HW6 - Discrete environment / algorithm

Arian mohammadkhani - 810603136

In [1]:
```
import gymnasium as gym
import numpy as np
import random
from collections import deque
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import time
import seaborn as sns
import pandas as pd
```

Custom MountainCar Environment (Discrete Actions):

In [2]:
```
class SimpleMountainCarEnv:
    def __init__(self):
        self.position_bounds = [-1.2, 0.6]
        self.velocity_bounds = [-0.07, 0.07]
        self.force = 0.001
        self.gravity = 0.0025
        self.max_steps = 200
        self.reset()

    def reset(self):
        self.position = np.random.uniform(-0.6, -0.4)
        self.velocity = 0.0
        self.steps = 0
        return np.array([self.position, self.velocity])

    def step(self, action):
        force_effect = (action - 1) * self.force
        gravity_effect = -np.cos(3 * self.position) * self.gravity

        self.velocity += force_effect + gravity_effect
        self.velocity = np.clip(self.velocity, *self.velocity_bounds)

        self.position += self.velocity
        self.position = np.clip(self.position, *self.position_bounds)

        if self.position in self.position_bounds:
            self.velocity = 0.0

        self.steps += 1
        done = False
        reward = -1

        if self.position >= 0.5:
            done = True
            reward = 1

        if self.steps >= self.max_steps:
            done = True

        return np.array([self.position, self.velocity]), reward, done
```

State Discretization and Utility Functions:

In [3]:
```
n_position_buckets = 30
n_velocity_buckets = 30

def discretize_state(state):
    position, velocity = state
    pos_scale = (position - (-1.2)) / (0.6 - (-1.2))
    vel_scale = (velocity - (-0.07)) / (0.07 - (-0.07))

    pos_bucket = int(pos_scale * (n_position_buckets - 1))
    vel_bucket = int(vel_scale * (n_velocity_buckets - 1))

    pos_bucket = min(n_position_buckets - 1, max(0, pos_bucket))
    vel_bucket = min(n_velocity_buckets - 1, max(0, vel_bucket))

    return (pos_bucket, vel_bucket)

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

Q-Learning Training Function:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
    env = SimpleMountainCarEnv()
```

```
    q_table = np.zeros((n_position_buckets, n_velocity_buckets, 3))
    episode_rewards = []

    alpha = 0.1
    gamma = 0.99
    epsilon = 1.0
    epsilon_decay = 0.999
    epsilon_min = 0.05
    n_episodes = 10000

    start_time = time.time()

    for episode in range(n_episodes):
        state_continuous = env.reset()
        state_disc = discretize_state(state_continuous)
        total_reward = 0
        done = False

        while not done:
            if np.random.random() < epsilon:
                action = np.random.choice([0, 1, 2])
            else:
                action = np.argmax(q_table[state_disc])

            next_state_continuous, reward, done = env.step(action)
            next_state_disc = discretize_state(next_state_continuous)

            best_next_action = np.argmax(q_table[next_state_disc])
            td_target = reward + gamma * q_table[next_state_disc + (best_next_action,)]
            td_error = td_target - q_table[state_disc + (action,)]
            q_table[state_disc + (action,)] += alpha * td_error

            state_disc = next_state_disc
            total_reward += reward

        episode_rewards.append(total_reward)

        if epsilon > epsilon_min:
            epsilon *= epsilon_decay

        if (episode + 1) % 1000 == 0:
            print(f"Q-learning Episode {episode + 1}, Total Reward: {total_reward}, Epsilon: {epsilon:.3f}")

    end_time = time.time()
    print(f"Q-learning Training finished! Time elapsed: {end_time - start_time:.2f} seconds")
    return q_table, episode_rewards
```

Evaluating Q-Learning Performance (Visual & Statistical):

In [5]:

```
def evaluate_q_learning_visual(q_table):
    env_eval = gym.make("MountainCar-v0", render_mode="human")
    state, _ = env_eval.reset()
    state_disc = discretize_state(state)
    done = False
    total_reward = 0

    while not done:
        action = np.argmax(q_table[state_disc])
        next_state, reward, terminated, truncated, _ = env_eval.step(action)
        done = terminated or truncated
        state_disc = discretize_state(next_state)
        total_reward += reward
        env_eval.render()
        time.sleep(0.01)

    print(f"\nFinal Evaluation Total Reward (Q-learning): {total_reward}")
    env_eval.close()

def evaluate_q_learning(q_table, n_runs=10):
    env = gym.make("MountainCar-v0")
    steps_list = []

    for _ in range(n_runs):
        state = env.reset()
        state = state[0] if isinstance(state, tuple) else state
        state_disc = discretize_state(state)
        done = False
        steps = 0

        while not done and steps < 200:
            action = np.argmax(q_table[state_disc])
            result = env.step(action)

            if len(result) == 5:
                next_state, _, terminated, truncated, _ = result
                done = terminated or truncated
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

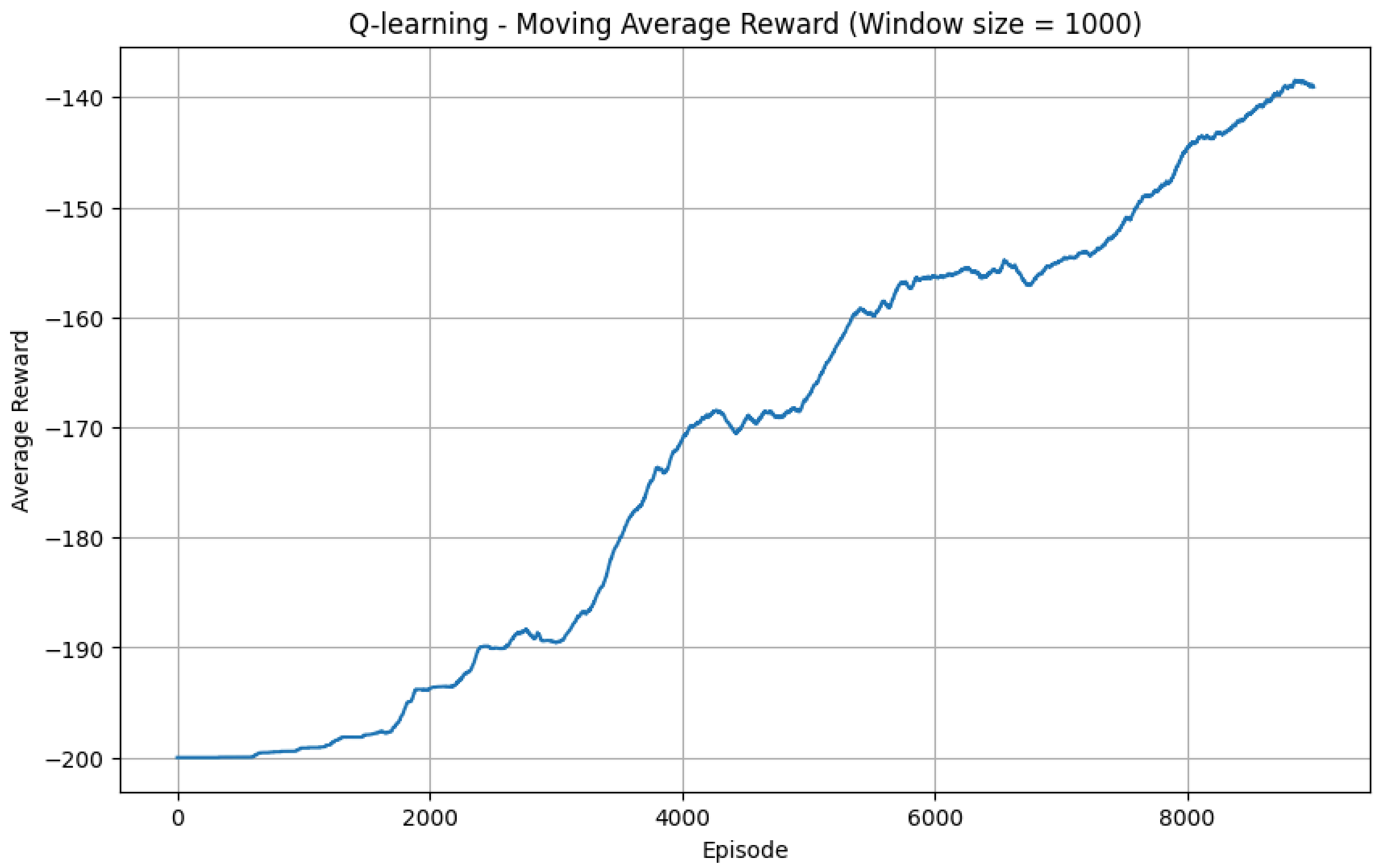
```
                next_state, _, done, _ = result
```

