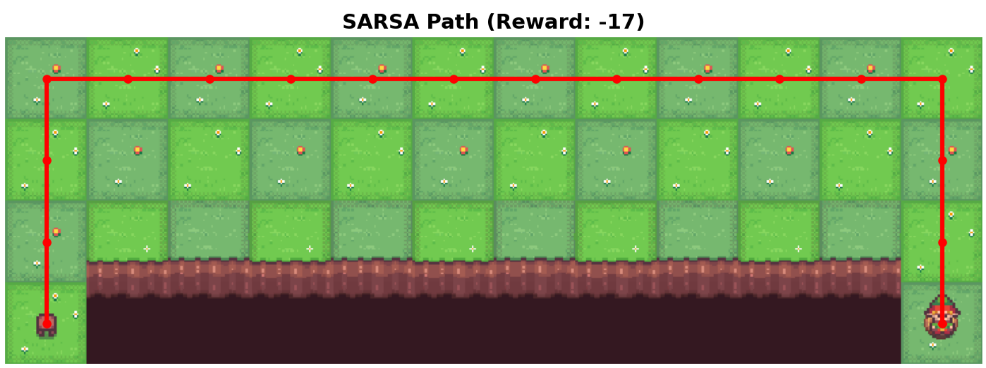

Figure 2: SARSA Path

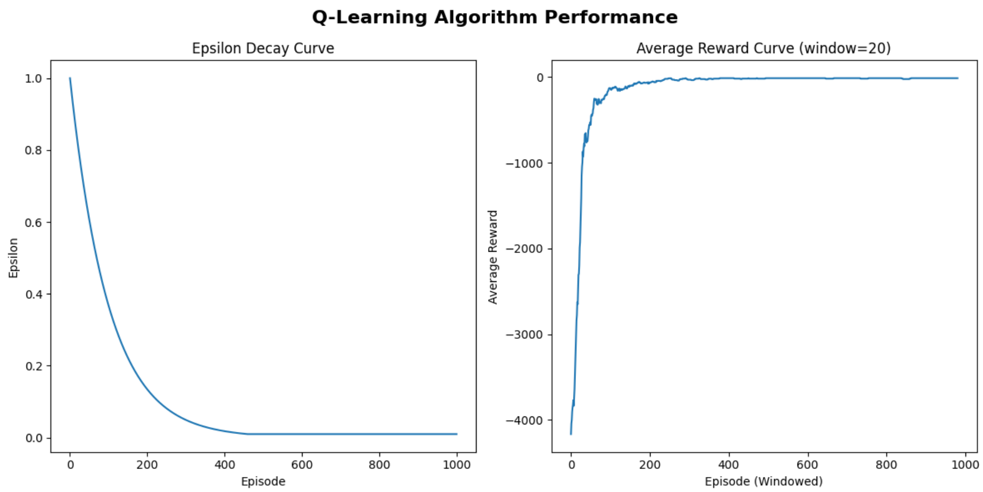

Figure 3: Qlearning Performance

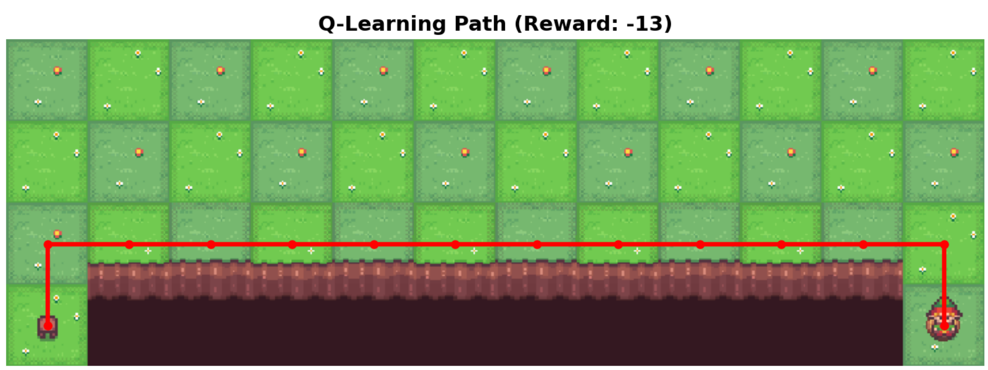

Figure 4: Qlearning Path

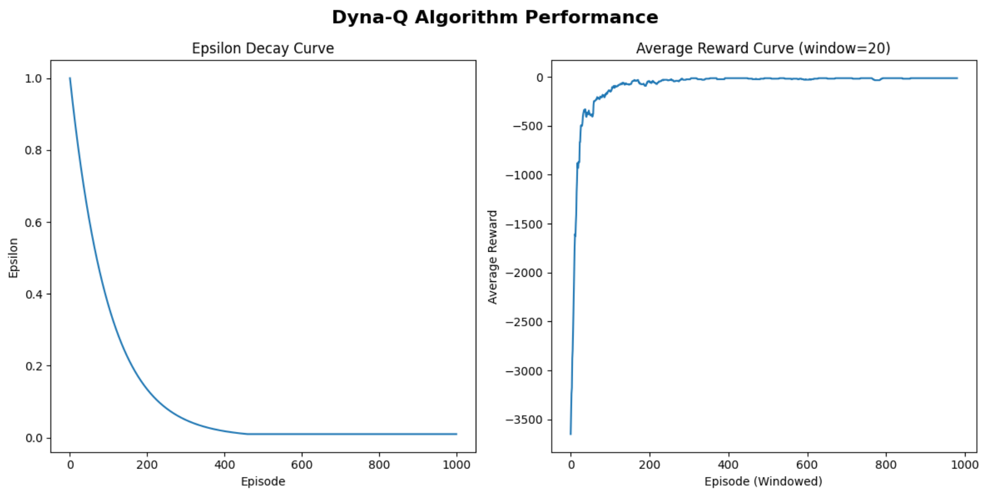

Figure 5: Dyna-Q Performance

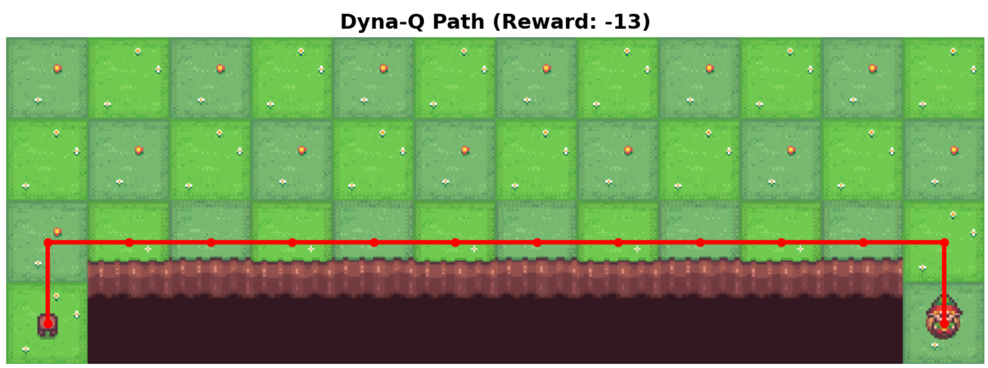

Figure 6: Dyna-Q Path

1. **Path difference between Sarsa and Q-learning:**

The paths generated by Sarsa and Q-learning exhibit notable differences due to their underlying learning strategies. Q-Learning finds the optimal path, which runs directly along the edge of the cliff with a reward of -13. While SARSA finds a "safer" and longer path that detours to the topmost row of the grid, far away from the cliff with a cost of -17.

This is because Q-learning is an off-policy algorithm that learns the optimal policy regardless of the agent's actions, leading it to exploit the cliff-edge path for maximum reward. In contrast, Sarsa is an on-policy algorithm that learns the value of the policy being followed, which includes the risk of falling into the cliff. As a result, Sarsa tends to favor safer paths that avoid high-risk areas, even if they are longer.

- SARSA(on-policy):Its update must account for the actual next action (a') chosen by its ϵ-greedy policy5. When on the cliff's edge, the ϵ-greedy policy has a non-zero chance of randomly selecting "down," incurring a massive -100 penalty. SARSA learns to factor in this exploration risk. It converges on a more "conservative" policy, preferring the longer path (more -1 penalties) to avoid the risk of being near the cliff.
