Value-Based Methods (Q-Learning)

Overview

This project implements **Q-Learning**, a **value-based reinforcement learning** algorithm that learns an optimal policy by estimating the value (Q-value) of state-action pairs. The objective is for the agent to learn to select the best actions to maximize cumulative rewards in the **CartPole-v1** environment.

Key Features

1. **Environment**:

   - The agent operates in OpenAI Gym's **CartPole-v1** environment, where it learns to balance a pole on a moving cart.

2. **Q-Table**:

   - The Q-table is initialized with zeros. It stores Q-values for all possible state-action pairs.
   - Each Q-value represents the expected future reward for taking a specific action in a specific state.

3. **Algorithm**:

   - The **Q-learning algorithm** is used to update the Q-values iteratively.
   - The **Bellman equation** is used to update the Q-values: [Q(s, a) = Q(s, a) + \alpha \times \left(\text{reward} + \gamma \times \max_a Q(s', a) - Q(s, a) \right)]
   - **Hyperparameters**:
     - **? (alpha)**: Learning rate.
     - **? (gamma)**: Discount factor for future rewards.
     - **? (epsilon)**: Exploration rate to balance exploration and exploitation.

4. **Exploration and Exploitation**:

   - The agent either **explores** by taking random actions with probability ? or **exploits** the learned Q-values to choose the best action.

How It Works

1. **Q-Table Initialization**:

   - The Q-table is initialized to zeros, with dimensions corresponding to the state space and action space.

2. **Q-Learning Process**:

   - In each episode:
     1. The agent begins from the initial state.
     2. The agent selects an action based on the epsilon-greedy policy (explore or exploit).
     3. The agent performs the action and receives a reward and the next state.
     4. The Q-value for the state-action pair is updated using the Bellman equation.

3. **Policy Evaluation**:

   - After training, the agent selects the best action for each state based on the learned Q-values.
   - The agent's performance is evaluated by running multiple test episodes and calculating the average reward.

Code Walkthrough

1. **Q-Table Initialization**:

```
 n_actions = env.action_space.n
n_states = env.observation_space.shape[0]
Q = np.zeros((n_states, n_actions))
```

2. **Q-Learning Algorithm**:

```
def q_learning(env, n_episodes=1000):
    for episode in range(n_episodes):
        state = env.reset()
        done = False
        while not done:
            if np.random.rand() < epsilon:
                action = env.action_space.sample()  # Explore
            else:
                action = np.argmax(Q[state])  # Exploit

            next_state, reward, done, _ = env.step(action)
            Q[state, action] += alpha * (reward + gamma * np.max(Q[next_state]) - Q
            state = next_state
```

3. **Evaluate Performance**:

```
 total_rewards = 0
for _ in range(10):
    state = env.reset()
    done = False
    while not done:
        action = np.argmax(Q[state])  # Exploit learned policy
        state, reward, done, _ = env.step(action)
        total_rewards += reward
print(f"Average Reward: {total_rewards / 10}")
```

Advantages

- **Model-Free**: No need to learn or simulate the environment dynamics.
- **Exploration and Exploitation**: The epsilon-greedy strategy ensures a balance between exploration and exploitation.
- **Simple and Effective**: Works well for small state-action spaces.

Future Work

- **State Space Discretization**: Modify the environment or use function approximation (e.g., deep Q-networks) for continuous state spaces.
- **Advanced Exploration Strategies**: Implement decay for the exploration rate (?) to reduce exploration over time.
- **Comparison**: Compare Q-learning with other RL algorithms like SARSA or Deep Q-Networks (DQN).

References

- Q-Learning Wikipedia
- OpenAI Gym Documentation