Multi-Agent Reinforcement Learning Simulation

This project showcases the intricate development of a Multi-Agent Reinforcement Learning (MARL) environment where agents engage with dynamic surroundings to achieve predefined objectives. The simulation incorporates the Proximal Policy Optimization (PPO) algorithm to train agents in navigating, learning from their environment, and adapting their behaviors while maintaining efficiency and scalability.

Table of Contents

1. Directory Structure

The project directory is organized for clarity and efficiency:

```
project-root/
├── data/              # Placeholder for input data
├── logs/              # TensorBoard logs and performance metrics
├── models/            # Saved trained models
├── notebooks/         # Jupyter notebooks for experiments
├── outputs/           # Simulation outputs (e.g., GIFs)
├── src/               # Source code for the project
│   ├── environment.py    # Custom environment definition
│   ├── formation.py      # Code for rendering and simulation
│   ├── trainer.py        # Training script using PPO
│   ├── shepely_p.py      # Placeholder for additional utilities
├── venv/              # Virtual environment for dependencies
├── dependencies.txt      # List of Python dependencies
├── README.md          # Project documentation
```

---

2. Project Dependencies

The project leverages state-of-the-art libraries and frameworks to enable functionality:

- **gymnasium**: A toolkit for constructing and interacting with reinforcement learning environments.
- **stable-baselines3**: An efficient library of RL algorithms, including PPO.
- **pettingzoo**: An interface specifically designed for Multi-Agent Reinforcement Learning (MARL).
- **supersuit**: Simplifies preprocessing environments for training.
- **shapely**: Offers tools for handling geometric objects and spatial reasoning.
- **pygame**: Facilitates simulation rendering and real-time visualization.
- **imageio**: Manages the generation and manipulation of simulation GIFs.

- **pillow**: A powerful library for image rendering and processing.

Dependencies are listed in `dependencies.txt` for streamlined installation, ensuring reproducibility across systems.

---

3. Source Files

a) environment.py

Purpose

Defines a custom multi-agent environment following the PettingZoo AEC (Agent-Environment Cycle) API. This ensures agents operate cohesively within a shared environment, adhering to reinforcement learning principles.

Key Components

Functions

1. `dist(p1, p2)`

    - **Purpose**: Calculates the Euclidean distance between two points, aiding agents in evaluating proximity.
    - **Arguments**:
        - `p1`: First point (tuple or NumPy array).
        - `p2`: Second point (tuple or NumPy array).
    - **Returns**: A float representing the distance between the points.

2. `draw(self)`

    - **Purpose**: Custom rendering logic for the environment, visualizing agents, landmarks, and polygons in the environment.
    - **Arguments**: None (operates on the environment's `self`).
    - **Returns**: None (draws directly to the screen).

Classes

1. `raw_env`

    - **Purpose**: Extends PettingZoo's `SimpleEnv`, defining the environment's core logic and interactions.
    - **Arguments**:

- `num_agents` (int): Number of agents in the environment.
- `max_cycles` (int): Maximum steps per episode.
- `continuous_actions` (bool): Whether actions are continuous.
- `render_mode` (str): Mode for rendering the environment (`rgb_array`, etc.).

- **Attributes**:
  - `metadata`: Metadata for the environment (e.g., name, render mode).

2. `Scenario`

- **Purpose**: Implements the logic for the environment's agents, landmarks, rewards, and observations.
- **Methods**:
  - `make_world(num_agents)`: Initializes agents and world properties.
  - `reset_world(world, np_random)`: Resets the environment for a new simulation.
  - `calc_closest_dist(agent, others)`: Computes the closest distance between an agent and others.
  - `point_in_shape(point)`: Checks if a point is inside the target shape.
  - `reward(agent, world)`: Calculates the reward for a given agent based on its position.
  - `observation(agent, world)`: Generates agent-specific observations to guide decision-making (relative positions).

---

b) formation.py

Purpose

Simulates agent interactions in a dynamic environment controlled by pre-trained PPO models. Captures frames from agent activities and compiles them into a GIF for visualization.

Key Components

Code Breakdown

1. **Environment Initialization**:

- `raw_env`: The custom environment is initialized with 10 agents and a maximum of 500 steps per episode.

2. **Loading Pre-Trained Models**:

- Models (`circle_model.zip`, `mountains_model.zip`) are loaded for controlling agents under different conditions (circle and mountain regions).

3. **Simulation Loop**:
    - Iterates over agents using the `agent_iter()` method of the environment.
    - Decides the action using the pre-trained models.
    - Captures frames of the simulation every 10 steps for visualization.
4. **Output**:
    - Saves the simulation as a GIF (`simulation.gif`) in the `outputs` directory.