```
            state_disc = discretize_state(next_state)
            steps += 1

        steps_list.append(steps)

    env.close()
    return np.mean(steps_list)
```

Run Q-Learning Training and Evaluation:

In [6]:
```
if __name__ == "__main__":
    start_time = time.time()
    q_table, reward_history = train_q_learning()
    time_q = time.time() - start_time
    evaluate_q_learning_visual(q_table)
    avg_steps = evaluate_q_learning(q_table)
    window = 1000
    moving_avg = np.convolve(reward_history, np.ones(window)/window, mode='valid')
    plt.figure(figsize=(10, 6))
    plt.plot(moving_avg)
    plt.title(f"Q-learning - Moving Average Reward (Window size = {window})")
    plt.xlabel("Episode")
    plt.ylabel("Average Reward")
    plt.grid(True)
    plt.show()
    print(f"\nAverage steps over evaluation runs: {avg_steps}")
```

```
Q-learning Episode 1000, Total Reward: -200, Epsilon: 0.368
Q-learning Episode 2000, Total Reward: -200, Epsilon: 0.135
Q-learning Episode 3000, Total Reward: -153, Epsilon: 0.050
Q-learning Episode 4000, Total Reward: -175, Epsilon: 0.050
Q-learning Episode 5000, Total Reward: -153, Epsilon: 0.050
Q-learning Episode 6000, Total Reward: -146, Epsilon: 0.050
Q-learning Episode 7000, Total Reward: -152, Epsilon: 0.050
Q-learning Episode 8000, Total Reward: -156, Epsilon: 0.050
Q-learning Episode 9000, Total Reward: -153, Epsilon: 0.050
Q-learning Episode 10000, Total Reward: -157, Epsilon: 0.050
Q-learning Training finished! Time elapsed: 24.73 seconds

Final Evaluation Total Reward (Q-learning): -164.0
```




```
Average steps over evaluation runs: 141.3
```

Double Q-Learning Training Function:

In [7]:
```
def train_double_q_learning():
    q_table_1 = np.zeros((n_position_buckets, n_velocity_buckets, 3))
    q_table_2 = np.zeros((n_position_buckets, n_velocity_buckets, 3))
    episode_rewards = []

    alpha = 0.1
    gamma = 0.99
    epsilon = 1.0
    epsilon_decay = 0.995
    epsilon_min = 0.1
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```python
    env = SimpleMountainCarEnv()
    start_time = time.time()

    for episode in range(n_episodes):
        state_continuous = env.reset()
        state_disc = discretize_state(state_continuous)
        total_reward = 0
        done = False

        while not done:
            if np.random.random() < epsilon:
                action = np.random.choice([0, 1, 2])
            else:
                q_sum = q_table_1[state_disc] + q_table_2[state_disc]
                action = np.argmax(q_sum)

            next_state_continuous, reward, done = env.step(action)
            next_state_disc = discretize_state(next_state_continuous)

            if np.random.random() < 0.5:
                best_next_action = np.argmax(q_table_1[next_state_disc])
                td_target = reward + gamma * q_table_2[next_state_disc + (best_next_action,)]
                td_error = td_target - q_table_1[state_disc + (action,)]
                q_table_1[state_disc + (action,)] += alpha * td_error
            else:
                best_next_action = np.argmax(q_table_2[next_state_disc])
                td_target = reward + gamma * q_table_1[next_state_disc + (best_next_action,)]
                td_error = td_target - q_table_2[state_disc + (action,)]
                q_table_2[state_disc + (action,)] += alpha * td_error

            state_disc = next_state_disc
            total_reward += reward

        episode_rewards.append(total_reward)

        if epsilon > epsilon_min:
            epsilon *= epsilon_decay

        if (episode + 1) % 1000 == 0:
            print(f"Double Q-learning Episode {episode + 1}, Total Reward: {total_reward}, Epsilon: {epsilon:.3f}")

    end_time = time.time()
    print(f"Double Q-learning Training finished! Time elapsed: {end_time - start_time:.2f} seconds")
    return q_table_1, q_table_2, episode_rewards
```

Q-Learning Evaluation Functions:

In [8]:

