Indra RL Lab Documentation

Release 0.0.0

Indra RL Lab Team

2024-10-03

GETTING STARTED

CHAPTER

ONE

PREREQUISITES

Table 1: Software Requirements

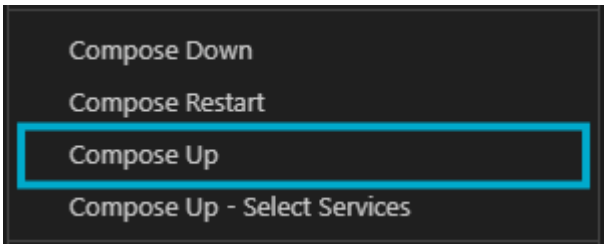
Software	Download Link	Tested Version
Docker Desktop	Docker Desktop	4.33.1
Visual Studio Code	VS Code	•
Windows X Server (for Docker GUI visualization)	VCXsrv	64.1.20.14.0

CHAPTER

TWO

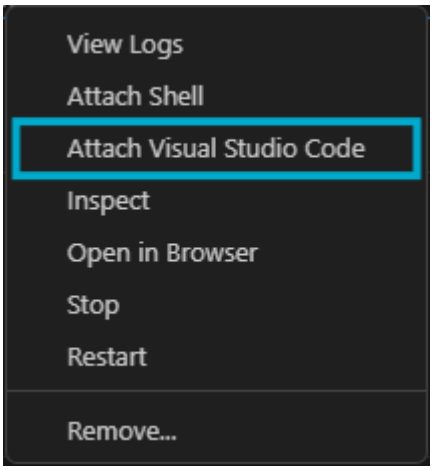
INSTALLATION

1. Clone the repository.
2. Install the Docker and Dev Containers extensions.
3. Navigate to the `docker-compose.yml` file and right-click 'Compose Up' to start the container (Ensure Docker Desktop is running).



If you don't have a dedicated Nvidia graphics card, please use the *docker-compose-no-gpu.yml* file.

1. Attach a Visual Studio Code to the running container by right-clicking on the running container in the Docker extension tab, and selecting 'Attach Visual Studio Code'.



2. Once attached to the running container, open a new terminal in the directory `/home` and build the ROS workspace by running:

```
bash build.bash
```

Expected output:

```
Building ROS project...
Command succeeded: bash -c 'cd /home/ros && source /opt/ros/humble/setup.bash && colcon␣
↪build --packages-select interfaces_pkg'
Starting >>> interfaces_pkg
Finished <<< interfaces_pkg [10.9s]

Summary: 1 package finished [11.4s]

Command succeeded: bash -c 'cd /home/ros && source /opt/ros/humble/setup.bash && colcon␣
↪build --symlink

Starting >>> examples_pkg
Starting >>> playground_pkg
Starting >>> ros_tcp_endpoint
Finished <<< examples_pkg [3.34s]
Finished <<< playground_pkg [3.50s]
Finished <<< ros_tcp_endpoint [3.66s]

Summary: 3 packages finished [4.09s]

ROS project built successfully!
```

2.1 Repository Structure

The only files you should be modifying here are the *config files* and the *custom models* to add your own torch architectures.

```
├── configs/
│   └── algorithm_config.yaml
├── experiments/
│   └── [Environment ID]/
│       └── [Algorithm]/
│           └── [Experiment Name]/
├── rl_pipeline/
│   ├── models/
│   │   ├── [Custom Blocks]
│   │   └── [Custom Feature Extractors]
│   ├── run/
│   │   └── rl_trainer.py
│   └── utils/
│       └── [Utility Scripts]
├── play.py
└── train.py
```

Note

- **configs/**: Contains YAML configuration files for experiments. Here everything related to the trainig is set up, you'll find specific example configurations for each algorithm including all the hyperparameters.
- **experiments/**: Stores experiment data and results locally.

- **rl_pipeline/:**
  - **models/**: Directory where one can add custom model architectures and blocks.
  - **run/:**
    - `rl_trainer.py`: The main trainer class.
    - `train.py`: Script to initiate training.
    - `play.py`: Script to run a trained agent.
  - **utils/**: Utility scripts for environment setup and other functionalities.

CHAPTER

THREE

DEPLOYMENT

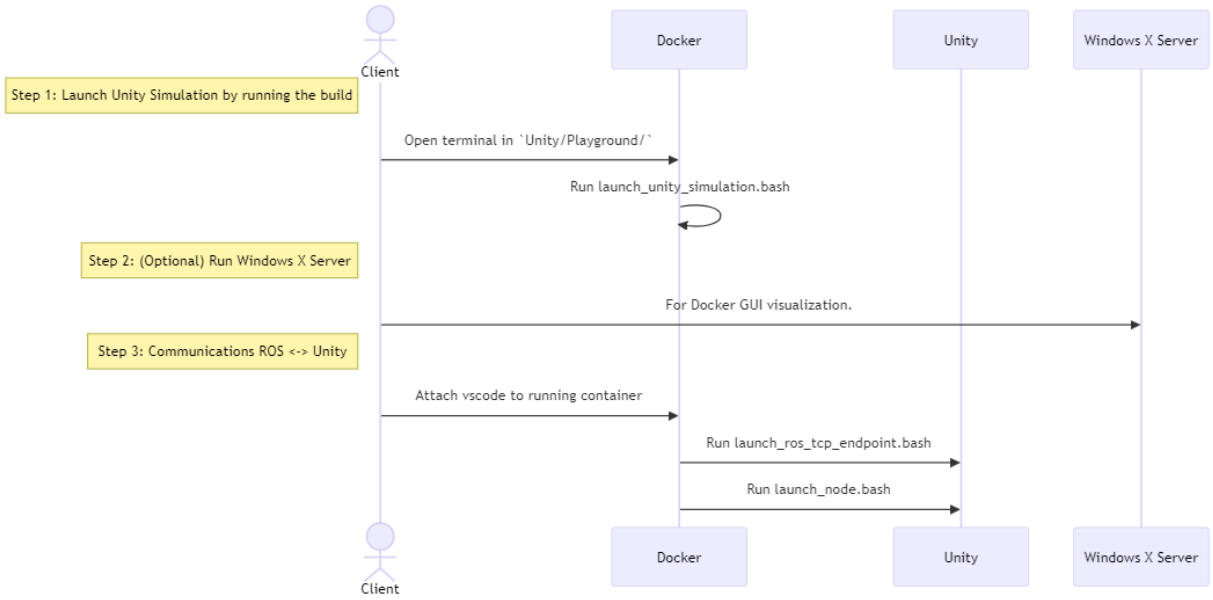
1. Launch the Unity simulation. This can be done in the Unity Editor (for development) or by running the build (for deployment).
   - Running the scene from the Unity Editor
     - Open the Unity Project in Playground
     - In the Unity Editor, open and play each of the User Cases, for example `Use Case 1` scene in Unity Scene
   - Running the build
     - Open a terminal and run the file Launch Unity Simulation

     ```
     launch_unity_simulation.bat
     ```
2. (Optional) Run Windows X Server for Docker GUI visualization.
3. In Visual Studio Code attached to the running container, open two new terminals and run the following commands in each one of them:

   ```
   bash launch_ros_tcp_endpoint.bash
   ```

   ```
   bash launch_node.bash
   ```

The `launch_node.bash` file will launch the package and node specified in the configuration, executing the training logic.


