Search and Rescue Multi-Agent Simulation Documentation

Contents

Chapter 1

Introduction

This document provides an in-depth overview of the **Search and Rescue Multi-Agent Simulation**, which utilizes the **PettingZoo** library for modeling a cooperative multi-agent environment. The simulation is designed to replicate dynamic rescue operations where rescuers navigate an environment containing obstacles, victims, and predefined safe zones to achieve specific objectives.

The following chapters describe the functionalities, components, structure, and limitations of the project.

Chapter 2

Functionalities

This project presents a comprehensive simulation of a **Search and Rescue** scenario, designed within a multi-agent environment using the **PettingZoo** library. The environment models a dynamic rescue operation where cooperative agents must navigate a space containing both obstacles and targets to achieve predefined goals. The simulation incorporates the following components:

2.1 Environment Setup

- **Configurable Parameters:** Users can define key elements of the environment, including:
  - Number of **rescuers** (agents responsible for guiding victims).
  - Number of **victims** (missing agents to be rescued).
  - Number of **trees** (obstacles obstructing navigation).
  - Number of **safe zones** (designated target areas for rescue).
- **Agent Interaction:** Rescuers interact with victims using a *push-based collision mechanism*. Upon collision, rescuers apply a controlled force that nudges victims toward their corresponding safe zones.
- **Safe Zone Placement:** Safe zones are fixed at the four corners of the environment. Each zone is assigned a specific **type** (e.g., A, B, C, D), and only victims of the matching type are considered successfully rescued when they reach the zone.

2.2 Rescuers (Agents)

Rescuers are intelligent agents that:

- Operate under motion constraints such as limited speed, acceleration, and maneuverability.
- Demonstrate reactive behavior by interacting dynamically with victims and avoiding obstacles.
- Use a collision mechanism to push victims while accounting for spatial constraints.

2.3 Victims (Missing Agents)

Victims represent passive agents scattered within the environment. They:

- Are assigned specific **types** (e.g., A, B, C, D) that correspond to safe zone types.
- Must be guided to their matching safe zones to be considered rescued.
- Exhibit limited or no independent movement, relying solely on rescuers for navigation.

2.4 Safe Zones

Safe zones serve as stationary rescue targets for victims:

- Safe zones are predefined at the four corners of the environment.
- Each safe zone is assigned a unique **type**, represented by attributes such as color (e.g., red, green, blue, and yellow).
- Successful rescues occur only when victims of matching types reach their corresponding safe zones.

2.5 Obstacles (Trees)

Trees are static obstacles that complicate agent navigation:

- Trees are randomly distributed within the environment and occupy fixed positions.
- Agents must navigate around these obstacles to avoid collisions, as collisions result in penalties.
- Obstacles add complexity by blocking direct paths between rescuers and victims or safe zones.

2.6 Collision Avoidance

Collision avoidance ensures smooth agent navigation:

- **Agent-Agent Avoidance:** Rescuers must actively avoid collisions with each other and victims. A distance-based mechanism evaluates relative agent positions to reduce unintended overlaps or blockages.
- **Obstacle Avoidance:** Agents are penalized for colliding with trees. Collisions are detected using:
  - Euclidean distance calculations between agent and obstacle centers.
  - Thresholds determined based on the sizes of agents and obstacles.

  Agents dynamically adjust their paths to avoid obstacles.

- **Line-of-Sight Blocking:** Trees may obstruct the line-of-sight between agents (e.g., rescuers and victims). When visibility is blocked, agents must navigate indirectly.
- **Reactive Behavior:** The environment promotes **indirect navigation** strategies, such as moving tangentially to obstacles, to simulate realistic planning behaviors.

2.7 Multi-Agent Interaction and Dynamics

- **Agent Collaboration:** Although agents operate independently, their shared objective of rescuing victims promotes emergent collaborative behaviors, such as efficient task distribution.
- **Motion Dynamics:** Agents operate under constraints such as speed limits and acceleration, simulating realistic movement. Rescuers apply controlled forces to guide victim agents.

2.8 Training and Evaluation

- **Training Framework:** The simulation uses Stable-Baselines3's **PPO (Proximal Policy Optimization)** algorithm for training.
- **RecurrentPPO Integration:** Partial support for **RecurrentPPO** allows for future memory-based enhancements.
- **Configurable Timesteps:** Users can adjust the training duration to control convergence efficiency.
- **Performance Logging:**
  - **TensorBoard:** Provides real-time performance visualization.
  - **CSV Logs:** Records key training metrics for offline analysis.
- **Evaluation:** Supports:
  - Manual model selection for evaluation.
  - Automatic loading of the most recent trained model.

