Multi-Agent Particle Environment Documentation

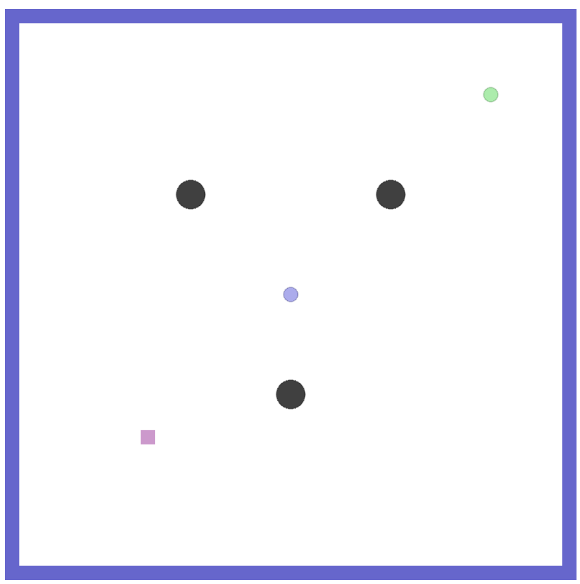
Gthelurd

November 15, 2024

1 Introduction

The original code for this project is sourced from the OpenAI GitHub repository: Multi-Agent Particle Environment. Before reading this document, please refer to the `README.md` file for a systematic introduction to the code files.

The environment used in this course is the `simple_tag.py` sub-environment, which has been modified to include boundary walls, three fixed obstacles, and one check-in points.



2 Environment Setup

2.1 Dependencies

The environment requires the following dependencies:

- Python 3.8+

- Pytorch
- Gym 0.10.5
- Numpy
- Scipy
- PIL (Pillow)

It is recommended to use Anaconda to create a virtual environment for configuration.

The framework can be either TensorFlow or PyTorch, depending on your needs. However, I used torch in the environment files.

It is recommended to use the Ubuntu system. The environment has not been tested on Windows, so it is not guaranteed to run successfully.

2.2 Installing Dependencies with Conda

Anaconda is highly recommended for installation.As I packed the environment files in the 'py38.yaml' file.

To install the dependencies specified in the 'py38.yaml' file into your Conda environment, follow these steps:

```
# Create a new Conda environment and activate it:
conda create --name myenv
conda activate myenv
# install the dependencies defined in py38.yaml:
conda env update --file py38.yaml
# Verify that all dependencies are correctly installed by
# listing the packages in the current environment:
conda list
```

3 Running the Code

After successful installation, the code should run without errors. The steps to run the code are as follows:

```
cd multiagent-envs-ML
python demo.py
```

You will see the identical environment windows, as there are two agents.The `demo.py` file can be considered the main function.

4 Environment Code Overview

4.1 Core Files

The most important environment files are `core.py`, `simple_tag.py`, and `environment.py`. The general calling relationship is: `core` is called by `simple_tag`, and `simple_tag` is called by `environment`.

- **core.py**: Declares various entities present in the environment, such as agents, borders, landmarks, and checks.
- **simple_tag.py**: Sets parameters for entities (initial positions, accelerations, etc.) and reward settings.
- **environment.py**: Provides the classic environment interface for reinforcement learning algorithms (step, reset, render, etc.). You can rewrite the `self._set_action` function in `environment.py` according to the given tasks.

4.2 Action Control

The control action (`action`) is a discrete value within the range $[0, 1]$ and is a 2x5-dimensional vector corresponding to two agents. Each agent is controlled by a 1x5-dimensional action vector, where the first dimension is not used, and the remaining four dimensions are used. So if you want

Action	**Description**
0	NOOP (No Operation)
1	UP (Move Up)
2	RIGHT (Move Right)
3	DOWN (Move Down)
4	LEFT (Move Left)

Table 1: Discrete Actions

to control the first agent to move up and the second agent to move right, you can set the action as $[1, 2]$.

4.3 Policy and Visualization

The `policy.py` file is used for keyboard control and is only for demonstration purposes (used in `interactive.py`).