Client
Docker
Unity
Windows X Server
Step 1: Launch Unity Simulation by running the build
Open terminal in `Unity/Playground/`
Run launch_unity_simulation.bash
Step 2: (Optional) Run Windows X Server
For Docker GUI visualization.
Step 3: Communications ROS <-> Unity
Attach vscode to running container
Run launch_ros_tcp_endpoint.bash
Run launch_node.bash
Client
Docker
Unity
Windows X Server

CHAPTER

FOUR

TRAINING

To set up an experiment, modify one of the YAML configuration files in the `configs/` directory or create a new one. You can adjust hyperparameters, architecture, and specify a path to a trained model for preloading.

(MISSING) WHERE PATH TO PRETRAINED MODEL??? (MISSING) WHY NEED TO NOT SAY ATARI_GAME?? (MISSING) WHY ONLY RUN 1 EPISODE?

Example configuration file: `configs/base_ppo_config.yaml`

```yaml
experiment:
  name: 'BipedalWalker-Benchmark'

environment:
  id: 'BipedalWalker-v3'
  env_config: 'None'
  render_mode: 'rgb_array'
  monitor: true
  video_wrapper: true
  video_trigger: 5000
  video_length: 500

training:
  algorithm: 'ppo'
  pretrained_model: 'None'
  use_wandb: true
  algorithm_parameters:
    policy: 'MlpPolicy'
    learning_rate: 0.0003
    n_steps: 8192
    batch_size: 512
    n_epochs: 5
    gamma: 0.99
    gae_lambda: 0.95
    clip_range: 0.2
    ent_coef: 0.0
    vf_coef: 0.5
    max_grad_norm: 0.5

  architecture:
    net_arch: {'pi': [128, 128], 'vf': [128, 128]}
    features_extractor_class: 'ResnetMLP'
    features_extractor_kwargs:
```

(continues on next page)

(continued from previous page)

```
      features_dim: 128
    activation_fn: 'ReLU'
    share_features_extractor: false

  training:
    eval:
      seed: 5
      num_episodes: 5
      num_evals: 15
    total_timesteps: 500000
    device: 'cuda'
    log_points: 10
    verbose: 2

evaluation:
  num_episodes: 1

play:
  experiment: 'experiment_date'
  pretrained_model: 'model'
```

4.1 Adding Custom Models

To use a custom architecture, add your PyTorch model class to the `rl_pipeline/models/feature_extractors` directory. Your model should inherit from the base extractor class provided in `rl_pipeline/models/feature_extractors/base_extractor`.

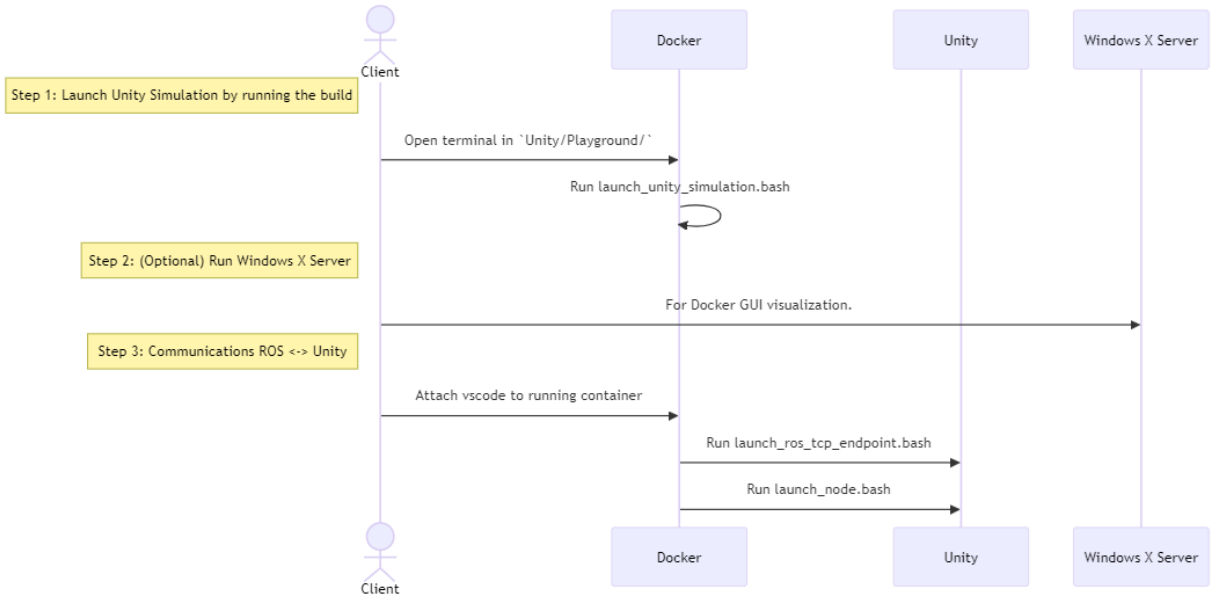
```
# rl_pipeline/models/custom_cnn.py
import torch
from rl_pipeline.models.feature_extractors.base_extractor import BaseFeaturesExtractor

class CustomCNN(BaseFeaturesExtractor):
    def __init__(self, observation_space, features_dim=256):
        super(CustomCNN, self).__init__(observation_space, features_dim)
        # Define your custom layers here

    def forward(self, observations):
        # Implement forward pass
        return features
```

Also, in `rl_pipeline/configs` directory the configuration should be updated to say that we use our custom model.

```
architecture:
  features_extractor_class: 'CustomCNN'
  features_extractor_kwargs:
    features_dim: 256
```

4.2 Running Training

To run an experiment, follow the steps mentioned in the main readme:



1. Define the simulation parameters in the `config.yml` file:

   (MISSING) NOT IN CURRENT config.yml:

```yaml
ros:
  package_name: "examples_pkg"
  node_name: "train"
```

   When running the environment as a Unity standalone build, other parameters such as the number of parallel environments, the time scale, pause, and headless mode flags can be modified:

```yaml
n_environments: 1

ros:
  package_name: "examples_pkg"
  node_name: "train"

unity:
  build_path: "build/Playground.exe"
  headless_mode: false
  pause: false
  sample_time: 0.0
  time_scale: 1.0
```

2. Launch the Unity simulation:

```
launch_unity_simulation.bat
```

3. In Visual Studio Code attached to the running container, open two new terminals and run the following commands:

```
bash launch_ros_tcp_endpoint.bash
bash launch_node.bash
```

The `launch_node.bash` file launches the package and node specified in the configuration.