- Q-Learning(off-policy):Its update is based on the theoretical best-possible next action ($\max a'$). It learns the value of the optimal policy without regard for the risks taken during exploration. Therefore, it finds the shortest path, ignoring the risk that the ϵ-greedy exploration policy might accidentally step off the cliff.

2. **Training efficiency between model-based RL (dyna-Q) and model-free alorithms (Sarsa or Q-learning)**

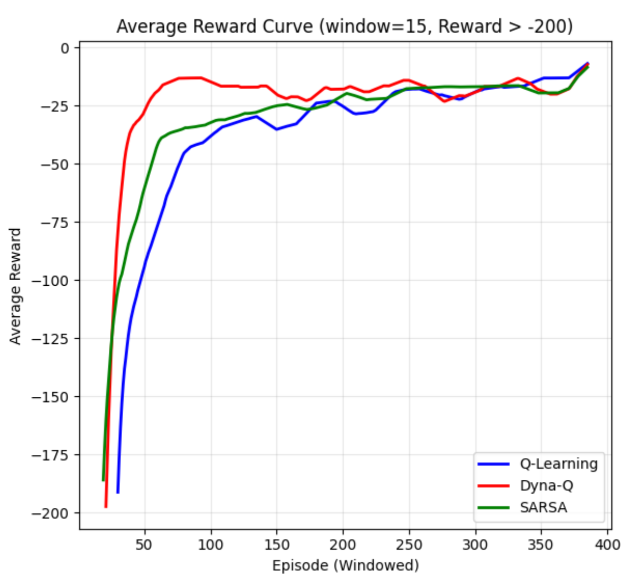

Figure 7: training efficiency comparison

As shown in the graph, Dyna-Q (red line) is significantly more training-efficient than both Q-Learning (blue line) and SARSA (green line). Dyna-Q's average reward curve rises the fastest, reaching a high-reward plateau within approximately 50-75 episodes. In contrast, SARSA and Q-Learning require 100-150 episodes to reach a similar level of performance.

This difference is mainly because Dyna-Q is a model-based algorithm, while SARSA and Q-Learning are model-free.

- Model-Free (Sarsa/Q-Learning): These agents get only one Q-value update for each step of real interaction with the environment, based on the single experience tuple (s, a, r, s').

- Model-Based (Dyna-Q): Dyna-Q simultaneously learns a model of the environment (i.e., $Model(s, a) \to r, s'$). For each single step of real experience, the Dyna-Q agent performs one direct Q-update (like Q-Learning) and additionally performs n planning steps (where $n = 100$ in the code). In these 100 planning steps, it uses its learned model to simulate experiences and updates its Q-table with these simulated experiences.

Therefore, Dyna-Q gets 101 learning opportunities per real interaction, while SARSA and Q-Learning gets only one. This allows Dyna-Q to squeeze much more learning out of each piece of real experience, leading to greater efficiency in terms of episodes (interactions).