```python
def evaluate_double_q_learning(q_table_1, q_table_2):
    env_eval = gym.make("MountainCar-v0", render_mode="human")
    state, _ = env_eval.reset()
    state_disc = discretize_state(state)
    done = False
    total_reward = 0

    while not done:
        q_sum = q_table_1[state_disc] + q_table_2[state_disc]
        action = np.argmax(q_sum)

        next_state, reward, terminated, truncated, _ = env_eval.step(action)
        done = terminated or truncated
        state_disc = discretize_state(next_state)
        total_reward += reward
        env_eval.render()
        time.sleep(0.01)

    print(f"\nFinal Evaluation Total Reward (Double Q-learning): {total_reward}")
    env_eval.close()
    return total_reward
```

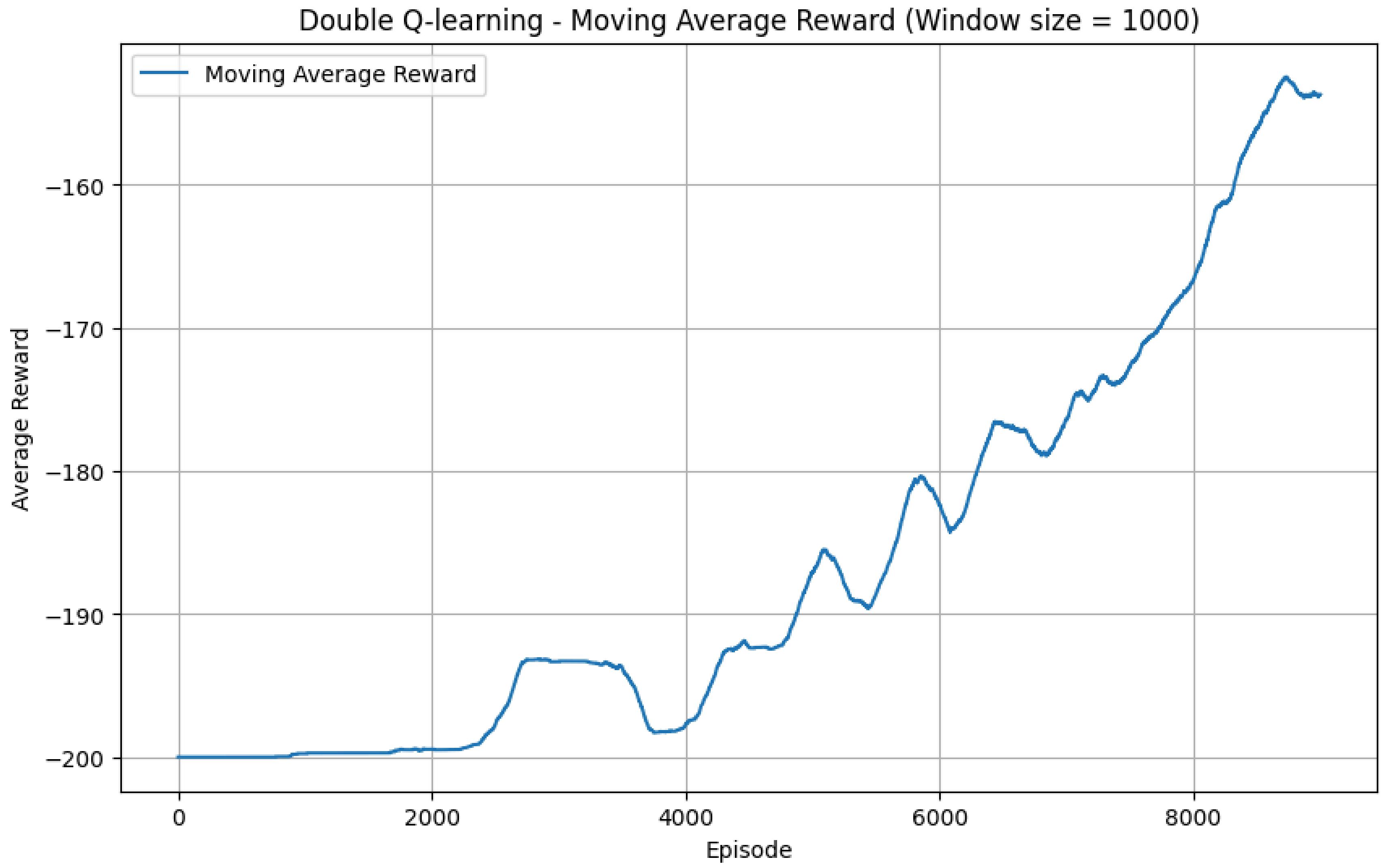
Run Double Q-Learning Training and Evaluation:

In [9]:

```python
if __name__ == "__main__":
    start_time = time.time()
    q1, q2, double_q_rewards = train_double_q_learning()
    time_dq = time.time() - start_time
    avg_reward_dq = evaluate_double_q_learning(q1, q2)
    window = 1000
    moving_avg_dq = np.convolve(double_q_rewards, np.ones(window)/window, mode='valid')
    plt.figure(figsize=(10, 6))
    plt.plot(moving_avg_dq, label="Moving Average Reward")
    plt.title(f"Double Q-learning - Moving Average Reward (Window size = {window})")
    plt.xlabel("Episode")
    plt.ylabel("Average Reward")
    plt.grid(True)
    plt.legend()
    plt.show()
    print(f"Average reward in final evaluation: {avg_reward_dq}")
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
Double Q-learning Episode 1000, Total Reward: -200, Epsilon: 0.100
Double Q-learning Episode 2000, Total Reward: -200, Epsilon: 0.100
Double Q-learning Episode 3000, Total Reward: -200, Epsilon: 0.100
Double Q-learning Episode 4000, Total Reward: -200, Epsilon: 0.100
Double Q-learning Episode 5000, Total Reward: -156, Epsilon: 0.100
Double Q-learning Episode 6000, Total Reward: -192, Epsilon: 0.100
Double Q-learning Episode 7000, Total Reward: -200, Epsilon: 0.100
Double Q-learning Episode 8000, Total Reward: -159, Epsilon: 0.100
Double Q-learning Episode 9000, Total Reward: -150, Epsilon: 0.100
Double Q-learning Episode 10000, Total Reward: -157, Epsilon: 0.100
Double Q-learning Training finished! Time elapsed: 29.05 seconds