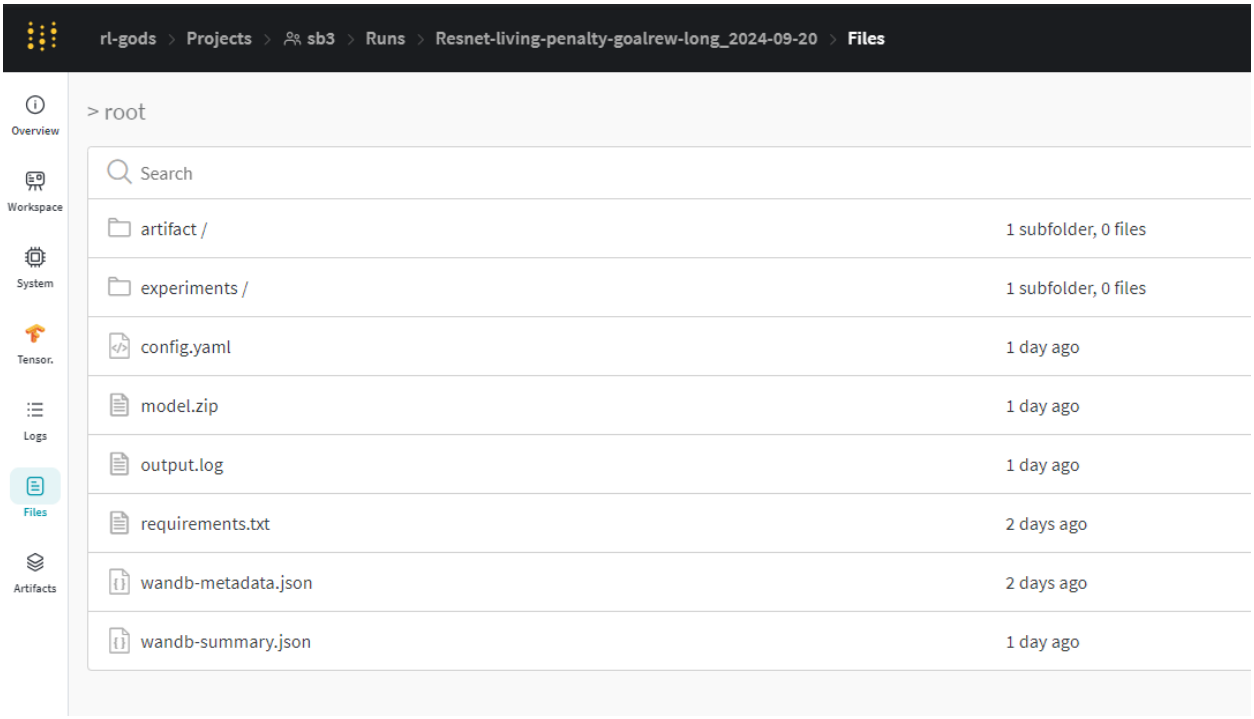
4.3 Loading Models

(MISSING) EXAMPLE:

In the `config` file, fill the `play` section with the name of the experiment and the model. Ensure the model and experiment folder follow the specified structure.

You can also download models from the Weights & Biases page under the `/files` section.

(MISSING) EXAMPLE:



4.3.1 Other Environments

You can also run experiments on other gymnasium environments by running the `train_example.py` script with your configuration:

(MISSING) WHERE SCRIPT???:

```
python train_example.py --config configs/base_ppo_config.yaml
```

CHAPTER

FIVE

TESTING

CHAPTER

SIX

WEIGHTS & BIASES LOGGING

The framework integrates with Weights & Biases for experiment tracking.

6.1 W&B logging

1. Set `use_wandb: true` in the `training` section of your configuration file.

```yaml
training:
  algorithm: 'ppo'
  pretrained_model: 'None'
  use_wandb: true
```

2. Run `wandb login` and enter your API key.

6.2 Access W&B Logs

Visit wandb.ai to see the runs under the project named `sb3` (or adjust the project name in `RLTrainer`).

```python
if self._config['use_wandb']:
        self._wandb_run = wandb.init(
            project="sb3",
            name=exp_name,
            group=wandb_group,
            config=config,
            sync_tensorboard=True,
            monitor_gym=True,
            save_code=False,
        )
```

6.3 Local Logging

All experiment data is saved locally in the `experiments/` directory:

(MISSING) WHERE AND WY??:

```
experiments/
└── [Environment ID]/
    └── [Algorithm]/
        └── [Experiment Name]/
            ├── videos/
            └── model.zip
```

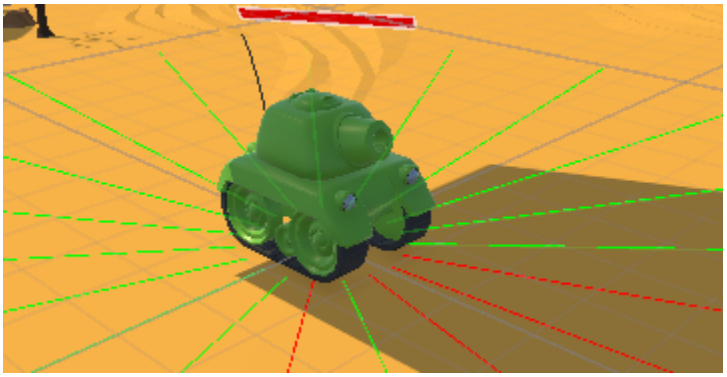
CHAPTER

SEVEN

USE CASE 1

7.1 Basic Navigation Example Environment

Use Case 1 integrates with the Gym library to create a basic simulation environment for tank navigation. To get observations from the environment, it uses a Lidar sensor whose rays can be seen in the following image:



Note

The rays in red appear when we are touching an object, which happends when an object is at a distance equal or smaller than 10 m to the tank, otherwise they are green.

7.2 Objectives

- Provide a fundamental environment for tank navigation using ROS and Gym.
- Enable integration with reinforcement learning algorithms by specifying observation, reward, and state management methods.
- Implement a training script that uses the Stable Baselines 3 library to train agents in the environment. It uses configuration files to try different algorithms, architectures, and hyperparameters.
- Develop a test script to validate the environment's functionality and behavior.

7.3 UC1Environment Class

The class *UC1Environment* is defined, that inherits from *EnvironmentNode* and configures a Gym environment with specific parameters defining the observation, action spaces,and the reward range. These configurations are crucial for defining the interaction between the agent and the environment, ensuring that both the agent's actions and the feedback it receives are appropriately scaled and represented.

7.3.1 Initialization

__init__(self, environment_id: int): This constructor sets up the environment by initializing ROS parameters and Gym environment parameters.