B. Task 2: Deep Reinforcement Learning

1. Description

In this task, we implement a Deep Q-Network (DQN) agent to solve a more complex control problem: the "LunarLander-v2" gym environment. The goal is to control a spaceship and land it smoothly between two flags on the moon's surface. Unlike the previous task, the state space in this environment is continuous and high-dimensional (an 8-dimensional vector), which makes a tabular Q-table infeasible. Therefore, we use a neural network as a function approximator to estimate the action-value function $Q(s, a)$. The provided code, `dqn.py`, implements a DQN agent with enhancements such as a target network and experience replay.

2. Formulation

The core of the DQN algorithm is using a deep neural network to learn the optimal Q-value function.

1. **State (s_t):** This value is an 8-dimensional continuous vector representing the lander's physical status:

$$s_t = [x, y, v_x, v_y, \theta, \omega, leg_1, leg_2]$$

   where (x, y) are coordinates, (v_x, v_y) are linear velocities, θ is the angle, ω is the angular velocity, and (leg_1, leg_2) are booleans indicating ground contact for each leg.

2. **Action (a_t):**This is an integer $a_t \in \{0, 1, 2, 3\}$, representing four discrete actions: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine.

3. **Q-Value Approximation (Q-Network)** The action-value function $Q(s, a; \theta)$ is approximated by a neural network with parameters θ. The network architecture implemented in `dqn.py` is a Multi-Layer Perceptron (MLP):

   - **Input Layer:** 8 neurons (matching the state dimension)
   - **Hidden Layer 1:** 120 neurons with ReLU activation
   - **Hidden Layer 2:** 84 neurons with ReLU activation
   - **Output Layer:** 4 neurons (matching the action dimension), providing the Q-value for each possible action from the input state

4. **Loss Function and Update:** The network is trained by sampling a mini-batch of experiences (s, a, r, s', d) from a replay Buffer. The network's parameters θ are updated by minimizing the Mean Squared Error (MSE) loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s',d)\sim\text{Buffer}}\left[(y - Q(s, a; \theta))^2\right]$$

   where y is the TD target. Unlike standard DQN, the implementation in `dqn.py` (after we modified it) uses the **Double DQN** rule to calculate the target, which helps reduce overestimation of Q-values. Double DQN decouples action selection from action evaluation:

(a) The **main network** ($Q(s, a; \theta)$) is used to **select** the best action a^* in the next state s':

$$a^* = \arg\max_{a'} Q(s', a'; \theta)$$

(b) The **target network** ($Q(s, a; \theta')$) is used to **evaluate** the value of that action a^*.

The TD target y is therefore calculated as:

$$y = r + \gamma Q(s', a^*; \theta') = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta')$$

3. Implementation

a. **Core Data Structures:**

(a) **QNetwork Class:** A neural network class is defined using `torch.nn`. Each instance is a Multi-Layer Perceptron (MLP) with an 8-neuron input layer, two hidden layers (120 and 84 neurons) with ReLU activation, and a 4-neuron output layer.

(b) **Target Network:** A second instance of `QNetwork` is created, named `target_network`. Its weights are periodically synchronized with the main `q_network` (every `target_network_frequency` steps) to stabilize training.

(c) **Replay Buffer:** An instance of `ReplayBuffer` from the `stable_baselines3` library is implemented. It stores up to `buffer_size` (100,000) of past transitions (state, action, reward, next state, done flag).

(d) **Optimizer:** A `torch.optim.Adam` optimizer is initialized to update the parameters of the main `q_network`.

b. **Algorithm Execution Flow:**

(a) **Initialization:** Parse command-line arguments, set random seeds for reproducibility, create the Gym environment using `make_env`, and initialize the `q_network`, `target_network`, `optimizer`, and `replay_buffer`. A TensorBoard `SummaryWriter` is also set up for logging.

(b) **Main Loop:** The agent interacts with the environment in a single loop iterating for `total_timesteps` .

(c) **Epsilon Calculation:** At each global step, the exploration probability `epsilon` is calculated using a `linear_schedule` function. It decays linearly from `start_e` to `end_e` over the first `exploration_fraction` of total timesteps.

(d) **Action Selection:** An ϵ-greedy policy is used. A random number is compared to `epsilon`. If less, a random action is sampled (exploration). Otherwise, the main `q_network` predicts Q-values for the current state, and the action with the maximum Q-value is chosen (exploitation).

(e) **Environment Interaction:** The chosen action is passed to `envs.step()`, which returns the `next_obs`, `rewards`, `dones` flag, and `infos` dictionary.

(f) **Store Experience:** The transition tuple (`obs`, `next_obs`, `actions`, `rewards`, `dones`) is added to the `replay_buffer`.

(g) **State Transition:** The current observation `obs` is updated to `next_obs`. If the episode is `dones`, the environment is automatically reset.

(h) **Learning Trigger:** The learning process (Q-update) begins only after `global_step` exceeds `learning_starts` and executes every `train_frequency` steps.

(i) **Sample Batch:** A mini-batch of `batch_size` transitions is randomly sampled from the `replay_buffer`.

(j) **TD Target Calculation:** The TD target is calculated using the Double DQN rule. First, the main `q_network` is used to select the best action for the next state (`next_actions`). Then, the `target_network` is used to evaluate the Q-value of that selected action (`target_max`). The final target is computed as $y = r + \gamma \times \text{target\_max} \times (1 - d)$.

(k) **Loss Calculation:** The `F.mse_loss` (Mean Squared Error) is computed between the `td_target` (y) and the `old_val` (the Q-value of the action actually taken, predicted by the main `q_network`).

