Multi-Agent Reinforcement Learning (MARL) Implementation

Overview

This project demonstrates **Multi-Agent Reinforcement Learning (MARL)** by simulating multiple agents interacting in an environment. Each agent independently learns its policy using **Q-Learning**, a model-free RL algorithm.

Features

- **Environment**: Multi-agent adaptation of the CartPole-v1 environment from OpenAI's Gym.
- **Agents**: Two independent agents learning simultaneously within the same environment.
- **Learning Algorithm**: Q-Learning with individual Q-tables for each agent.
- **Evaluation**: Trained agents are tested for performance over multiple episodes.

Setup

Required Libraries

- `numpy` : Numerical computations.
- `gym` : Reinforcement learning environments.

Installation

Install the necessary dependencies:

```
pip install numpy gym
```

How It Works

1. **Initialization**:
    - Each agent initializes its own **Q-table**, with dimensions `[states x actions]`.
    - Learning parameters:
        - **Alpha (?)**: Learning rate.
        - **Gamma (?)**: Discount factor.
        - **Epsilon (?)**: Exploration rate.
2. **Training**:
    - Each agent selects actions based on an **epsilon-greedy policy**:
        - **Exploration**: Selects a random action.
        - **Exploitation**: Selects the action with the highest Q-value.
    - After executing the action, the **Q-value** is updated using the Bellman equation: [Q(s,a) = Q(s,a) + \alpha \times \left[r + \gamma \times \max Q(s',a') - Q(s,a)\right]]
    - Training continues for a specified number of episodes.
3. **Testing**:

- The agents execute their learned policies to evaluate performance.
- Average rewards over test episodes are computed.

Code Walkthrough

1. **Environment Setup**:

```
env = gym.make('CartPole-v1')
n_agents = 2
```

2. **Q-Table Initialization**:

```
Q = [np.zeros((env.observation_space.shape[0], env.action_space.n)) for _ in range
```

3. **Epsilon-Greedy Policy**:

```
if np.random.rand() < epsilon:
    action = env.action_space.sample()  # Exploration
else:
    action = np.argmax(Q[i][state])  # Exploitation
```

4. **Q-Value Update**:

```
Q[i][state, action] += alpha * (reward + gamma * np.max(Q[i][next_state]) - Q[i][s
```

5. **Testing**:

```
for _ in range(10):
    actions = [np.argmax(Q[i][state]) for i in range(n_agents)]
    next_states, rewards, dones, _ = env.step(actions)
    total_rewards[i] += rewards[i]
```

Expected Outcomes

- **Policy Development**: Each agent learns a policy to maximize rewards independently.
- **Performance Evaluation**: Agents demonstrate improved performance through accumulated rewards during testing.

Key Advantages

- Enables multiple agents to learn and interact simultaneously.
- Independent learning ensures flexibility and scalability.
- Modular implementation for adapting to complex multi-agent environments.

Future Enhancements

- **Communication**: Enable agents to share information for cooperative strategies.
- **Centralized Training**: Combine Q-values into a shared table for coordinated learning.
- **Advanced Algorithms**: Implement deep reinforcement learning (e.g., MADDPG) for continuous action spaces.

References

- OpenAI Gym Documentation: https://gym.openai.com
- Reinforcement Learning with Python: Sutton & Barto