- **ROS Initialization**: Calls the parent class constructor (*EnvironmentNode.__init__*) to configure the ROS environment with specific message types and identifiers.

```
EnvironmentNode.__init__(
    self,
    environment_name="uc1_environment",
    environment_id=environment_id,
    step_service_msg_type=UC1EnvironmentStep,
    reset_service_msg_type=UC1EnvironmentReset,
)
```

- **Gym Environment Initialization**:
  - **Observation Space**: Defines the observation space with a shape of 25 and a range from `-1.0` to `1.0`.

```
self.observation_space = gym.spaces.Box(
    low=-1.0,
    high=1.0,
    shape=(25,),
    dtype=np.float32
)
```

  - **Action Space**: Defines the action space with a shape of 2, corresponding to the linear and angular velocity, and a range from `-1.0` to `1.0`.

```
self.action_space = gym.spaces.Box(
    low=-1.0,
    high=1.0,
    shape=(2,),
    dtype=np.float32
)
```

  - **Reward Range**: Sets a reward range from `-1.0` to `1.0`.

```
self.reward_range = (-1.0, 1.0)
```

- **Environment Parameters**: Sets parameters for velocity, yaw rate, and episode time limits:

```
self._min_linear_velocity = -5.0
self._max_linear_velocity = 5.0
self._max_yaw_rate = 5.0
```

(continues on next page)

(continued from previous page)

```
self._max_episode_time_seconds = 60.0
self._episode_start_time_seconds = None
```

Variables for tracking the target distance are also initialized:

```
self._current_target_distance = None
self._previous_target_distance = None
```

7.3.2 Methods

- **convert_action_to_request(self, action: np.ndarray = None)**: Converts the Gym action values into a ROS request format, scaling and mapping the action parameters to the ranges required by the ROS message fields.
  - **Action Scaling**: Converts action values from Gym to ROS format:
    - **Linear Velocity**: Scales *action[0]* from [-1.0, 1.0] to *[self._min_linear_velocity, self._max_linear_velocity]*.
    - **Yaw Rate**: Scales *action[1]* to *[0.0, self._max_yaw_rate]*.

```
def convert_action_to_request(self, action: np.ndarray = None):
    # action = np.array([linear_velocity, yaw_rate])

    # Scale the action values
    linear_velocity = (action[0] + 1.0) * (self._max_linear_velocity - self._
↪min_linear_velocity) / 2.0 + self._min_linear_velocity
    yaw_rate = action[1] * self._max_yaw_rate

    # Fill the step request
    self.step_request.action.tank.target_twist.y = linear_velocity
    self.step_request.action.tank.target_twist.theta = yaw_rate

    return self.step_request
```

- **convert_response_to_state(self, response)**: Converts the ROS response into a Gym-compatible state format by extracting the *state* from the ROS response.

```
def convert_response_to_state(self, response):
    return response.state
```

- **reset(self)**: Resets the environment to its initial state and clears previous values related to target distance and health.

```
self._episode_start_time_seconds = time.time()
self._previous_target_distance = None
```

And it is ensured that any additional reset procedures from the parent class are also executed.

```
return super().reset()
```

- **observation(self, state) -> np.ndarray**: Provides the current observation based on the environment's state.
  - **Target Relative Position**: Computes the relative position of the target in the global coordinate system by subtracting the tank's position from the target's position. This position is then adjusted for the tank's yaw using a rotation transformation.

```
target_relative_position = np.array([
    state.target_pose.x - state.tank.pose.x,
    state.target_pose.y - state.tank.pose.y,
    0.0
])

yaw = state.tank.pose.theta
r = Rotation.from_euler('z', yaw)
target_relative_position = r.apply(target_relative_position)
target_relative_position = target_relative_position[:2]
```

- **Normalizing the target's relative position** based on the distance to ensure it falls within a specific range. If the distance is less than 1.0, the position is used as is; otherwise, it is scaled to be within the range [0, 1].

```
self._current_target_distance = np.linalg.norm(target_relative_
↪position)
target_relative_position_normalized = target_relative_position␣
↪if self._current_target_distance < 1.0 else target_relative_
↪position / self._current_target_distance
```

- **Linear and Angular Velocities**: Normalized to fit within a specified range.

```
linear_velocity_normalized = (state.tank.twist.y - self._min_
↪linear_velocity) / (self._max_linear_velocity - self._min_
↪linear_velocity) * 2 - 1
angular_velocity_normalized = state.tank.twist.theta / self._max_
↪yaw_rate
```

- **Lidar Data**: Normalizes the lidar data to fit within the range [0, 1] based on the minimum and maximum range values.

```
ranges = np.array(state.tank.smart_laser_scan.ranges)
lidar_ranges_normalized = (ranges - state.tank.smart_laser_scan.
↪range_min) / (state.tank.smart_laser_scan.range_max - state.
↪tank.smart_laser_scan.range_min)
```

- **Health Information**: Normalized only for the agent's health.

```
self._current_health_normalized = state.tank.health_info.health /
↪ state.tank.health_info.max_health
```

Note

Combined Observation: Concatenates all these normalized values into a single observation array that represents the state of the environment.

```
observation = np.concatenate([
   target_relative_position_normalized,
   [linear_velocity_normalized],
   [angular_velocity_normalized],
   lidar_ranges_normalized,
   [self._current_health_normalized]
```

```
])

return observation
```

- **reward(self, state, action: np.ndarray = None) -> float. Computes the reward as a floating-point value for the agent based on the current state of the environment and actions taken. It is computed as follows:**
  - **Initial Reward Setup**. The reward is initialized to the tank's current normalized health:

    ```
    reward = self._current_health_normalized
    ```

    This value ensures that the agent's health is factored into the reward calculation, encouraging actions that maintain or improve the tank's health.
  - **Distance-Based Reward**. If a previous target distance has been recorded (i.e., the agent has taken at least one step), it is computed the difference between the previous distance and the current distance to the target. The difference is multiplied by a factor of *20.0* to increase the reward the agent gets for being closer to the target.

    ```
    if self._previous_target_distance is not None:
       reward += 20.0 * (self._previous_target_distance - self._current_target_
    ↪distance)
    ```
  - **Update Previous Distance**. After calculating the reward, the current distance is stored as the previous distance for use in the next time step:

    ```
    self._previous_target_distance = self._current_target_distance

    return reward
    ```

- **terminated(self, state) -> bool**: Determines whether the current episode has ended based on the state of the environment.
  - Checks if the current timer count of a trigger sensor is bigger than its maximum allowed value.

    ```
    has_reached_target = state.target_trigger_sensor.timer_count > state.
    ↪target_trigger_sensor.max_timer_count
    ```
  - Checks if the tank has died by evaluating if its health is less than or equal to 0.0.

    ```
    has_died = state.tank.health_info.health <= 0.0
    ```

  The episode is considered terminated if either the tank has died or it has reached the target.

  ```
  terminated = has_died or has_target_died

  return terminated
  ```

- **truncated(self, state) -> bool**: Checks if the episode has been truncated due to exceeding the maximum allowed time.