  Evaluation computes average rewards and per-agent statistics.

2.9 Reward System

- **Positive Rewards:** Rescuers are rewarded for:
  - Moving victims closer to their corresponding safe zones.
  - Successfully guiding victims to their matching safe zones.
- **Shaping Rewards:** Incremental rewards are provided for reducing the distance between victims and safe zones.

- **Penalties:**
  - Collisions with obstacles.
  - Exceeding environment boundaries.
  - Victims remaining stationary.

2.10 Observation Space

Agents receive structured observations:

- Relative positions of landmarks (trees and safe zones).
- Positions and velocities of agents (rescuers and victims).
- Distances to victims, obstacles, and target zones.

Observations are partially obstructed by obstacles, introducing **partial observability** and requiring agents to make decisions under uncertainty.

Chapter 3

Structure of the Code

The project is divided into four key components, each responsible for a specific aspect of the simulation pipeline: environment definition, agent training, policy evaluation, and workflow orchestration. The files are modular and designed to work cohesively to implement the search-and-rescue simulation.

3.1 Custom Environment: `sar_env_updated.py`

The `sar_env_updated.py` file defines a multi-agent search-and-rescue environment using the PettingZoo Multi-Particle Environment (MPE) framework. It models agents (rescuers and victims), landmarks (trees and safe zones), and interactions to enable reinforcement learning experiments.

3.1.1 Environment Dynamics

The environment operates in a 2D space where each agent i has a position $\mathbf{x}_i$ and velocity $\mathbf{v}_i$:

$$\mathbf{x}_i = [x_i, y_i], \quad \mathbf{v}_i = [v_{x_i}, v_{y_i}]. \tag{3.1}$$

Positions are updated based on velocities over a time step Δt:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \Delta t \cdot \mathbf{v}_i. \tag{3.2}$$

The boundaries of the environment are defined as:

$$x_{\min} \leq x_i \leq x_{\max}, \quad y_{\min} \leq y_i \leq y_{\max}, \tag{3.3}$$

and agents are penalized if they move out of bounds.

Code Implementation:

```
def reset_world(self, world, np_random, reset_landmarks=True):
    for agent in world.agents:
        agent.state.p_pos = np_random.uniform(-0.5, +0.5, world.dim_p)
        agent.state.p_vel = np.zeros(world.dim_p)
    for landmark in world.landmarks:
        if reset_landmarks:
            landmark.state.p_pos = np_random.uniform(-0.5, +0.5, world.dim_p)
```

3.1.2 Collision Detection

Collisions occur when the distance between two entities (e.g., agents or landmarks) is less than the sum of their radii:

$$\|\mathbf{x}_1 - \mathbf{x}_2\| \leq r_1 + r_2, \tag{3.4}$$

where r_1 and r_2 are the radii of the entities.

Code Implementation:

```
def is_collision(self, agent1, agent2):
    delta_pos = agent1.state.p_pos - agent2.state.p_pos
    dist = np.sqrt(np.sum(np.square(delta_pos)))
    dist_min = agent1.size + agent2.size
    return dist < dist_min
```

3.1.3 Reward Mechanism

The reward mechanism governs the behavior of two types of agents in the environment:

1. **Rescuers (adversaries):** Agents rewarded for guiding victims toward matching safe zones while avoiding obstacles and penalties.
2. **Victims (non-adversaries):** Victims do not independently earn rewards, but their state impacts the rewards of rescuers.

The reward function for agent i is defined as:

$$R_i = \begin{cases} R_{\text{rescuer}} & \text{if agent } i \text{ is a rescuer,} \\ R_{\text{victim}} & \text{if agent } i \text{ is a victim.} \end{cases} \tag{3.5}$$

Rescuer Reward: `adversary_reward()`

Rescuers are rewarded for successfully guiding victims toward their corresponding safe zones and penalized for undesirable behaviors such as collisions and moving out of bounds.

The rescuer reward R_{rescuer} consists of three components:

$$R_{\text{rescuer}} = R_{\text{shaping}} + R_{\text{success}} - R_{\text{penalty}}. \tag{3.6}$$

Shaping Reward: Shaping rewards encourage rescuers to reduce the distance to victims and help victims approach safe zones:

$$R_{\text{shaping}} = -\beta \cdot d_{\text{rescuer-victim}} + \frac{1}{1 + d_{\text{victim-safe}}}. \tag{3.7}$$

Here:

- $d_{\text{rescuer-victim}}$ is the Euclidean distance between the rescuer and victim.
- $d_{\text{victim-safe}}$ is the Euclidean distance between the victim and the matching safe zone.
- $\beta > 0$ is a penalty weight.