Final Evaluation Total Reward (Double Q-learning): -161.0
```




```
Average reward in final evaluation: -161.0
```

DQN Implementation: Replay Buffer, Network Architecture, and Training Loop:

In [10]:
```
class ReplayBuffer:
    def __init__(self, capacity=100000):
        self.buffer = deque(maxlen=capacity)

    def push(self, transition):
        self.buffer.append(transition)

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*batch)
        return (
            torch.tensor(np.array(state), dtype=torch.float32),
            torch.tensor(action, dtype=torch.int64),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(np.array(next_state), dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32)
        )

    def __len__(self):
        return len(self.buffer)

class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

def train_dqn():
    env = SimpleMountainCarEnv()
    alpha = 0.001
    gamma = 0.99
    epsilon_min = 0.01
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```python
    epsilon_decay = 0.999
    batch_size = 64
    n_episodes = 1000
    max_steps = 200
    target_update_freq = 10

    input_dim = 2
    output_dim = 3

    policy_net = DQN(input_dim, output_dim)
    target_net = DQN(input_dim, output_dim)
    target_net.load_state_dict(policy_net.state_dict())
    target_net.eval()

    optimizer = optim.Adam(policy_net.parameters(), lr=alpha)
    criterion = nn.MSELoss()

    replay_buffer = ReplayBuffer()

    def select_action(state, epsilon):
        if random.random() < epsilon:
            return random.choice([0, 1, 2])
        else:
            state_t = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
            with torch.no_grad():
                q_values = policy_net(state_t)
            return q_values.argmax().item()

    episode_rewards = []
    start_time = time.time()

    for episode in range(n_episodes):
        state = env.reset()
        total_reward = 0
        done = False

        for step in range(max_steps):
            action = select_action(state, epsilon)
            next_state, reward, done = env.step(action)

            replay_buffer.push((state, action, reward, next_state, done))
            state = next_state
            total_reward += reward

            if len(replay_buffer) >= batch_size:
                states, actions, rewards, next_states, dones = replay_buffer.sample(batch_size)

                q_values = policy_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

                with torch.no_grad():
                    max_next_q_values = target_net(next_states).max(1)[0]
                    targets = rewards + gamma * max_next_q_values * (1 - dones)

                loss = criterion(q_values, targets)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

            if done:
                break

        episode_rewards.append(total_reward)
        epsilon = max(epsilon_min, epsilon * epsilon_decay)

        if (episode + 1) % target_update_freq == 0:
            target_net.load_state_dict(policy_net.state_dict())

        if (episode + 1) % 100 == 0:
            print(f"DQN Episode {episode + 1}, Total Reward: {total_reward}, Epsilon: {epsilon:.3f}")

    end_time = time.time()
    print(f"DQN Training finished! Time elapsed: {end_time - start_time:.2f} seconds")
    return policy_net, episode_rewards
```

DQN Evaluation: Visualizing Agent Performance:

In [11]:

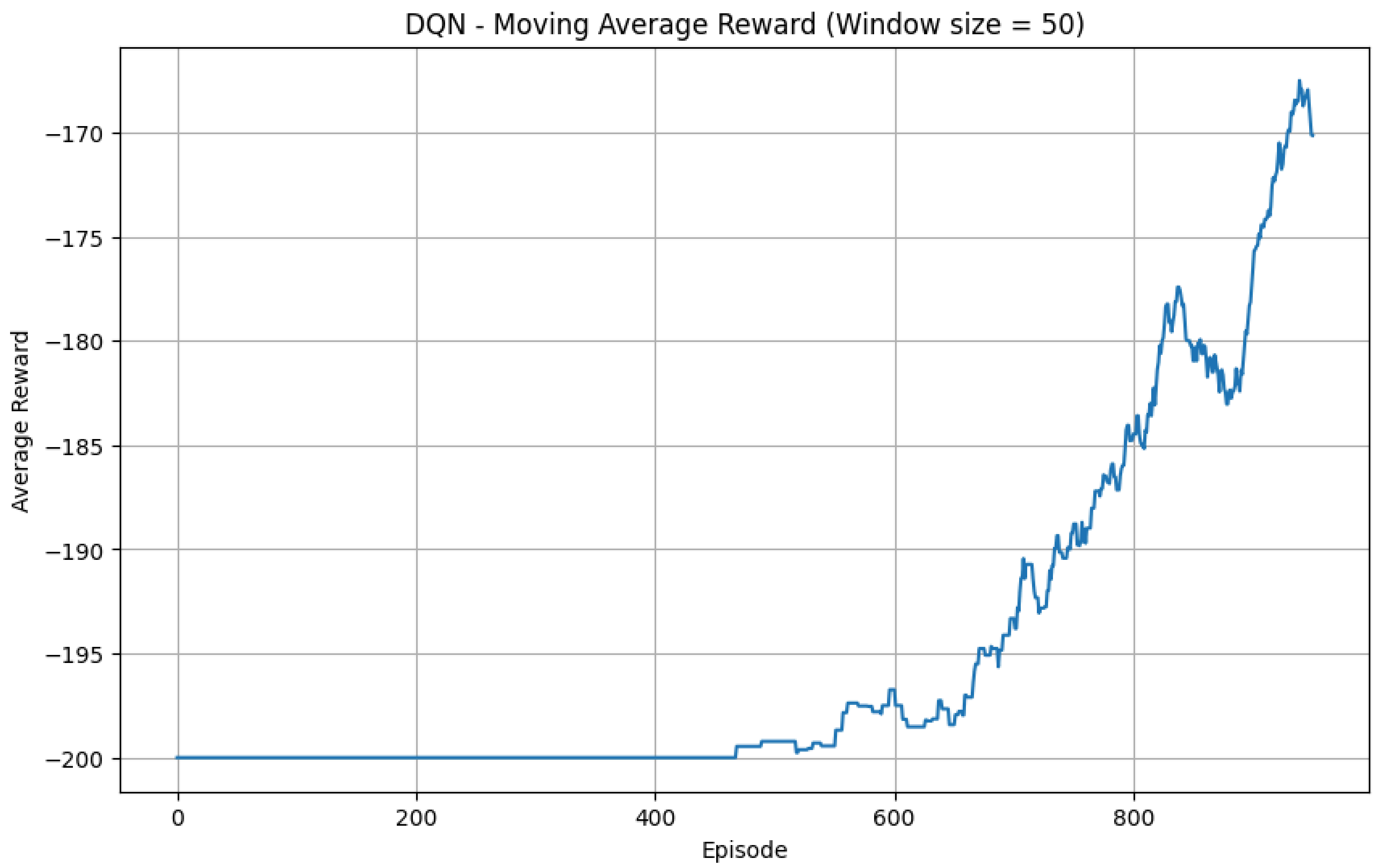
```python
def evaluate_dqn(policy_net):
    env_eval = gym.make("MountainCar-v0", render_mode="human")
    state_continuous, _ = env_eval.reset()
    total_reward = 0
    done = False

    while not done:
        state_t = torch.tensor(state_continuous, dtype=torch.float32).unsqueeze(0)
        with torch.no_grad():
            q_values = policy_net(state_t)
                                          tem()
```