(l) **Optimization:** The optimizer's gradients are cleared (`zero_grad()`), the loss is backpropagated (`loss.backward()`), gradient clipping is applied using `clip_grad_norm_` to prevent exploding gradients, and the optimizer updates the `q_network` weights (`optimizer.step()`).

(m) **Target Network Update:** Every `target_network_frequency` steps, the `target_network`'s weights are synchronized by loading the state dictionary from the main `q_network`.

4. Parameter Tuning

After multiple rounds of experimentation and parameter tuning, the current set of values (**learning_rate** = 2.5e-4, **gamma** = 0.995, **buffer_size** = 100,000, **batch_size** = 256, and **target_network_frequency** = 500) was determined to be a near-optimal solution for the LunarLander-v2 task. The agent was trained for a total of 500,000 timesteps.

A high gamma value encourages the agent to prioritize long-term rewards (achieving a successful landing) over immediate, small rewards. The selected **batch_size** (256) offered a good balance between stability and computational speed. The **buffer_size** was set to 100,000 to ensure a large and diverse set of experiences.

The **epsilon** decay schema was tuned to start at 1.0 (full exploration) and decay linearly to 0.01 over 15% of the total timesteps. This extended exploration phase allows the agent to discover variable landing strategies before exploiting them. Finally, a **max_grad_norm** of 10.0 was applied to clip gradients, preventing gradient explosion and stabilizing the learning process.

The selected parameters achieve an excellent balance between stable learning, sufficient exploration, and efficient convergence, leading to a high-performance agent.

We use the average reward over 200 test episodes as the final performance metric.

In our experiments, by only tuning hyperparameters on the original dqn.py code (such as gamma, epsilon decay, etc.), the best average reward we achieved was approximately 220. Subsequently, by tuning parameters and optimizing the network structure, the average reward increased to around 254. Finally, building on the previous optimizations, we implemented key techniques such as Double DQN and gradient clipping. This approach yielded the best performance, reaching an average reward of approximately 276, and it is the solution we ultimately adopted. Additionally, we also attempted to use multi-frame state stacking instead of a single frame for training, but the results were not satisfactory.

We have placed a video demonstration in the ”videos” folder, which achieved a score of 281.45.

Note: You may find two versions of `dqn.py` in the submission: `dqn_original.py` (the original code provided) and `dqn.py` (our final modified version). That’s because the assignment requires to make comments on the original one, so I wrote comments for both of them. The results mentioned above are based on our modified version.

C. Task 3: Improve Exploration Schema

Beyond the ϵ-greedy strategy, **Upper Confidence Bound** (UCB) is a more efficient and directed exploration method.

1. *Idea*

The core idea of UCB is "Optimism in the face of uncertainty". Unlike ϵ-greedy, which performs completely random exploration, UCB explores based on the degree of uncertainty about the value of each action.

ϵ-greedy might randomly select an action it already knows is bad. UCB, however, selects the action that it is most uncertain about but has the potential to be the best.

The UCB algorithm balances "Exploitation" and "Exploration" using a formula. When selecting an action a in state s, it chooses the action that maximizes the following expression:

$$a_t = \arg\max_a \left[Q(s,a) + C\sqrt{\frac{\ln t}{N(s,a)}} \right]$$

1. $Q(s,a)$ **(Exploitation Term):** This is the agent's current estimated value of taking action a in state s. The higher this value, the more the agent wants to exploit this action.
2. $C\sqrt{\frac{\ln t}{N(s,a)}}$ **(Exploration Term):** This is the uncertainty bonus.
    - t is the total number of decision steps (or episodes) so far.
    - $N(s,a)$ is the number of times action a has been selected in state s.
    - C is a constant that balances the weight of exploitation and exploration.

How it works:

- If $N(s,a)$ is small (i.e., this action has been tried infrequently in this state), the denominator is small, causing the exploration term to become very large. This strongly encourages the agent to try this action it knows little about.
- If $N(s,a)$ is large (i.e., this action has been tried many times), the denominator is large, and the exploration term approaches 0. The choice will then be based primarily on the actual $Q(s,a)$ value.
- As the total steps t increases, $\ln t$ grows slowly, ensuring that the agent never completely stops exploring.

2. *Pros*

1. **Efficient and Directed Exploration:** UCB's exploration is not random but strategic. It prioritizes exploring actions with the highest "potential" (highest uncertainty) rather than wasting time on actions known to be suboptimal.

2. **No Epsilon Decay Tuning:** The effectiveness of ϵ-greedy heavily relies on the initial ϵ value and a complex decay schedule. UCB adjusts its exploration automatically based on visit counts, and the parameter C is relatively less sensitive.

3. Cons

1. **Difficult to Scale to Large State Spaces:** The form of UCB requires maintaining an exact visit count $N(s, a)$ for every state-action pair (s, a). This is feasible in tabular environments (like Task 1, Cliff-walking) but is impossible in environments with large or continuous state spaces (like Task 2, LunarLander-v2).

2. **Complex to Combine with Deep Learning:** In DQN, states are high-dimensional vectors, and a neural network cannot directly store $N(s, a)$. For example, s is not an integer, but rather an 8-dimensional vector of floating-point numbers (e.g., [0.123, -0.456, 0.789, ..., 1.0]). In two consecutive decisions, it is virtually impossible for the agent to visit the exact same floating-point state. To apply UCB's "optimism" in Deep RL, researchers must use very complex methods to estimate uncertainty, such as:

   - **Pseudo-Counts:** Using the novelty of states to estimate $N(s, a)$.
   - **Bayesian Neural Networks:** Using NNs to output a distribution over Q-values rather than a single point estimate, thereby quantifying uncertainty.

   These methods are all significantly more complex to implement than the ϵ-greedy strategy.