Success Reward: A large reward is given for successfully guiding a victim to its corresponding safe zone:

$$R_{\text{success}} = \begin{cases} 100, & \text{if victim reaches its matching safe zone,} \\ 0, & \text{otherwise.} \end{cases} \tag{3.8}$$

Penalty Terms: Penalties are applied for collisions with obstacles and for moving out of bounds:

$$R_{\text{penalty}} = \gamma \cdot \mathbf{1}_{\text{collision}} + \eta \cdot \mathbf{1}_{\text{out-of-bounds}}, \tag{3.9}$$

where $\gamma > 0$ and $\eta > 0$ are penalty scalars.

Code Implementation:

```
def adversary_reward(self, agent, world):
    reward = 0
    shape = True
    victims = [v for v in self.good_agents(world) if not v.saved]  # Unsaved victims
    safezones = self.safezones(world)

    if shape:
        for victim in victims:
            # Penalize distance to victim
            distance_to_victim = np.linalg.norm(agent.state.p_pos - victim.state.p_po
            reward -= distance_to_victim * 0.1

            # Reward for reducing victim-safezone distance
            matching_safezone = next((sz for sz in safezones if sz.type == victim.typ
            if matching_safezone:
                distance_to_safezone = np.linalg.norm(victim.state.p_pos - matching_s
                reward += 1.0 / (1 + distance_to_safezone)

                # Large bonus for successful delegation
                if self.is_correctly_delegated(victim, matching_safezone):
                    reward += 100

    # Penalty for collisions
    for obstacle in world.landmarks:
        if obstacle.collide and self.is_collision(agent, obstacle):
            reward -= 10

    # Penalty for being out of bounds
    reward -= self.bound(agent.state.p_pos)

    return reward
```

Victim Reward: `agent_reward()`

Victims do not independently earn rewards but are penalized for being out of bounds, which indirectly impacts rescuer rewards.

The victim reward R_{victim} is:

$$R_{\text{victim}} = -\eta \cdot \mathbf{1}_{\text{out-of-bounds}}. \tag{3.10}$$

Code Implementation:

```
def agent_reward(self, agent, world):
    if agent.saved:
        return 0  # Saved victims earn no reward
    reward = 0
    # Penalty for being out of bounds
    reward -= self.bound(agent.state.p_pos)
    return reward
```

Combining Rewards

The overall reward mechanism for the environment can be summarized as:

$$R_{\text{total}} = \sum_{i \in \mathcal{A}} R_i, \tag{3.11}$$

where R_i is the reward for each agent, $\mathcal{A}$ is the set of all agents, and R_{total} is the cumulative reward for the environment.

3.1.4 Observation Space

The observation for agent i consists of its state and relative positions of nearby entities:

$$\mathbf{o}_i = \left[\mathbf{v}_i, \mathbf{x}_i, \mathbf{x}_{\text{rel agents}}, \mathbf{x}_{\text{rel landmarks}}\right], \tag{3.12}$$

where $\mathbf{x}_{\text{rel}}$ denotes relative positions:

$$\mathbf{x}_{\text{rel}} = \mathbf{x}_{\text{other}} - \mathbf{x}_i. \tag{3.13}$$

Code Implementation:

```
def observation(self, agent, world):
    entity_pos = [entity.state.p_pos - agent.state.p_pos for entity in world.landmark
    other_pos = [
        other.state.p_pos - agent.state.p_pos
        for other in world.agents if other is not agent
    ]
    return np.concatenate([agent.state.p_vel] + [agent.state.p_pos] + entity_pos + ot
```

3.2 Training Pipeline: `train_updated.py`

The `train_updated.py` file implements the training process for the search-and-rescue environment using the **Proximal Policy Optimization (PPO)** algorithm from Stable-Baselines3. The pipeline is optimized for efficient reinforcement learning by leveraging parallel simulations, observation preprocessing, performance logging, and model-saving mechanisms.

3.2.1 Reinforcement Learning Setup

The PPO algorithm trains agents (rescuers) to maximize the cumulative reward function R_{total}, as defined in the environment.

Objective Function: PPO optimizes the clipped surrogate objective:

$$L^{\text{PPO}}(\theta) = E_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right], \tag{3.14}$$

where:

- $r_t(\theta) = \frac{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}{\pi_{\theta_{\text{old}}}(\mathbf{a}_t|\mathbf{s}_t)}$ is the probability ratio between the current and old policies.
- $\hat{A}_t$ is the estimated advantage at time step t.
- ϵ is the clipping threshold (e.g., 0.2) to prevent excessively large updates.