The actual scale of the entire screen is $[-1, 1] \times [-1, 1]$, with the center of the screen as the origin. The screen width and height are 800x800 pixels.

4.4 Image Information Extraction

In the `/multiagent-envs-ML` folder, the `image.py` file provides a preliminary method to obtain image information. To extract target information from this, students need to implement their own methods.

Previous students have obtained image information in this way and used image processing (CenterNet algorithm) to obtain the positions of all agents, target points, and obstacles. This is the first task requirement from the instructor: to obtain the position information of each object through image processing.

Here is an example implementation of the `img_to_observation` function, which processes the image to extract the positions of agents, adversaries, check-in points, and obstacles:

```
import cv2
import numpy as np

def img_to_observation(image):
template_agent = cv2.imread('./images/agent.png')
template_adversary = cv2.imread('./images/adversary.png')
template_check = cv2.imread('./images/check.png')
template_obstacle = cv2.imread('./images/obstacle.png')
if template_agent is None or template_adversary is None or template_check is None or template_obstac
    raise ValueError("Failed to load one or more template images.")
resized_image = cv2.resize(image, (800, 800), 0, 0, cv2.INTER_MAX)
def find_object_positions(template):
    result = cv2.matchTemplate(resized_image, template, cv2.TM_CCOEFF_NORMED)
    threshold = 0.75
    loc = np.where(result >= threshold)
    positions = list(zip(*loc[::-1]))
    return positions
agent_positions = find_object_positions(template_agent)
adversary_positions = find_object_positions(template_adversary)
check_positions = find_object_positions(template_check)
obstacle_positions = find_object_positions(template_obstacle)
if agent_positions is None:
    agent_positions = np.array([400, 400])
if adversary_positions is None:
    adversary_positions = np.array([400, 400])
def calculate_mean_position(positions):
    if len(positions) == 0:
        return None
    positions = np.array(positions)
    mean_position = np.mean(positions, axis=0).astype(int)
    return mean_position
agent_pos = calculate_mean_position(agent_positions)
adversary_pos = calculate_mean_position(adversary_positions)
check_pos = calculate_mean_position(check_positions)
obstacle_pos = []
distance = []
for pos in obstacle_positions:
    obstacle_pos.append(pos)
    distance.append(np.linalg.norm(agent_pos - pos))
sorted_indexes = np.argsort(distance)
res = np.concatenate((
    check_pos - agent_pos,          # Relative position of the check point to the agent
    agent_pos,                      # Agent position
    adversary_pos - agent_pos,      # Relative position of the adversary to the agent
    obstacle_pos[sorted_indexes[0]] - agent_pos,
    # Relative position of the first obstacle to the agent
```

```
    obstacle_pos[sorted_indexes[1]] - agent_pos,
    # Relative position of the second obstacle to the agent
    obstacle_pos[sorted_indexes[2]] - agent_pos
    # Relative position of the third obstacle to the agent
)) / 256
return res
```

This function uses template matching to locate the positions of different objects in the environment and calculates their relative positions to the agent.

4.5 Direct Information Retrieval

There is code in `simple_tag.py` that directly retrieves position information:

```
def observed(self, agent, world):
    # get positions of all entities in this agent's reference frame
    entity_pos = []
    for entity in world.landmarks:
        if not entity.boundary:
            entity_pos.append(entity.state.p_pos - agent.state.p_pos)
    # communication of all other agents
    comm = []
    other_pos = []
    other_vel = []
    check_pos = []
    check_pos.append(agent.state.p_pos - world.check[0].state.p_pos)
    for other in world.agents:
        if other is agent:
            continue
        comm.append(other.state.c)
        other_pos.append(other.state.p_pos - agent.state.p_pos)
        other_vel.append(other.state.p_vel)
    dists = np.sqrt(np.sum(np.square(agent.state.p_pos - other_pos)))
    return np.concatenate([agent.state.p_pos] + other_pos + check_pos + entity_pos + [agent.state.p_
```

However, this method does not meet the instructor's requirements, as the instructor wants students to use image processing to obtain state information.