III. DISCUSSION AND CONCLUSION

A. Discussion of Findings

In **Task 1 (Cliff-walking)**, we observed the fundamental differences between RL algorithms. The Sarsa (on-policy) agent learned a ”safer” path (Reward: -17), while the Q-Learning (off-policy) agent found the optimal but riskier path along the cliff edge (Reward: -13). Furthermore, Dyna-Q (model-based) showed superior sample efficiency, converging in far fewer episodes than the model-free methods, as it used its learned model for 100 planning steps per real interaction.

In **Task 2 (LunarLander-v2)**, the high-dimensional continuous state space required a DQN. We tuned the agent, using the average reward over 200 test episodes as our metric. Our final solution, which achieved an average reward of 276, was built by applying both Double DQN and gradient clipping.

In **Task 3**, we analyzed UCB as an alternative exploration strategy. UCB relies on precise visit counts $N(s, a)$, which is infeasible when the state is a high-dimensional float vector where the exact same state is almost never revisited. This makes the simplicity of ϵ-greedy far more practical for DQN.

B. Conclusion

This report successfully demonstrated the trade-offs between on-policy (safer) and off-policy (more optimal) methods, as well as the sample efficiency gains of model-based (Dyna-Q) over model-free (Sarsa/Q-Learning) algorithms. For the complex Lunar Lander task, we found that modern enhancements like Double DQN and gradient clipping are essential for achieving stable, high-performance results.

IV. APPENDIX

附录 A: Source Code for SARSA:

```python
# -*- coding:utf-8 -*-
# Train Sarsa in cliff-walking environment
import math, os, time, sys
import numpy as np
import random
import gym
from agent import SarsaAgent

# construct the environment
env = gym.make("CliffWalking-v0")
# get the size of action space
num_actions = env.action_space.n
all_actions = np.arange(num_actions)
# set random seed and make the result reproducible
RANDOM_SEED = 0
env.seed(RANDOM_SEED)
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

####### START CODING HERE #######
num_states = env.observation_space.n

# construct the intelligent agent.
agent = SarsaAgent(
    all_actions=all_actions,
    state_dim=num_states,
    alpha=0.1,
    gamma=0.9,
    epsilon=1.0,
    epsilon_decay=0.99,
    epsilon_min=0.01,
)

episode_rewards = []
epsilon_values = []

# start training
for episode in range(1000):
    # record the reward in an episode
    episode_reward = 0
    # reset env
```

```
        s = env.reset()
        # agent interacts with the environment
        a = agent.choose_action(s)
        for iter in range(500):
            s_, r, isdone, info = env.step(a)
            a_ = agent.choose_action(s_)
            agent.learn(s, a, r, s_, a_)
            s = s_
            a = a_
            episode_reward += r
            if isdone:
                # time.sleep(0.1)
                break
        episode_rewards.append(episode_reward)
        epsilon_values.append(agent.epsilon)
        agent.decay_epsilon()

        if (episode + 1) % 50 == 0:
            print(
                "episode:",
                episode + 1,
                "episode_reward:",
                episode_reward,
                "epsilon:",
                agent.epsilon,
            )

    print("\ntraining over\n")

    # close the render window after training.
    env.close()

    import matplotlib.pyplot as plt
    plt.figure(figsize=(12, 6))
    plt.suptitle('SARSA Algorithm Performance', fontsize=16, fontweight='bold')
    plt.subplot(1, 2, 1)
    plt.plot(epsilon_values)
    plt.title('Epsilon Decay Curve')
    plt.xlabel('Episode')
    plt.ylabel('Epsilon')

    def moving_average(data, window_size=20):
        return np.convolve(data, np.ones(window_size)/window_size, mode='valid')

    avg_rewards = moving_average(episode_rewards)
```

```python
    plt.subplot(1, 2, 2)
    plt.plot(avg_rewards)
    plt.title(f'Average Reward Curve (window={20})')
    plt.xlabel('Episode (Windowed)')
    plt.ylabel('Average Reward')

    plt.tight_layout()
    plt.show()

    s = env.reset()
    agent.epsilon = 0.0
    episode_reward = 0
    isdone = False
    path = [s]

    while not isdone:
        env.render()
        time.sleep(0.3)
        a = agent.choose_action(s)
        s, r, isdone, info = env.step(a)
        path.append(s)
        episode_reward += r

    print(f"Test complete! Final path reward: {episode_reward}")

    img = env.render(mode='rgb_array')
    env.close()
    plt.figure(figsize=(12, 4))
    plt.imshow(img)
    coords = [(divmod(s, 12)[1] * img.shape[1] / 12 + img.shape[1] / 24,
               divmod(s, 12)[0] * img.shape[0] / 4 + img.shape[0] / 8) for s in path]
    plt.plot([c[0] for c in coords], [c[1] for c in coords], 'r-', linewidth=3, marker='o', markersize=5)
    plt.title(f'SARSA Path (Reward: {episode_reward})', fontsize=14, fontweight='bold')
    plt.axis('off')
    plt.savefig('sarsa_path.png', dpi=150, bbox_inches='tight')
    plt.show()
    ####### END CODING HERE #######
```

Listing 1: Source Code for SARSA (cliff_walk_sarsa.py)