```
        next_state, reward, terminated, truncated, _ = env_eval.step(action)
        done = terminated or truncated
        state_continuous = next_state
        total_reward += reward
        env_eval.render()
        time.sleep(0.01)

    print(f"\nFinal Evaluation Total Reward (DQN): {total_reward}")
    env_eval.close()
    return total_reward
```

DQN Training and Performance Visualization

In [12]:
```
if __name__ == "__main__":
    start_time = time.time()
    policy_net, dqn_rewards = train_dqn()
    time_dqn = time.time() - start_time
    window_size = 50
    moving_avg = np.convolve(dqn_rewards, np.ones(window_size) / window_size, mode='valid')
    plt.figure(figsize=(10, 6))
    plt.plot(moving_avg)
    plt.title("DQN - Moving Average Reward (Window size = 50)")
    plt.xlabel("Episode")
    plt.ylabel("Average Reward")
    plt.grid(True)
    plt.show()
    avg_reward = evaluate_dqn(policy_net)
    print(f"Average reward in final evaluation: {avg_reward}")
```

```
DQN Episode 100, Total Reward: -200, Epsilon: 0.905
DQN Episode 200, Total Reward: -200, Epsilon: 0.819
DQN Episode 300, Total Reward: -200, Epsilon: 0.741
DQN Episode 400, Total Reward: -200, Epsilon: 0.670
DQN Episode 500, Total Reward: -200, Epsilon: 0.606
DQN Episode 600, Total Reward: -200, Epsilon: 0.549
DQN Episode 700, Total Reward: -200, Epsilon: 0.496
DQN Episode 800, Total Reward: -178, Epsilon: 0.449
DQN Episode 900, Total Reward: -200, Epsilon: 0.406
DQN Episode 1000, Total Reward: -160, Epsilon: 0.368
DQN Training finished! Time elapsed: 216.68 seconds
```




```
Final Evaluation Total Reward (DQN): -156.0
Average reward in final evaluation: -156.0
```

General Evaluation Functions for RL Models:

In [13]:
```
def evaluate_model(model, model_type, n_runs=10):
    env = gym.make("MountainCar-v0")
    steps_list = []
    successes = 0

    for _ in range(n_runs):
        state = env.reset()
        state = state[0] if isinstance(state, tuple) else state
        done = False
        steps = 0
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
        while not done and steps < 200:
```

```
            if model_type == "Q-learning":
                state_disc = discretize_state(state)
                action = np.argmax(model[state_disc])
            elif model_type == "Double Q-learning":
                state_disc = discretize_state(state)
                action = np.argmax(model[0][state_disc] + model[1][state_disc])
            else:  # DQN
                state_t = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
                with torch.no_grad():
                    action = model(state_t).argmax().item()

            next_state, _, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            state = next_state
            steps += 1

        steps_list.append(steps)
        if steps < 200:
            successes += 1

    env.close()
    return np.mean(steps_list), successes/n_runs

def evaluate_steps_to_goal(model, model_type, n_episodes=100):
    env = gym.make("MountainCar-v0")
    steps_history = []

    for _ in range(n_episodes):
        state = env.reset()
        state = state[0] if isinstance(state, tuple) else state
        done = False
        steps = 0

        while not done and steps < 200:
            if model_type == "Q-learning":
                state_disc = discretize_state(state)
                action = np.argmax(model[state_disc])
            elif model_type == "Double Q-learning":
                state_disc = discretize_state(state)
                action = np.argmax(model[0][state_disc] + model[1][state_disc])
            else:  # DQN
                state_t = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
                with torch.no_grad():
                    action = model(state_t).argmax().item()

            next_state, _, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            state = next_state
            steps += 1

        steps_history.append(steps)