Hyperparameter Configuration: Key hyperparameters are carefully tuned to optimize learning efficiency:

$$\text{Hyperparameters} = \{\alpha,\ \text{batch size},\ \text{n steps},\ \text{epochs}\}, \tag{3.15}$$

where:

- α is the learning rate.
- `batch size` controls the number of samples per gradient update.
- `n steps` determines how many environment steps are collected per update.
- `epochs` specifies how many times data is replayed.

Code Implementation:

```
from stable_baselines3 import PPO

model = PPO(
    "MlpPolicy",
    env,
    learning_rate=3e-4,
    n_steps=2048,
    batch_size=64,
    n_epochs=10,
    verbose=1
)
```

3.2.2 Observation Preprocessing

The observations from the environment are preprocessed to ensure a consistent input format for the neural network policy.

Padding Observations: Observations are padded using **SuperSuit**'s `pad_observations_v0` wrapper:

$$\mathbf{o}_{\text{padded}} = \text{pad}(\mathbf{o}_{\text{raw}}, d_{\max}), \tag{3.16}$$

where $\mathbf{o}_{\text{raw}}$ is the raw observation vector, and $d_{\max}$ is the maximum observation dimension.

Vectorized Environments: To enable efficient parallel simulations, the environment is vectorized using **SuperSuit**'s `concat_vec_envs_v1`:

$$\mathcal{E}_{\text{vec}} = [\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n], \tag{3.17}$$

where n is the number of parallel environments.

Code Implementation:

```
import supersuit as ss
env = ss.pad_observations_v0(env)
env = ss.concat_vec_envs_v1(env, 4, num_cpus=1)
```

3.2.3 Performance Logging

Training progress is logged to facilitate real-time monitoring and offline analysis.

TensorBoard Logging: Metrics such as rewards, losses, and learning rates are visualized in real time using TensorBoard:

$$\text{Logs} = \left\{R_{\text{total}}, L^{\text{PPO}}, \alpha, \ldots\right\}. \tag{3.18}$$

CSV Logs: Detailed performance metrics, including per-episode rewards and training loss, are saved in CSV format for post-processing and analysis.

Code Implementation:

```
from stable_baselines3.common.logger import configure

# TensorBoard and CSV Logger
new_logger = configure("logs/", ["stdout", "csv", "tensorboard"])
model.set_logger(new_logger)
```

3.2.4 Training Stagnation Detection

To ensure efficient training, a callback mechanism stops training if no performance improvement is observed over a predefined number of evaluation steps:

$$\text{Stop if: } E[R_{\text{eval}}] \leq \text{threshold}. \tag{3.19}$$

Code Implementation:

```
from stable_baselines3.common.callbacks import StopTrainingOnNoModelImprovement

callback = StopTrainingOnNoModelImprovement(
    max_no_improvement_evals=10,
    min_evals=5,
    verbose=1
)
model.learn(total_timesteps=1e6, callback=callback)
```

3.2.5 Model Saving

Trained models are saved with unique timestamps for easy identification:

$$\text{Model}_{\text{save}} = \text{PPO}_{\text{timestamp}}. \tag{3.20}$$

The saved model can be reused for evaluation, fine-tuning, or deployment:

$$\pi_{\theta_{\text{saved}}} = \text{load}(\text{PPO}_{\text{timestamp}}). \tag{3.21}$$

Code Implementation:

```
model.save(f"ppo_rescue_model_{int(time.time())}")
```

3.3 Evaluation Pipeline: `eval_updated.py`

The `eval_updated.py` script is responsible for evaluating trained policies in the search-and-rescue environment. It measures the performance of agents by running the environment over multiple episodes and computing cumulative rewards, success rates, and other relevant metrics.

3.3.1 Policy Loading

The evaluation pipeline supports manual and automated policy loading. A trained model $\pi_{\theta_{\text{trained}}}$ is loaded either by specifying its file path or automatically selecting the most recent model.

Manual Loading: The user provides the file path for the saved model:

$$\pi_{\theta_{\text{loaded}}} = \text{load}(\texttt{path\_to\_model}). \tag{3.22}$$

Automatic Loading: The script identifies the latest saved model using timestamps:

$$\pi_{\theta_{\text{loaded}}} = \arg\max_{\theta} \left(t_{\text{saved}}\right), \tag{3.23}$$

where t_{saved} denotes the timestamp of each saved model.