  To do so, the elapsed time since the episode started is calculated:

  ```
  episode_time_seconds = time.time() - self._episode_start_time_seconds
  ```

  And if the elapsed time exceeds the maximum limit, it is truncated:

```
truncated = episode_time_seconds > self._max_episode_time_seconds

return truncated
```

- **info(self, state) -> dict**: Provides additional information about the environment's state.

```
def info(self, state) -> dict:

    return {}
```

- **render(self, render_mode: str = 'human')**: Renders the current state of the environment based on the specified render mode.
  - **Render Mode Validation**. First it checks if the provided *render_mode* is valid. It supports two modes: *'human'* and *'rgb_array'*. If an invalid mode is specified, a *ValueError* is raised.

```
valid_render_modes = ['human', 'rgb_array']

if render_mode not in valid_render_modes:
    raise ValueError(f"Invalid render mode: {render_mode}. Valid render␣
↪modes are {valid_render_modes}")
```

  - **State Extraction and Image Decompression**. Extracts the current state from *self.step_response* and decompresses the image data from the state.

```
state = self.step_response.state

# Decompress the image
np_arr = np.frombuffer(state.compressed_image.data, np.uint8)
image = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
```

  - **Rendering Based on Mode:, which can be:**
    - *'human'*: Displays the image in a window using OpenCV.
    - *'rgb_array'*: Returns the image as a NumPy array.

```
if render_mode == 'human':
    cv2.imshow("ShootingExampleEnvironment", image)
    cv2.waitKey(1)

elif render_mode == 'rgb_array':
    return image
```

7.4 Training Script for UC1 Environment

Function *train_uc1()* is responsible for training reinforcement learning agents within the **UC1Environment** using the **Stable Baselines 3** library. To do so, it sets up the training environment, loads configurations, and manages the training process using the *RLTrainer* class from the *rl_pipeline* module.

1. **Firstly, we load the Configuration Files:**
   - *config.yml*: Holds general environment settings.

```
n_environments: 1
use_case: uc1

unity:
build_name: "uc1/Playground"  # assume they are on builds/<machine>/
↪<build_name>/<extension>, you dont need to set anything about the␣
↪machine, just by running a .bash or .bat is enough
headless_mode: false
pause: false
sample_time: 0.0
time_scale: 1.0
```

- *base_ppo_config.yaml*: Contains specific configurations for the Proximal Policy Optimization (PPO) algorithm. It can be changed in order to test various algorithms, architectures and hyperparameter values. Its id in the environment section must be specified, so we will name it *NavigationExample* ` as we are in Use Case 1:

```
id: 'NavigationExample'
env_config: 'None'
render_mode: 'rgb_array'
monitor: true
video_wrapper: false
video_trigger: 5000
video_length: 200
```

So it results in the following lines:

```
def train_uc1():
    # Load the configuration file
    config_file_path = "config.yml"
    train_config_path = 'rl_pipeline/configs/base_ppo_config.yaml'
```

The files are loaded using *yaml.safe_load()* to ensure safe reading of YAML content.

```
config = yaml.safe_load(open(config_file_path, 'r'))
train_config = yaml.safe_load(open(train_config_path, 'r'))
```

2. **Experiment Name and Logging Directory**. The experiment name is dynamically created based on the current date, and a logging directory is structured to include the environment ID and algorithm name:

```
exp_name = f"{train_config['experiment']['name']}_{str(datetime.date.today())}"
log_dir = (Path('experiments/') / train_config['environment']['id'] /
           train_config['training']['algorithm'] / exp_name)
```

3. **Creating the Environment**. A vectorized environment, which allows the training of multiple agents in parallel, is created using the *UC1Environment.create_vectorized_environment()* method, where the number of environments (*n_environments*) is determined from the configuration file:

```
n_environments = config["n_environments"]

vec_env = UC1Environment.create_vectorized_environment(
    n_environments=n_environments,
    return_type="stable-baselines",
```

(continues on next page)

(continued from previous page)

```
    monitor=train_config['environment']['monitor']
)
```

4. **Video Recording** (Optional). If video recording is enabled in the configuration, the *VecVideoRecorder* is used to wrap the environment and record videos at every *video_trigger* step:

```
if train_config['environment'].get('video_wrapper'):
    vec_env = VecVideoRecorder(
        vec_env,
        video_folder=f"{str(log_dir / 'videos')}",
        record_video_trigger=lambda x: x % train_config.get('environment').get(
↪'video_trigger') == 0,
        video_length=train_config.get('environment').get('video_length')
    )
```

5. **Resetting the Environment**. The *vec_env.reset()* call resets the vectorized environment to its initial state before training begins to ensure that all agents start from a clean state.

```
vec_env.reset()
```

6. **Pre-trained Model Handling** (Optional). The path to a pre-trained model is obtained from the training configuration file to facilitate file operations, as long as the path is not set to *'None'*.

```
pm_path = train_config['training']['pretrained_model']
pretrained_model = None if pm_path == 'None' else Path(pm_path)
```

7. **Trainer Initialization and Execution**: The *RLTrainer* class is instantiated using the given environment (*vec_env*), training configuration, logging directory, optional pre-trained model, experiment name, and group information for tracking experiments via WandB.

```
trainer = RLTrainer(env=vec_env, config=train_config['training'], log_
↪dir=log_dir, pretrained_model=pretrained_model,
                    exp_name=exp_nlame, wandb_group=train_config[
↪'environment']['id'])
```

   Once initialized, the *run* method is called to start the training, with no external evaluation environment (which allows the model to be tested independently without influencing the ongoing training) or logger provided.

```
trainer.run(eval_env=None, logger=None)
```

7.5 Test Script for UC1 Environment

This script tests *UC1Environment* called by function *test_uc1()* by using: #. **test_gym_environment**: Tests a single environment. #. **test_vectorized_environment**: Tests a vectorized environment with multiple instances.

7.5.1 test_uc1()

This function is the entry point for testing both a single as well as a vectorized environment.

```
def test_uc1():
    # Run tests for both environments
    test_gym_environment()
    test_vectorized_environment()
```

7.5.2 test_gym_environment()

This function tests a single instance of the *UC1Environment* by creating a gym environment, resetting it, taking actions, and rendering it continuously until the environment is terminated or truncated.

1. **Create the Environment**. The function begins by creating an instance of *UC1Environment* using the method *create_gym_environment*. In this case, *environment_id=0* as it is User Case 1.

   ```
   env = UC1Environment.create_gym_environment(environment_id=0)
   ```

2. A **Communication Monitor** is attached to the environment for debugging internal state information.

   ```
   communication_monitor = CommunicationMonitor(env)
   ```

3. **Reset the Environment**, bringing it to its initial state before testing.

   ```
   env.reset()
   ```

4. **Define Initial Action** by setting to an array of zeros corresponding to the linear and angular velocity: *[0.0, 0.0]*.