    env.close()
    return steps_history
```

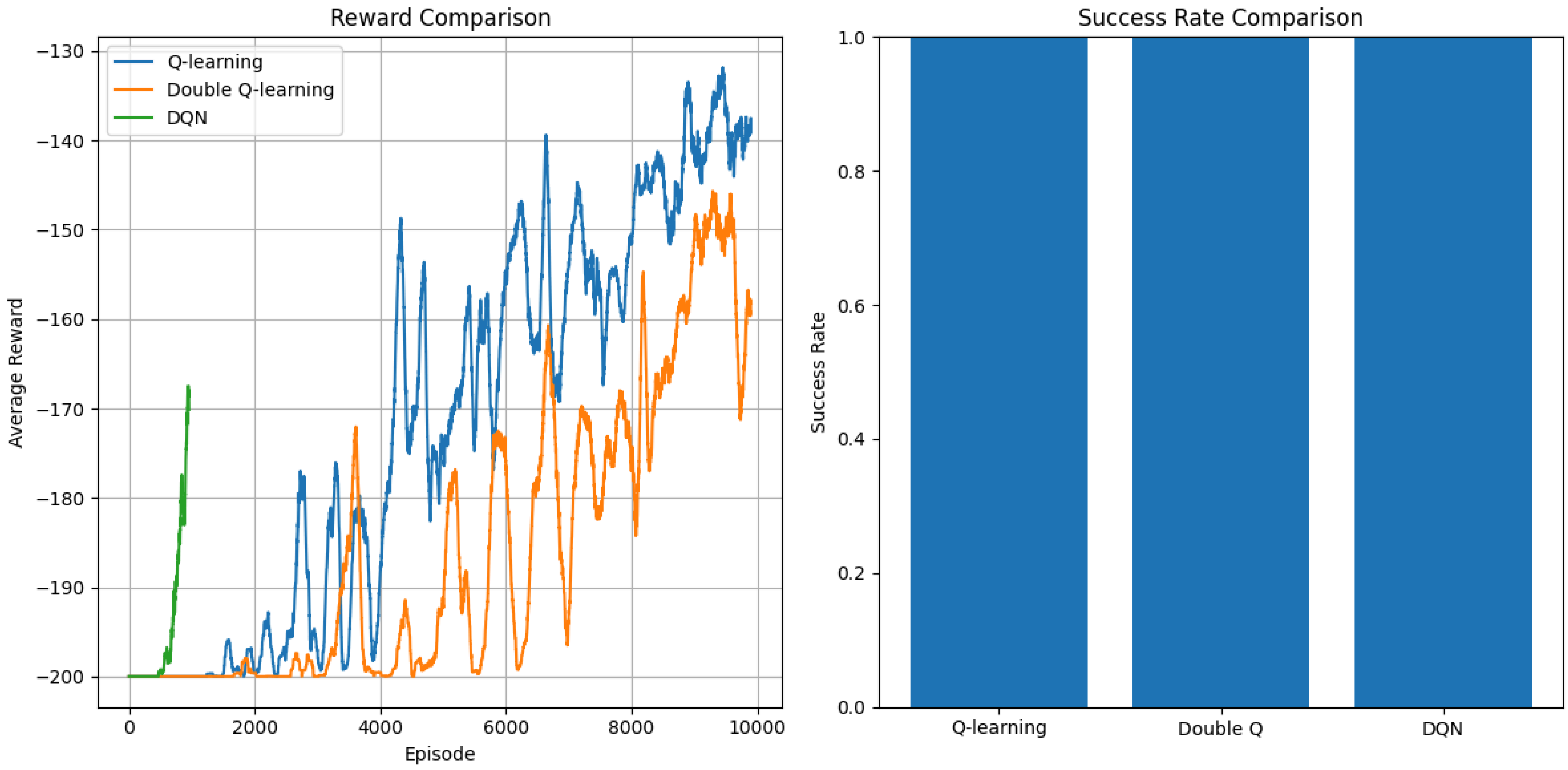
In this section, the program **trains all the algorithms, evaluates their performance on the MountainCar environment, and compares the results using tables and plots**. It also uses a **boxplot and cumulative success plot to show the distribution of steps to reach the goal and the success rate over episodes.

In [14]:
```
if __name__ == "__main__":
    # Evaluate models
    q_steps, q_success = evaluate_model(q_table, "Q-learning")
    dq_steps, dq_success = evaluate_model((q1, q2), "Double Q-learning")
    dqn_steps, dqn_success = evaluate_model(policy_net, "DQN")

    # Generate results table
    results = [
        ["Q-learning", q_steps, q_success, len(reward_history), time_q],
        ["Double Q-learning", dq_steps, dq_success, len(double_q_rewards), time_dq],
        ["DQN", dqn_steps, dqn_success, len(dqn_rewards), time_dqn]
    ]

    print("\nModel Comparison Table:")
    print("{:<20} {:<15} {:<15} {:<15} {:<15}".format(
        "Model", "Avg Steps", "Success Rate", "Episodes", "Train Time (s)"))
    for row in results:
        print("{:<20} {:<15.1f} {:<15.2f} {:<15} {:<15.1f}".format(*row))