5 Innovative Parts

5.1 Advanced Navigation and Decision-Making

The environment includes advanced navigation and decision-making algorithms that enhance the agents' ability to interact with the environment effectively. These algorithms include functions to find the nearest adversary, nearest obstacle, and nearest check-in point, as well as a function to calculate the escape direction based on these factors.

5.1.1 Nearest Adversary Detection

The `get_nearest_adv` function identifies the nearest adversary to a given agent by calculating the Euclidean distance between the agent and all adversaries in the environment.

```
def get_nearest_adv(self, agent, world):
    nearest_adv = None
    min_dist = 100000
    for a in agent:
        if not a.adverary:
            good_agent = a
    for a in agent:
        if a.adverary:
            dist = np.linalg.norm(good_agent.state.p_pos - a.state.p_pos)
            if dist < min_dist:
                min_dist = dist
                nearest_adv = a
    return nearest_adv
```

5.1.2 Nearest Obstacle Detection

The `get_nearest_obstacle` function identifies the nearest obstacle to a given agent by calculating the Euclidean distance between the agent and all obstacles in the environment.

```
def get_nearest_obstacle(self, agent, world):
    nearest_obstacle = None
    min_dist = 100000
    for i, landmark in enumerate(world.landmarks):
        dist = np.linalg.norm(agent.state.p_pos - landmark.state.p_pos)
        if dist < min_dist:
            min_dist = dist
            nearest_obstacle = landmark
    return nearest_obstacle
```

5.1.3 Nearest Check-in Point Detection

The `get_check_point` function identifies the nearest check-in point to a given agent by calculating the Euclidean distance between the agent and all check-in points in the environment.

```
def get_check_point(self, agent, world):
    for check in self.checkpoints(world):
        dist = np.linalg.norm(agent.state.p_pos - check.state.p_pos)
        if dist < min_dist:
            min_dist = dist
            nearest_check = check
    return nearest_check
```

5.1.4 Escape Direction Calculation

The `calculate_escape_direction` function calculates the optimal escape direction for an agent based on the positions of the nearest adversary, nearest obstacle, and nearest check-in point. The function also introduces a threshold for decision-making and adds random perturbations to the direction to enhance unpredictability.

```
def calculate_escape_direction(self, agent, world):
    # Get the nearest adversary
    nearest_adv = world.agents[0]
    escape_direction = agent.state.p_pos - nearest_adv.state.p_pos
    distance_to_adv = np.linalg.norm(escape_direction)
    escape_direction /= distance_to_adv

    # Predict the adversary's next position
    adv_velocity = nearest_adv.state.p_vel
    adv_direction = adv_velocity / np.linalg.norm(adv_velocity)

    # Get the goal point
    goal = world.check[0]
    goal_direction = goal.state.p_pos - agent.state.p_pos
    goal_direction /= np.linalg.norm(goal_direction)

    # Get the nearest obstacle
    nearest_obs = self.get_nearest_obstacle(agent, world)
    obs_direction = agent.state.p_pos - nearest_obs.state.p_pos
    obs_distance = np.linalg.norm(agent.state.p_pos - nearest_obs.state.p_pos)
    obs_direction /= obs_distance

    # Set threshold
    threshold = 0.28

    # Calculate the final direction
    if distance_to_adv < threshold:
        # Move away from the adversary while moving towards the goal
        escape_weight = 0.9 - 0.8 * (distance_to_adv / threshold)
        goal_weight = 1.0 - escape_weight
        final_direction = escape_weight * escape_direction + goal_weight * goal_direction
    else:
        final_direction = 0.8 * goal_direction

    # Add avoidance strategy for obstacles
    if obs_distance < 0.16:  # If very close to an obstacle, increase avoidance weight
        final_direction = +0.8 * obs_direction - 0.2 * goal_direction

    # Introduce random perturbation
    if distance_to_adv > threshold and distance_to_adv < threshold + 0.2:  # Introduce random pertur
```

```
        perturbation = np.random.normal(0, 0.5, 2)
        final_direction += perturbation
    final_direction /= np.linalg.norm(final_direction)

    return final_direction
```