Code Implementation:

```
import os
from stable_baselines3 import PPO

model_path = "ppo_rescue_model_latest"
if not os.path.exists(model_path):
    model_path = sorted(os.listdir("models/"))[-1]  # Automatically load the latest
model = PPO.load(model_path)
```

3.3.2 Evaluation Process

The evaluation runs the environment for N_{episodes} episodes, where agents act based on the loaded policy $\pi_{\theta_{\text{loaded}}}$. The environment produces a trajectory:

$$\tau = \{(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1}) \mid t = 0, 1, \ldots, T\}, \tag{3.24}$$

where:

- $\mathbf{s}_t$ is the state at time step t.
- $\mathbf{a}_t$ is the action selected by the policy π_θ.
- r_t is the immediate reward.
- T is the total number of steps in an episode.

The cumulative reward for an episode is:

$$R_{\text{episode}} = \sum_{t=0}^{T} r_t. \tag{3.25}$$

Code Implementation:

```
def evaluate_policy(model, env, n_episodes=10):
    total_rewards = []
    for episode in range(n_episodes):
        obs = env.reset()
        episode_reward = 0
        done = False
        while not done:
            action, _ = model.predict(obs)
            obs, reward, done, _ = env.step(action)
            episode_reward += reward
        total_rewards.append(episode_reward)
    return total_rewards
```

3.3.3 Performance Metrics

Performance metrics are computed over multiple evaluation episodes to assess policy effectiveness.

Average Reward: The average cumulative reward across all episodes is:

$$\bar{R} = \frac{1}{N_{\text{episodes}}} \sum_{i=1}^{N_{\text{episodes}}} R_{\text{episode},i}. \tag{3.26}$$

Per-Agent Rewards: Individual agent rewards $R_{\text{agent},i}$ are computed for each rescuer and victim:

$$R_{\text{agent},i} = \sum_{t=0}^{T} r_{i,t}. \tag{3.27}$$

Success Rate: The success rate is defined as the proportion of victims successfully guided to their corresponding safe zones:

$$\text{Success Rate} = \frac{N_{\text{successful rescues}}}{N_{\text{victims}}}. \tag{3.28}$$

Code Implementation:

```
rewards = evaluate_policy(model, env, n_episodes=10)
avg_reward = sum(rewards) / len(rewards)
print(f"Average Reward: {avg_reward}")

# Display success rate
success_rate = num_successful_rescues / total_victims
print(f"Success Rate: {success_rate}")
```

3.3.4 Rendering the Environment

To visually verify agent behavior, the evaluation pipeline supports rendering the environment in real time. The rendering process provides visual feedback on agent trajectories, obstacle avoidance, and successful rescues.

Rendering Equation: The environment is rendered at each time step t:

$$\text{Render}(\mathbf{x}_t) \quad \forall t \in [0, T], \tag{3.29}$$

where $\mathbf{x}_t$ represents the positions of agents and landmarks.

Code Implementation:

```
env.render()  # Call render function to display real-time environment visuals
```

3.4 Workflow Orchestration: `main_updated.py`

The `main_updated.py` script serves as the central control hub for the project, orchestrating the **training** and **evaluation** workflows of the search-and-rescue environment. It allows users to configure the environment, initiate training, and evaluate the performance of trained policies.

3.4.1 Core Functionality

The script provides two primary execution modes:

1. **Training Mode:**
    - Calls the `train()` function to train agents using the PPO algorithm.
    - Initializes the environment and sets up the training pipeline, including logging mechanisms for monitoring progress.
    - Saves the trained models at regular intervals with unique timestamps for future evaluation or fine-tuning.
2. **Evaluation Mode:**
    - Calls the `eval()` function to evaluate the trained policy over multiple episodes.
    - Computes and displays performance metrics such as average rewards and per-agent performance.
    - Supports rendering the environment for real-time visualization of agent behavior.

3.4.2 Modular Design

The script is designed to be modular, enabling users to:

- Easily switch between training and evaluation modes using command-line arguments.
- Configure environment parameters (e.g., number of agents, obstacles, and safe zones) directly within the script.
- Extend the existing functionality for tasks such as fine-tuning, testing, or benchmarking policies.

3.4.3 Execution Flow

The execution flow is as follows:

- The user specifies the desired mode (`train` or `eval`) when running the script.
- The script initializes the environment and invokes the appropriate workflow based on the selected mode.
- Results, such as logs and trained models, are saved for analysis and future use.

3.4.4 Example Usage

The script can be executed as follows:

```
# Run in training mode
python main_updated.py --mode train

# Run in evaluation mode
python main_updated.py --mode eval
```