Key Variables

- `frames`: List of simulation frames captured during the simulation.
- `custom_objects`: Defines overrides for model-specific attributes during loading.

Functions

- No standalone functions (loop logic directly implemented).

---

c) trainer.py

Purpose

Implements the PPO algorithm to train agents in the custom environment, enabling them to learn adaptive strategies for achieving objectives.

Key Components

Classes

1. `EarlyStoppingCallback`
    - **Purpose**: Stops training when the mean reward exceeds a specified threshold.
    - **Arguments**:
        - `reward_threshold` (float): Threshold for early stopping.
        - `verbose` (int): Logging verbosity level.
    - **Methods**:
        - `_on_step()`: Checks reward condition and stops training if the threshold is met.

Training Loop

1. **Environment Setup**:

- Environment (`env`) is initialized, converted to `ParallelEnv`, and wrapped with Supersuit for compatibility with PPO.

2. **Model Definition**:
   - PPO model with an MLP policy is initialized.
3. **Training**:
   - Iteratively trains the model for `50000` timesteps per iteration, saving the model at each step.
4. **Logging**:
   - Outputs metrics (e.g., rewards, losses) to TensorBoard, facilitating performance analysis.
5. **Output**:
   - Saves trained models in the `models` directory.

Functions

- **None** (logic is directly implemented in the script).

---

4. Simulation Environment Details

Agent Design and Functionality

- **Good Agents**:
  - Tasked with "arresting" bad agents by surrounding them collaboratively.
  - Exhibit adaptive movement patterns guided by PPO.
- **Bad Agents**:
  - Strategically avoid capture by employing evasive maneuvers.

Environment Features

- Agents navigate dynamic environments containing obstacles (e.g., mountains, polygons).
- Observations include spatial relationships between agents and obstacles.
- Rewards incentivize proximity to objectives and penalize unfavorable actions.

---

5. Training Process

PPO Implementation

- Leverages `stable-baselines3` for robust PPO implementation.
- Balances exploration (testing new strategies) and exploitation (refining successful strategies).

Logging and Monitoring

- Logs are stored in `logs/tensorboard` for analysis using TensorBoard.
- Tracks metrics such as:
    - Reward progression over episodes.
    - Policy loss and value function loss.

---

6. Simulation Visualization

GIF Generation

- Frames are captured at intervals, documenting agent movements and actions.
- Visual elements highlight:
    - Agent positions.
    - Dynamic interactions with obstacles.
    - Success in achieving objectives.

Agent Interaction

- Good agents (blue circles) pursue bad agents (red triangles).
- The generated GIF visualizes collaborative strategies and evasive maneuvers.

---

7. How to Run the Project

Setup

a) Create a virtual environment:

> python3 -m venv venv
> source venv/bin/activate
> pip install -r dependencies.txt

b) Run training:

> python3 src/trainer.py

c) Run simulation:

> python3 src/formation.py

---

8. Outputs and Interpretation

Logs:

- TensorBoard metrics are stored in `logs/tensorboard`.
- Start TensorBoard:

> tensorboard –logdir=logs/tensorboard

Models:

- Trained PPO models are saved in the `models` directory for reuse in simulations.

Simulation:

- The output GIF is saved in `outputs/simulation.gif`.
- Demonstrates:
    - Agent strategies and dynamics.
    - Visualizes success in achieving objectives.

---

9. Challenges and Solutions

Challenges

a) **Multi-Agent Coordination**:

- Designing reward functions that incentivize cooperation among good agents while discouraging bad agents from grouping.

b) **Complex Environments**:

- Balancing agent interaction with static obstacles like polygons and mountains.

c) **Efficient Logging**:

- Handling extensive data logs during training without impacting performance.

<u>Solutions</u>

- Reward functions were iteratively refined to align agent behaviors with objectives.
- Dynamic boundaries were introduced to prevent agents from being stuck outside frames.
- Logging mechanisms were optimized by integrating TensorBoard for structured monitoring.

---

10. Future Enhancements

- **Increased Agent Complexity**:
  - Introduce hierarchical agent roles (e.g., leaders and followers).
  - Enable agents to learn advanced cooperative behaviors.
- **Dynamic Environments**:
  - Add moving obstacles to simulate real-world complexity.
  - Introduce varying terrain types to test agent adaptability.
- **Alternative Algorithms**:
  - Experiment with algorithms like Deep Deterministic Policy Gradient (DDPG) or Actor-Critic methods.
- **Scalability**:
  - Scale simulations to accommodate hundreds of agents in diverse scenarios.

This documentation now offers an exhaustive overview of the project, elucidating every stage of implementation, analysis, and potential advancements.