   ```
   action = np.array([0.0, 0.0])
   ```

5. **Main Loop**. An infinite loop (*while True*) is used to repeatedly take actions in the environment, observe the rewards, and render the environment until the episode is terminated or truncated.

   ```
   while True:
       observation, reward, terminated, truncated, info = env.step(action)
   ```

   In each iteration, the action is updated to *[1.0, 1.0]* to simulate a constant control signal, which directs the environment to take specific steps in both action dimensions. Alternatively, random actions can be generated using *np.random.uniform(-1.0, 1.0, 2)*.

   ```
   action = np.array([1.0, 1.0])
   # action = np.random.uniform(-1.0, 1.0, 2)
   ```

   After each action, the environment's state is rendered, allowing visual feedback of the simulation and if the environment reaches a terminal or truncated state, it is reset.

   ```
   env.render()

   if terminated or truncated:
       env.reset()
   ```

6. **Close the Environment** once the loop is manually stopped (e.g., by keyboard interrupt).

```
env.close()
```

7.5.3 test_vectorized_environment()

The *test_vectorized_environment* function is used to test a vectorized environment setup. This function initializes and interacts with multiple instances of the environment simultaneously. It loads configuration details from *config.yml* and uses a vectorized environment to execute actions across all instances.

1. **Loading the Configuration File config.yml** using the *yaml* library. This file holds general environment settings.

```
config_file_path = "config.yml"
config = yaml.safe_load(open(config_file_path))
```

2. **Creating the Vectorized Environment** using the *UC1Environment.create_vectorized_environment* method to obtain the specified number of environments running in parallel.

```
vec_env = UC1Environment.create_vectorized_environment(n_environments=n_
↪environments, return_type='gym')
```

3. **Resetting the Environment** to initialize all environments to their starting states to interact with them.

```
vec_env.reset()
```

4. **Defining Initial Actions** where each action (linear and angular velocity) is set to *[0.0, 0.0]*.

```
actions = [[0.0, 0.0] for _ in range(vec_env.num_envs)]
```

5. **Interacting with the Environment** in a continuous loop, by calling *vec_env.step(actions)* method to perform actions in all environments. This method returns observations, rewards, termination flags, truncation flags, and additional information for each environment. After each step, new random actions are generated for the next iteration.

```
while True:

    observations, rewards, terminateds, truncateds, infos = vec_env.
↪step(actions)
    actions = [np.random.uniform(-1, 1, size=2) for _ in range(vec_env.num_
↪envs)]
```

6. **Closing the Environment** after the loop (which runs indefinitely in this example) to free up resources. In practice, a condition would be needed to break out of the loop when testing is complete.

```
vec_env.close()
```

7.6 Example

With *launch_unity_simulation.bat* we start the simulation of multiple Unity instances with configurable parameters based on a YAML configuration file.

```
launch_unity_simulation.bat
```

At this point we have no connection between ROS and Unity, so the arrows are red:

With *launch_ros_tcp_endpoint.bash* we enable the communications between ROS and Unity. It reads configuration details from a YAML file and launches multiple instances of a TCP endpoint node, each on a different port. This is useful for running several parallel environments of a server.

```
bash launch_ros_tcp_endpoint.bash
```

So the arrows become blue in both directions to indicate the communication is established.

Now, each time we are passing a message, the corresponding arrow indicating its direction will appear in yellow and we will see the tank moving through the environment:

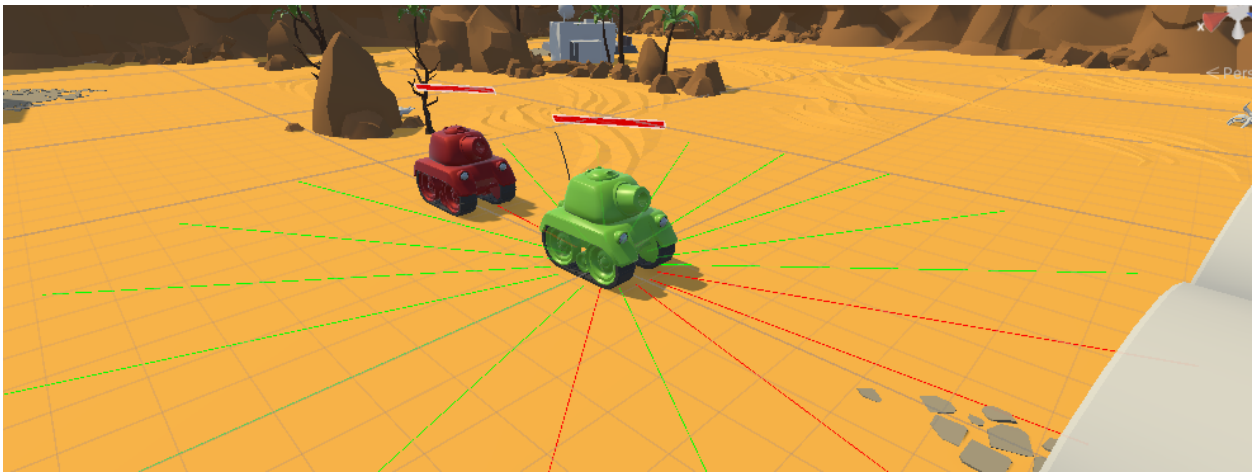
CHAPTER

EIGHT

USE CASE 2

8.1 Reach and shoot a static target

Use Case 2 integrates with the Gym library to create a simulation environment for reach and shoot a static target. The environment is designed to be used with reinforcement learning algorithms, allowing agents to learn to navigate and shoot a target autonomously.



Note

The rays in red appear when we are touching an object, which happends when an object is at a distance equal or smaller than 10 m to the tank, otherwise they are green.

8.2 Objectives

- Provide a customizable environment for reaching and shooting a target that does not move using ROS and Gym.
- Facilitate integration with reinforcement learning algorithms by defining observation, reward, and state management methods.
- Implement a training script that uses the Stable Baselines 3 library to train agents in the environment. It uses configuration files to try different algorithms, architectures, and hyperparameters.
- Develop a test script to validate the environment's functionality and behavior.

8.3 UC2Environment Class

The class *UC2Environment* is defined, that inherits from *EnvironmentNode* and configures a Gym environment with specific parameters defining the observation, action spaces,and the reward range. These configurations are crucial for defining the interaction between the agent and the environment, ensuring that both the agent's actions and the feedback it receives are appropriately scaled and represented.

8.3.1 Initialization

__init__(self, environment_id: int): This constructor initializes the environment by setting up ROS parameters and Gym environment parameters.