附录 B: Source Code for Q-Learning:

```
# -*- coding:utf-8 -*-
# Train Q-Learning in cliff-walking environment
import math, os, time, sys
import numpy as np
import random
import gym
from agent import QLearningAgent
##### START CODING HERE #####
# This code block is optional. You can import other libraries or define your
    utility functions if necessary.

##### END CODING HERE #####

# construct the environment
env = gym.make("CliffWalking-v0")
# get the size of action space
num_actions = env.action_space.n
all_actions = np.arange(num_actions)
# set random seed and make the result reproducible
RANDOM_SEED = 0
env.seed(RANDOM_SEED)
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

##### START CODING HERE #####
num_states = env.observation_space.n
# construct the intelligent agent.
agent = QLearningAgent(
    all_actions=all_actions,
    state_dim=num_states,
    alpha=0.1,
    gamma=0.9,
    epsilon=1.0,
    epsilon_decay=0.99,
    epsilon_min=0.01,
)

episode_rewards = []
epsilon_values = []

# start training
for episode in range(1000):
    # record the reward in an episode
```

```
    episode_reward = 0
    # reset env
    s = env.reset()

    # agent interacts with the environment
    for iter in range(500):
        # choose an action
        a = agent.choose_action(s)
        s_, r, isdone, info = env.step(a)
        agent.learn(s, a, r, s_)
        s= s_
        # update the episode reward
        episode_reward += r
        if isdone:
            time.sleep(0.1)
            break
    episode_rewards.append(episode_reward)
    epsilon_values.append(agent.epsilon)
    agent.decay_epsilon()

    if (episode + 1) % 50 == 0:
        print(
            "episode:",
            episode + 1,
            "episode_reward:",
            episode_reward,
            "epsilon:",
            agent.epsilon,
        )
print('\ntraining over\n')

# close the render window after training.
env.close()


import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
plt.suptitle('Q-Learning Algorithm Performance', fontsize=16, fontweight='bold')
plt.subplot(1, 2, 1)
plt.plot(epsilon_values)
plt.title('Epsilon Decay Curve')
plt.xlabel('Episode')
plt.ylabel('Epsilon')

def moving_average(data, window_size=20):
```

```
        return np.convolve(data, np.ones(window_size)/window_size, mode='valid')

    avg_rewards = moving_average(episode_rewards)
    plt.subplot(1, 2, 2)
    plt.plot(avg_rewards)
    plt.title(f'Average Reward Curve (window={20})')
    plt.xlabel('Episode (Windowed)')
    plt.ylabel('Average Reward')

    plt.tight_layout()
    plt.show()

    s = env.reset()
    agent.epsilon = 0.0
    episode_reward = 0
    isdone = False
    path = [s]

    while not isdone:
        env.render()
        time.sleep(0.3)
        a = agent.choose_action(s)
        s, r, isdone, info = env.step(a)
        path.append(s)
        episode_reward += r

    print(f"Test complete! Final path reward: {episode_reward}")

    img = env.render(mode='rgb_array')
    env.close()
    plt.figure(figsize=(12, 4))
    plt.imshow(img)
    coords = [(divmod(s, 12)[1] * img.shape[1] / 12 + img.shape[1] / 24,
               divmod(s, 12)[0] * img.shape[0] / 4 + img.shape[0] / 8) for s in path]
    plt.plot([c[0] for c in coords], [c[1] for c in coords], 'r-', linewidth=3, marker='o', markersize=5)
    plt.title(f'Q-Learning Path (Reward: {episode_reward})', fontsize=14, fontweight='bold')
    plt.axis('off')
    plt.savefig('qlearning_path.png', dpi=150, bbox_inches='tight')
    plt.show()
    ####### END CODING HERE #######
```

Listing 2: Source Code for Q-learning (cliff_walk_qlearning.py)

附录 C: Source Code for Dyna-Q:

```
# -*- coding:utf-8 -*-
# Train Q-Learning in cliff-walking environment
import math, os, time, sys
import numpy as np
import random
import gym
from agent import Dyna_QAgent

# construct the environment
env = gym.make("CliffWalking-v0")
# get the size of action space
num_actions = env.action_space.n
all_actions = np.arange(num_actions)
# set random seed and make the result reproducible
RANDOM_SEED = 0
env.seed(RANDOM_SEED)
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

##### START CODING HERE #####
num_states = env.observation_space.n

# construct the intelligent agent.
agent = Dyna_QAgent(
    all_actions=all_actions,
    state_dim=num_states,
    alpha=0.1,
    gamma=0.9,
    epsilon=1.0,
    epsilon_decay=0.99,
    epsilon_min=0.01,
    n_planning_steps=100,
)

episode_rewards = []
epsilon_values = []

# start training
for episode in range(1000):
    # record the reward in an episode
    episode_reward = 0
    # reset env
    s = env.reset()
```

```python

            # agent interacts with the environment
            for iter in range(500):
                # choose an action
                a = agent.choose_action(s)
                s_, r, isdone, info = env.step(a)
                agent.learn(s, a, r, s_)
                s= s_
                # update the episode reward
                episode_reward += r
                if isdone:
                    time.sleep(0.1)
                    break
            episode_rewards.append(episode_reward)
            epsilon_values.append(agent.epsilon)
            agent.decay_epsilon()

            if (episode + 1) % 50 == 0:
                print(
                    "episode:",
                    episode + 1,
                    "episode_reward:",
                    episode_reward,
                    "epsilon:",
                    agent.epsilon,
                )
        print('\ntraining over\n')

        # close the render window after training.
        env.close()

        import matplotlib.pyplot as plt
        plt.figure(figsize=(12, 6))
        plt.suptitle('Dyna-Q Algorithm Performance', fontsize=16, fontweight='bold')
        plt.subplot(1, 2, 1)
        plt.plot(epsilon_values)
        plt.title('Epsilon Decay Curve')
        plt.xlabel('Episode')
        plt.ylabel('Epsilon')

        def moving_average(data, window_size=20):
            return np.convolve(data, np.ones(window_size)/window_size, mode='valid')

        avg_rewards = moving_average(episode_rewards)
        plt.subplot(1, 2, 2)
```

```python
        plt.plot(avg_rewards)
        plt.title(f'Average Reward Curve (window={20})')
        plt.xlabel('Episode (Windowed)')
        plt.ylabel('Average Reward')

        plt.tight_layout()
        plt.show()



        s = env.reset()
        agent.epsilon = 0.0
        episode_reward = 0
        isdone = False
        path = [s]

        while not isdone:
            env.render()
            time.sleep(0.3)
            a = agent.choose_action(s)
            s, r, isdone, info = env.step(a)
            path.append(s)
            episode_reward += r

        print(f"Test complete! Final path reward: {episode_reward}")

        img = env.render(mode='rgb_array')
        env.close()
        plt.figure(figsize=(12, 4))
        plt.imshow(img)
        coords = [(divmod(s, 12)[1] * img.shape[1] / 12 + img.shape[1] / 24,
                   divmod(s, 12)[0] * img.shape[0] / 4 + img.shape[0] / 8) for s in path]
        plt.plot([c[0] for c in coords], [c[1] for c in coords], 'r-', linewidth=3, marker='o', markersize=5)
        plt.title(f'Dyna-Q Path (Reward: {episode_reward})', fontsize=14, fontweight='bold')
        plt.axis('off')
        plt.savefig('dyna_q_path.png', dpi=150, bbox_inches='tight')
        plt.show()
        ####### END CODING HERE #######
```