    # Plot comparison charts
    plt.figure(figsize=(12, 6))

    # Reward plot
    plt.subplot(1, 2, 1)
    plt.plot(moving_average(reward_history, 100), label="Q-learning")
    plt.plot(moving_average(double_q_rewards, 100), label="Double Q-learning")
    plt.plot(moving_average(dqn_rewards, 50), label="DQN")
    plt.title("Reward Comparison")
    plt.ylabel("Average Reward")
```

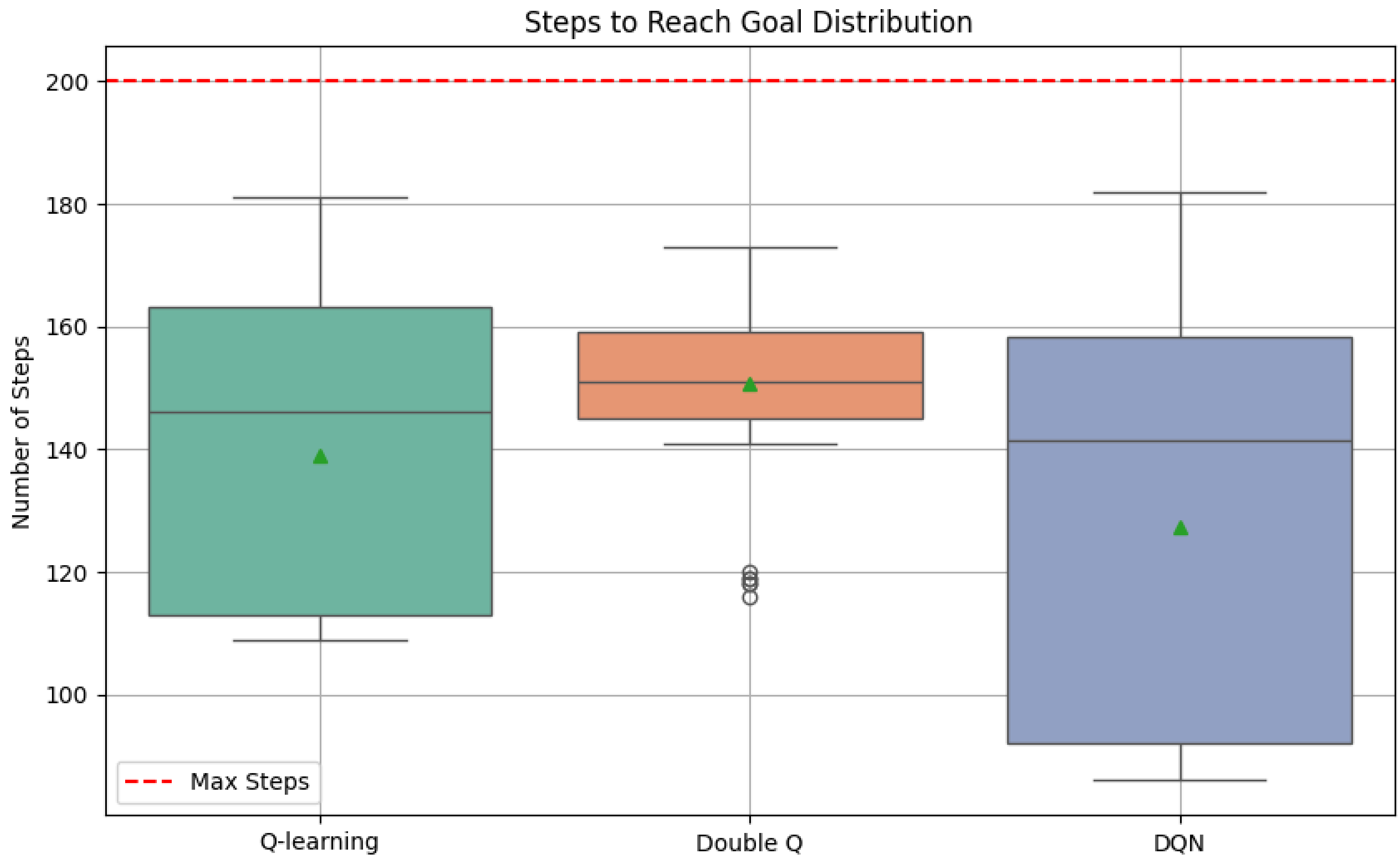
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

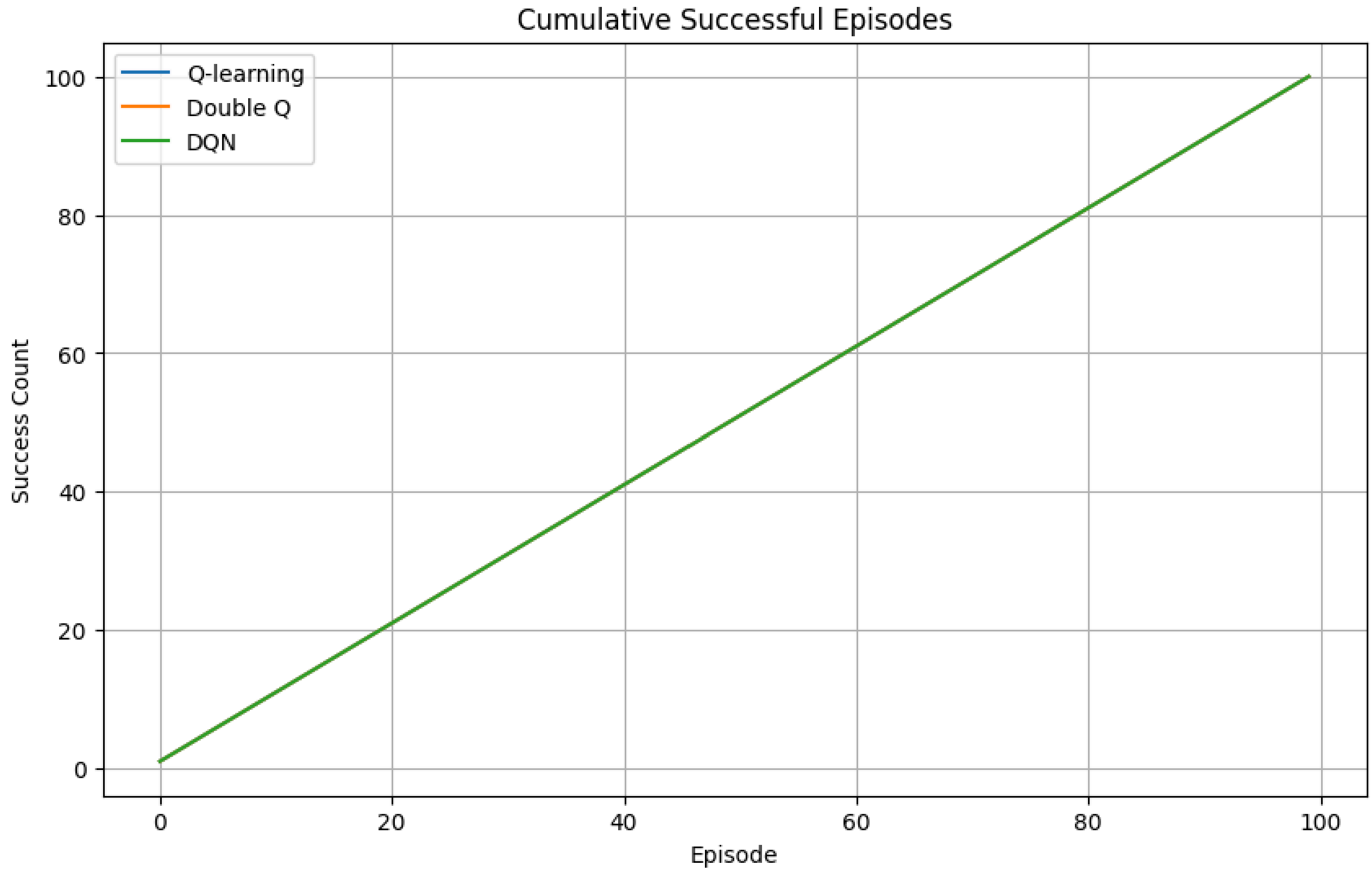
```python
    plt.legend()
    plt.grid()

    # Success rate bar chart
    plt.subplot(1, 2, 2)
    models = ["Q-learning", "Double Q", "DQN"]
    success_rates = [q_success, dq_success, dqn_success]
    plt.bar(models, success_rates)
    plt.title("Success Rate Comparison")
    plt.ylabel("Success Rate")
    plt.ylim(0, 1)

    plt.tight_layout()
    plt.show()

    # Time-to-goal evaluation
    q_steps_history = evaluate_steps_to_goal(q_table, "Q-learning")
    dq_steps_history = evaluate_steps_to_goal((q1, q2), "Double Q-learning")
    dqn_steps_history = evaluate_steps_to_goal(policy_net, "DQN")

    # Calculate statistics
    stats = {
        "Q-learning": {
            "mean": np.mean(q_steps_history),
            "std": np.std(q_steps_history),
            "success_rate": np.mean(np.array(q_steps_history) < 200)
        },
        "Double Q-learning": {
            "mean": np.mean(dq_steps_history),
            "std": np.std(dq_steps_history),
            "success_rate": np.mean(np.array(dq_steps_history) < 200)
        },
        "DQN": {
            "mean": np.mean(dqn_steps_history),
            "std": np.std(dqn_steps_history),
            "success_rate": np.mean(np.array(dqn_steps_history) < 200)
        }
    }