- **ROS Initialization**: Calls the parent class constructor (*EnvironmentNode.__init__*) to set up the ROS environment with specific message types and environment identifiers.

```
EnvironmentNode.__init__(
    self,
    environment_name="uc2_environment",
    environment_id=environment_id,
    step_service_msg_type=UC2EnvironmentStep,
    reset_service_msg_type=UC2EnvironmentReset,
)
```

- **Gym Environment Initialization**:
  - **Observation Space**: Defines the observation space with a shape of 27, as it adds to the observation space of Use Case 1 the turrent angle and the normalized health, and a range from `-1.0` to `1.0`.

```
self.observation_space = gym.spaces.Box(
    low=-1.0,
    high=1.0,
    shape=(27,),
    dtype=np.float32
)
```

  - **Action Space**: Defines the action space with a shape of 4, corresponding to the already defined linear and angular velocity in Use Case 1 the capacity to fire/not fire and the angle of the turret, and a range from `-1.0` to `1.0`.

```
self.action_space = gym.spaces.Box(
    low=-1.0,
    high=1.0,
    shape=(4,),
    dtype=np.float32
)
```

  - **Reward Range**: Establishes a reward range from `-1.0` to `1.0`.

```
self.reward_range = (-1.0, 1.0)
```

- **Environment Parameters**: Sets various parameters related to velocity, yaw rate, and episode time limits:

```
self._min_linear_velocity = -5.0
self._max_linear_velocity = 5.0
self._max_yaw_rate = 5.0
self._max_episode_time_seconds = 60.0
self._episode_start_time_seconds = None
```

Variables for tracking the target distance, already present in Use Case 1, and health of the units are also initialized:

```
self._current_target_distance = None
self._previous_target_distance = None
self._current_health_normalized = None
self._previous_health_normalized = None
self._current_target_health_normalized = None
self._previous_target_health_normalized = None
```

8.3.2 Methods

- **convert_action_to_request(self, action: np.ndarray = None)**: Converts the Gym values into a ROS request format, scaling and mapping the action parameters to the ranges and formats required by the appropriate ROS message fields.
  - **Action Scaling**: Converts the action values from the Gym environment to the ROS message format by scaling them according to the specified ranges:
    - **Linear Velocity**: Scales *action[0]* from the range [-1.0, 1.0] to the range *[self._min_linear_velocity, self._max_linear_velocity]*.
    - **Yaw Rate**: Scales *action[1]* to the range *[0.0, self._max_yaw_rate]*.
    - **Turret Target Angle**: Scales *action[2]* from the range [-1.0, 1.0] to the range *[0.0, 360.0]*.
    - **Fire**: Sets the *fire* flag to *True* if *action[3]* is greater than 0.5, otherwise *False*.
  - **Request Population**: Fills the ROS request message fields with the scaled action values.

```
def convert_action_to_request(self, action: np.ndarray = None):
    # action = np.array([linear_velocity, yaw_rate, turret_target_angle,
↪ fire])

    # Scale the action to the range [self._min_linear_velocity, self._
↪max_linear_velocity] when action[0] is in the range [-1.0, 1.0]
    linear_velocity = (action[0] + 1.0) * (self._max_linear_velocity -␣
↪self._min_linear_velocity) / 2.0 + self._min_linear_velocity
    yaw_rate = action[1] * self._max_yaw_rate

    # Scale the action to the range [0.0, 360.0] when action[2] is in␣
↪the range [-1.0, 1.0]
    turret_target_angle = (action[2] + 1.0) * 360.0 / 2.0
    fire = bool(action[3] > 0.5)

    # Fill the step request
    self.step_request.action.tank.target_twist.y = linear_velocity
    self.step_request.action.tank.target_twist.theta = yaw_rate
    self.step_request.action.tank.turret_actuator.target_angle = turret_
```

(continues on next page)

(continued from previous page)

```
↪target_angle
    self.step_request.action.tank.turret_actuator.fire = fire

    return self.step_request
```

In this User Case, we are adding when comparing to Use Case 1, the turret angle and the action of fire/not fire.

- **convert_response_to_state(self, response)**: Transforms the ROS response into a format suitable for Gym, returning the current state of the environment from the ROS response and ensuring that it is in a format that can be used within the Gym environment.

  It extracts the *state* attribute from the ROS response, which represents the current state of the environment. This ensures that the state information is formatted correctly for use in the Gym environment.

```
def convert_response_to_state(self, response):
    return response.state
```

- **reset(self)**: Resets the environment to its initial state, preparing it for a new episode by updating the episode start time and clearing previous values related to target distance and health. It also calls the `reset` method from the parent class (*EnvironmentNode*) to ensure any additional reset procedures defined in the parent class are also executed.
  - **Episode Start Time**: Updates the start time of the episode using the current time. This is used to track the elapsed time during the episode.

```
self._episode_start_time_seconds = time.time()
```

  - **Clear Previous Values**: Resets the previous values for target distance, already present in Use Case 1, and agent and target health to *None*. These values are used to compute rewards and determine the state of the environment.

```
self._previous_target_distance = None
self._previous_health_normalized = None
self._previous_target_health_normalized = None
```

  - **Call Parent Class Reset**: Calls the *reset* method from the parent class (*EnvironmentNode*) to ensure any additional reset procedures defined in the parent class are also executed.

```
return super().reset()
```

- **observation(self, state) -> np.ndarray**: Provides the current observation based on the environment's state.
  - **Target Relative Position**: Computes the relative position of the target in the global coordinate system by subtracting the tank's position from the target's position. This position is then adjusted for the tank's yaw using a rotation transformation.

```
target_relative_position = np.array([
    state.target_pose.x - state.tank.pose.x,
    state.target_pose.y - state.tank.pose.y,
    0.0
])

yaw = state.tank.pose.theta
```

(continues on next page)

(continued from previous page)

```
r = Rotation.from_euler('z', yaw)
target_relative_position = r.apply(target_relative_position)

target_relative_position = target_relative_position[:2]
```

- **Normalizing the target's relative position** based on the distance to ensure it falls within a specific range. If the distance is less than 1.0, the position is used as is; otherwise, it is scaled to be within the range [0, 1].

```
self._current_target_distance = np.linalg.norm(target_relative_position)
target_relative_position_normalized = target_relative_position if self._
↪current_target_distance < 1.0 else target_relative_position / self._
↪current_target_distance
```

- **Linear and Angular Velocities**: Normalized to fit within a specified range.

```
linear_velocity_normalized = (state.tank.twist.y - self._min_linear_
↪velocity) / (self._max_linear_velocity - self._min_linear_velocity) *␣
↪2 - 1
angular_velocity_normalized = state.tank.twist.theta / self._max_yaw_
↪rate
```

- **Lidar Data**: Normalizes the lidar data to fit within the range [0, 1] based on the minimum and maximum range values.

```
ranges = np.array(state.tank.smart_laser_scan.ranges)
lidar_ranges_normalized = (ranges - state.tank.smart_laser_scan.range_
↪min) / (state.tank.smart_laser_scan.range_max - state.tank.smart_
↪laser_scan.range_min)
```