Listing 3: Source Code for Dyna-Q (cliff_walk_dyna_q.py)

附录 D: Source Code for DQN:

```python
# -*- coding:utf-8 -*-
import argparse
import os
import random
import time

import gym
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from stable_baselines3.common.buffers import ReplayBuffer
from torch.utils.tensorboard import SummaryWriter

def parse_args():
    """parse arguments. You can add other arguments if needed."""
    parser = argparse.ArgumentParser()
    parser.add_argument("--exp-name", type=str, default=os.path.basename(__file__).rstrip(".py"),
        help="the␣name␣of␣this␣experiment")
    parser.add_argument("--seed", type=int, default=42,
        help="seed␣of␣the␣experiment")
    parser.add_argument("--total-timesteps", type=int, default=500000,
        help="total␣timesteps␣of␣the␣experiments")
    parser.add_argument("--learning-rate", type=float, default=2.5e-4,
        help="the␣learning␣rate␣of␣the␣optimizer")
    parser.add_argument("--buffer-size", type=int, default=100000,
        help="the␣replay␣memory␣buffer␣size")
    parser.add_argument("--gamma", type=float, default=0.995,
        help="the␣discount␣factor␣gamma")
    parser.add_argument("--target-network-frequency", type=int, default=500,
        help="the␣timesteps␣it␣takes␣to␣update␣the␣target␣network")
    parser.add_argument("--batch-size", type=int, default=256,
        help="the␣batch␣size␣of␣sample␣from␣the␣reply␣memory")
    parser.add_argument("--start-e", type=float, default=1.0,
        help="the␣starting␣epsilon␣for␣exploration")
    parser.add_argument("--end-e", type=float, default=0.01,
        help="the␣ending␣epsilon␣for␣exploration")
    parser.add_argument("--exploration-fraction", type=float, default=0.15,
        help="the␣fraction␣of␣`total-timesteps`␣it␣takes␣from␣start-e␣to␣go␣end-e")
    parser.add_argument("--learning-starts", type=int, default=5000,
```

```
            help="timestep␣to␣start␣learning")
        parser.add_argument("--train-frequency", type=int, default=4,
            help="the␣frequency␣of␣training")
        parser.add_argument("--max-grad-norm", type=float, default=10.0,
            help="the␣maximum␣norm␣for␣gradient␣clipping")
        args = parser.parse_args()
        args.env_id = "LunarLander-v2"
        return args

    def make_env(env_id, seed):
        """construct the gym environment"""
        env = gym.make(env_id)
        env = gym.wrappers.RecordEpisodeStatistics(env)
        env.seed(seed)
        env.action_space.seed(seed)
        env.observation_space.seed(seed)
        return env

    class QNetwork(nn.Module):
        """
        comments:

        the neural network model for approximating Q value function

        Inputs:State
        Outputs: Q-values for each possible action in that state.

        Here:
        Input layer: 8 (state dimension)
        Hidden layer 1: 120 neurons, ReLU activation
        Hidden layer 2: 84 neurons, ReLU activation
        Output layer: 4 (action dimension)
        """
        def __init__(self, env):
            super().__init__()
            self.network = nn.Sequential(
                nn.Linear(np.array(env.observation_space.shape).prod(), 120),
                nn.ReLU(),
                nn.Linear(120, 84),
                nn.ReLU(),
                nn.Linear(84, env.action_space.n),
            )

        def forward(self, x):
            return self.network(x)
```

```python

def linear_schedule(start_e: float, end_e: float, duration: int, t: int):
    """
    comments:

    Implements a linear decay for epsilon ( ) as part of the  -greedy strategy.
    - start_e: The initial value of epsilon
    - end_e: The final value of epsilon
    - duration: The total number of timesteps over which to decay from start_e to end_e.
    - t: The current timestep.

    When t >= duration, epsilon will be equal to end_e.
    When t < duration, epsilon will decrease linearly from start_e to end_e.
    """
    slope = (end_e - start_e) / duration
    return max(slope * t + start_e, end_e)

if __name__ == "__main__":

    """parse the arguments"""
    args = parse_args()
    run_name = f"{args.env_id}__{args.exp_name}__{args.seed}__{int(time.time())}"

    """we utilize tensorboard yo log the training process"""
    writer = SummaryWriter(f"runs/{run_name}")
    writer.add_text(
        "hyperparameters",
        "|param|value|\n|-|-|\n%s" % ("\n".join([f"|{key}|{value}|" for key, value in vars(args).items()])),
    )

    """
    comments:
    set the random seed to make the experiment reproducible(for numpy, torch, gym, random)
    check whether cuda is available, if available, use cuda to accelerate the training, else use cpu
    """
    random.seed(args.seed)
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    torch.backends.cudnn.deterministic = True
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

```python
    """
    comments:
    create the gym environment
    envs: the vectorized environment
    """
    envs = make_env(args.env_id, args.seed)

    """
    comments:
    Initialize the Q-network, target network, and optimizer.