    # Print results
    print("\nTime-to-Goal Performance:")
    print("{:<20} {:<15} {:<15} {:<15}".format(
        "Model", "Mean Steps", "Std Dev", "Success Rate"))
    for model, data in stats.items():
        print("{:<20} {:<15.1f} {:<15.1f} {:<15.2f}".format(
            model, data["mean"], data["std"], data["success_rate"]))

    # Visualization 1: Boxplot comparison
    plt.figure(figsize=(10, 6))
    sns.boxplot(data=[q_steps_history, dq_steps_history, dqn_steps_history],
                palette="Set2",
                showmeans=True)
    plt.xticks([0, 1, 2], ["Q-learning", "Double Q", "DQN"])
    plt.title("Steps to Reach Goal Distribution")
    plt.ylabel("Number of Steps")
    plt.axhline(200, color='r', linestyle='--', label="Max Steps")
    plt.legend()
    plt.grid(True)

    # Visualization 2: Cumulative success
    plt.figure(figsize=(10, 6))
    for name, steps in zip(["Q-learning", "Double Q", "DQN"], [q_steps_history, dq_steps_history, dqn_steps_history]):
        success = np.cumsum(np.array(steps) < 200)
        plt.plot(success, label=name)
    plt.title("Cumulative Successful Episodes")
    plt.xlabel("Episode")
    plt.ylabel("Success Count")
    plt.legend()
    plt.grid(True)

    plt.show()
```

```
Model Comparison Table:
Model                Avg Steps       Success Rate    Episodes        Train Time (s)
Q-learning           144.4           1.00            10000           24.7
Double Q-learning    152.9           1.00            10000           29.0
DQN                  139.4           1.00            1000            217.7
```




```
Time-to-Goal Performance:
Model               Mean Steps      Std Dev         Success Rate
Q-learning          139.0           24.9            1.00
Double Q-learning   150.6           11.5            1.00
DQN                 127.4           33.3            1.00
```





DQN Training with Multiple Loss, Optimizer, and Reward Configurations

In [15]:
```
def train_dqn2(loss_fn, optimizer_class, reward_func, n_episodes=1000, max_steps=200):
    env = SimpleMountainCarEnv()
    alpha = 0.001
    gamma = 0.99
    epsilon = 1.0
    epsilon_min = 0.01
    epsilon_decay = 0.995
    batch_size = 128
    target_update_freq = 10

    input_dim = 2
    output_dim = 3

    policy_net = DQN(input_dim, output_dim)
    target_net = DQN(input_dim, output_dim)
    target_net.load_state_dict(policy_net.state_dict())
    target_net.eval()

    optimizer = optimizer_class(policy_net.parameters(), lr=alpha)
    criterion = loss_fn()

    replay_buffer = ReplayBuffer()
    episode_rewards = []

    def select_action(state, epsilon):
        if random.random() < epsilon:
            return random.choice([0, 1, 2])
        else:
            state_t = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
            with torch.no_grad():
                q_values = policy_net(state_t)
            return q_values.argmax().item()

    start_time = time.time()

    for episode in range(n_episodes):
        state = env.reset()
        total_reward = 0
        done = False

        for step in range(max_steps):
            action = select_action(state, epsilon)
            next_state, reward, done = env.step(action)

            reward = reward_func(state, action, next_state, reward)

            replay_buffer.push((state, action, reward, next_state, done))
            state = next_state
            total_reward += reward

            if len(replay_buffer.buffer) >= batch_size:
                ards_batch, next_states, dones = replay_buffer.sample(batch_size)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
                q_values = policy_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
                with torch.no_grad():
                    max_next_q_values = target_net(next_states).max(1)[0]
                    targets = rewards_batch + gamma * max_next_q_values * (1 - dones)
                loss = criterion(q_values, targets)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

            if done:
                break

        episode_rewards.append(total_reward)
        epsilon = max(epsilon_min, epsilon * epsilon_decay)

        if (episode + 1) % target_update_freq == 0:
            target_net.load_state_dict(policy_net.state_dict())

    end_time = time.time()
    training_time = end_time - start_time

    return policy_net, episode_rewards, training_time

def original_reward(state, action, next_state, reward):
    return reward

def modified_reward(state, action, next_state, reward):
    return reward + 0.1 * next_state[0]

experiments = [
    {"loss": nn.MSELoss, "optimizer": optim.Adam, "reward": original_reward, "name": "Exp1"},
    {"loss": nn.SmoothL1Loss, "optimizer": optim.Adam, "reward": original_reward, "name": "Exp2"},
    {"loss": nn.MSELoss, "optimizer": optim.RMSprop, "reward": original_reward, "name": "Exp3"},
    {"loss": nn.MSELoss, "optimizer": optim.Adam, "reward": modified_reward, "name": "Exp4"},
    {"loss": nn.SmoothL1Loss, "optimizer": optim.RMSprop, "reward": original_reward, "name": "Exp5"},
    {"loss": nn.SmoothL1Loss, "optimizer": optim.Adam, "reward": modified_reward, "name": "Exp6"},
]

results = []

for exp in experiments:
    print(f"Running {exp['name']}...")
    policy_net, rewards, training_time = train_dqn2(exp["loss"], exp["optimizer"], exp["reward"])
    avg_reward = np.mean(rewards[-50:])
    avg_steps = len(rewards) / sum(rewards) if sum(rewards)!=0 else 0
    results.append({
        "Experiment": exp["name"],
        "Loss": exp["loss"].__name__,
        "Optimizer": exp["optimizer"].__name__,
        "Reward Function": exp["reward"].__name__,
        "Average Reward": avg_reward,
        "Training Time (s)": training_time
    })

df = pd.DataFrame(results)
df
```

```
Running Exp1...
Running Exp2...
Running Exp3...
Running Exp4...
Running Exp5...
Running Exp6...
```

Out[15]:

	Experiment	Loss	Optimizer	Reward Function	Average Reward	Training Time (s)
0	Exp1	MSELoss	Adam	original_reward	-123.540000	229.105738
1	Exp2	SmoothL1Loss	Adam	original_reward	-155.940000	237.326781
2	Exp3	MSELoss	RMSprop	original_reward	-139.840000	221.760704
3	Exp4	MSELoss	Adam	modified_reward	-139.108143	241.678231
4	Exp5	SmoothL1Loss	RMSprop	original_reward	-200.000000	252.048421
5	Exp6	SmoothL1Loss	Adam	modified_reward	-160.946625	253.719864

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js