    The target network is a copy of the Q-network and is used to stabilize
        training.
    - q_network (Main): Used for action selection (exploitation) and is the
        network that gets updated by the optimizer.
    - target_network: Used to calculate the TD Target value. Its weights are
        frozen.It stabilizes training.
    - Adam optimizer: Train the parameters of the q_network.
    """
    q_network = QNetwork(envs).to(device)
    optimizer = optim.Adam(q_network.parameters(), lr=args.learning_rate)
    target_network = QNetwork(envs).to(device)
    target_network.load_state_dict(q_network.state_dict())

    """
    comments:
    Initialize the Replay Buffer.
    - DQN uses a replay buffer to store past experiences (s, a, r, s', d).
    Sampling random batches from the buffer breaks the correlation between
        consecutive samples, stabilizing training.
    """
    rb = ReplayBuffer(
        args.buffer_size,
        envs.observation_space,
        envs.action_space,
        device,
        handle_timeout_termination=False,
    )

    """
    comments:
    Initialize the environment by resetting it and getting the first observation (
        state).
    """
    obs = envs.reset()
```

```python
        for global_step in range(args.total_timesteps):

            """
            comments:
            Calculate the current value of Epsilon.
            Uses the linear_schedule function based on the current global_step to determine the probability of exploration.
            """
            epsilon = linear_schedule(args.start_e, args.end_e, args.exploration_fraction * args.total_timesteps, global_step)

            """
            comments:
            Epsilon-Greedy ( -greedy) Action Selection.
            - With probability 'epsilon', choose a random action (exploration).
            - With probability '1-epsilon', choose the action with the highest predicted Q-value from the main q_network (exploitation).
            """
            if random.random() < epsilon:
                actions = envs.action_space.sample()
            else:
                q_values = q_network(torch.Tensor(obs).to(device))
                actions = torch.argmax(q_values, dim=0).cpu().numpy()

            """
            comments:
            Interact with the environment.
            Inputs:
                - actions: The actions chosen by the agent based on the  -greedy strategy.
            Outputs:
                - next_obs (s'): The next state.
                - rewards (r): The immediate reward.
                - dones (d): whether the episode has ended.
                - infos: extra info .
            """
            next_obs, rewards, dones, infos = envs.step(actions)
            # envs.render() # close render during training

            if dones:
                print(f"global_step={global_step}, episodic_return={infos['episode']['r']}")
                writer.add_scalar("charts/episodic_return", infos["episode"]["r"], global_step)
                writer.add_scalar("charts/episodic_length", infos["episode"]["l"],
```

```
                    global_step)

        """
        comments:
        Store the transition (s, a, r, s', d) in the replay buffer.
        """
        rb.add(obs, next_obs, actions, rewards, dones, infos)

        """
        comments:
        Update the current observation to the next observation.
        """
        obs = next_obs if not dones else envs.reset()

        if global_step > args.learning_starts and global_step % args.train_frequency == 0:

            """
            comments:
            Sample a random batch of experiences from the Replay Buffer.
            """
            data = rb.sample(args.batch_size)

            """
            comments:
            Calculate the TD Target value using Double DQN to reduce overestimation.

            Standard DQN:  y_j = r +   * max_a' Q_target(s', a')
            Double DQN:    y_j = r +   * Q_target(s', argmax_a' Q(s', a'))

            Double DQN decouples action selection and evaluation:
            - Use main Q-network to SELECT the best action
            - Use target network to EVALUATE that action
            This reduces the overestimation bias of standard DQN.
            """
            with torch.no_grad():
                # use online network to select actions
                next_q_values = q_network(data.next_observations)
                next_actions = next_q_values.argmax(dim=1, keepdim=True)
                # Use target network to evaluate the selected actions
                next_q_target_values = target_network(data.next_observations)
                target_max = next_q_target_values.gather(1, next_actions).squeeze()

```

```python
                td_target = data.rewards.flatten() + args.gamma * target_max * (1 - data.dones.flatten())
            old_val = q_network(data.observations).gather(1, data.actions).squeeze()
            loss = F.mse_loss(td_target, old_val)

            """
            comments:
            Log the loss and average Q-value to TensorBoard to monitor the training process.
            """
            if global_step % 100 == 0:
                writer.add_scalar("losses/td_loss", loss, global_step)
                writer.add_scalar("losses/q_values", old_val.mean().item(), global_step)

            """
            comments:
            Perform backpropagation and update the network.
            Apply gradient clipping to prevent gradient explosion.
            """
            optimizer.zero_grad()
            loss.backward()
            nn.utils.clip_grad_norm_(q_network.parameters(), args.max_grad_norm)
            optimizer.step()

            """
            comments:
            Periodically update the target network to match the weights of the main q_network.
            """
            if global_step % args.target_network_frequency == 0:
                target_network.load_state_dict(q_network.state_dict())

    """close the env and tensorboard logger"""
    envs.close()
    writer.close()
```

Listing 4: Source Code for DQN (dqn.py)