- **Health Information**: Normalized both for the agent's and the target's health. In Use Case 1 it was normalized only for the agent's health.

```
self._current_health_normalized = state.tank.health_info.health / state.
↪tank.health_info.max_health
self._current_target_health_normalized = state.target_health_info.
↪health / state.target_health_info.max_health
```

- **Turret Information**: Normalizes the turret's angle, cooldown (time remaining before the turret can fire again), and firing status. This did not exist in Use Case 1 as we were only navigating in the environment and not reaching and shoooting a static target.

```
turret_angle_normalized = state.turret_sensor.current_angle / 360.0
turret_cooldown_normalized = state.turret_sensor.cooldown * state.
↪turret_sensor.fire_rate
turret_has_fired = 1.0 if state.turret_sensor.has_fired else 0.0
```

Note

Combined Observation: Concatenates all these normalized values into a single observation array that represents the state of the environment.

```
observation = np.concatenate([
    target_relative_position_normalized,
    [linear_velocity_normalized],
    [angular_velocity_normalized],
    lidar_ranges_normalized,
    [self._current_health_normalized],
    [self._current_target_health_normalized],
    [turret_angle_normalized],
    [turret_cooldown_normalized],
    [turret_has_fired]
])

return observation
```

- **reward(self, state, action: np.ndarray = None) -> float**: Computes the reward as a floating-point value for the agent based on the current state of the environment and actions taken. It is computed as follows:
  - **Health Change Reward** between the current normalized health of the agent and the previous one. If there is no previous health value, the reward is set to 0.0.

    ```
    health_change_reward = self._current_health_normalized - self._
    ↪previous_health_normalized if self._previous_health_normalized␣
    ↪is not None else 0.0
    ```

  - **Target Health Change Reward**: This reward is based on the change in the target's health. It is calculated as the Health Change Reward.
  - **Distance Change Reward**. This reward reflects the change in distance to the target, but only if the current distance is greater than 4.0. If the previous target distance is not available, or the current distance is too small, the reward is set to 0.0.

    ```
    distance_change_reward = self._previous_target_distance - self._
    ↪current_target_distance if self._previous_target_distance is␣
    ↪not None and self._current_target_distance > 4.0 else 0.0
    ```

  - **Has Shot Reward**, which is determined by the action value at index 3. If this value is greater than 0.5, the agent receives a reward of -0.1.

    ```
    has_shot_reward = -0.1 if action[3] > 0.5 else 0.0
    ```

> **Note**
>
> **Total Reward:**
> The total reward is the sum of all individual rewards calculated above. The previous values for target distance, agent health, and target health are updated for use in the next step of the episode.
>
> ```
> reward = 0.0
> reward += health_change_reward
> reward += target_health_change_reward
> reward += distance_change_reward
> reward += has_shot_reward
> ```
>
> Finally, the method returns the computed reward.

```
self._previous_target_distance = self._current_target_distance
self._previous_health_normalized = self._current_health_normalized
self._previous_target_health_normalized = self._current_target_health_
↪normalized

return reward
```

- **terminated(self, state) -> bool**: Determines whether the current episode has ended based on the state of the environment.
  - Checks if the tank has died by evaluating if its health is less than or equal to 0.0.

    ```
    has_died = state.tank.health_info.health <= 0.0
    ```

  - Checks if the target has died in the same way as before. This is new with respect to Use Case 1.

    ```
    has_target_died = state.target_health_info.health <= 0.0
    ```

  The episode is considered terminated if either the tank or the target has died.

  ```
  terminated = has_died or has_target_died

  return terminated
  ```

- **truncated(self, state) -> bool**: Determines whether the current episode has been truncated based on the elapsed time.
  - **Elapsed Time Calculation** by subtracting the start time from the current time.

    ```
    episode_time_seconds = time.time() - self._episode_start_time_
    ↪seconds
    ```

  - **Truncation Condition**. Checks if the elapsed time exceeds the maximum allowed episode time to determine if the episode should be truncated.

    ```
    truncated = episode_time_seconds > self._max_episode_time_seconds
    ```

  Returns *True* if the episode has been truncated due to exceeding the maximum time limit, otherwise *False*.

  ```
  return truncated
  ```

- **info(self, state) -> dict**: Provides additional information about the current state of the environment, typically returning an empty dictionary.

  ```
  def info(self, state) -> dict:
      return {}
  ```

- **render(self, render_mode: str = 'human')**: Renders the current state of the environment based on the specified render mode.
  - **Render Mode Validation**. First it checks if the provided *render_mode* is valid. It supports two modes: *'human'* and *'rgb_array'*. If an invalid mode is specified, a *ValueError* is raised.

```
valid_render_modes = ['human', 'rgb_array']

if render_mode not in valid_render_modes:
    raise ValueError(f"Invalid render mode: {render_mode}. Valid render␣
↪modes are {valid_render_modes}")
```

– **State Extraction and Image Decompression**. Extracts the current state from *self.step_response* and decompresses the image data from the state.

```
state = self.step_response.state

# Decompress the image
np_arr = np.frombuffer(state.compressed_image.data, np.uint8)
image = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
```

– **Rendering Based on Mode:, which can be:**

* *'human'*: Displays the image in a window using OpenCV.
* *'rgb_array'*: Returns the image as a NumPy array.

```
if render_mode == 'human':
    cv2.imshow("ShootingExampleEnvironment", image)
    cv2.waitKey(1)

elif render_mode == 'rgb_array':
    return image
```

8.4 Training Script for UC2 Environment

Function *train_uc2()* is responsible for training reinforcement learning agents within the **UC2Environment** using the **Stable Baselines 3** library. To do so, it sets up the training environment, loads configurations, and manages the training process using the *RLTrainer* class from the *rl_pipeline* module.

1. **Firstly, we load the Configuration Files:**

   - *config.yml*: Holds general environment settings. It must be changed with respect to Use Case 1.

```
n_environments: 1
use_case: uc2

unity:
build_name: "uc2/Playground"  # assume they are on builds/<machine>/
↪<build_name>/<extension>, you dont need to set anything about the␣
↪machine, just by running a .bash or .bat is enough
headless_mode: false
pause: false
sample_time: 0.0
time_scale: 1.0
```

   - **base_ppo_config.yaml**. Contains specific configurations for the Proximal Policy Optimization (PPO) algorithm. It can be changed in order to test various algorithms, architectures and hyperparameter values. It is common to Use Case 1 bwe should generate a new id in the environment section, in this case *ShootingExample*:

```
id: 'ShootingExample'
env_config: 'None'
render_mode: 'rgb_array'
monitor: true
video_wrapper: false
video_trigger: 5000
video_length: 200
```

So it results in the following lines:

```
def train_uc2():
    # Load the configuration file
    config_file_path = "config.yml"
    train_config_path = 'rl_pipeline/configs/base_ppo_config.yaml'
```

The files are loaded using *yaml.safe_load()* to ensure safe reading of YAML content.

```
config = yaml.safe_load(open(config_file_path, 'r'))
train_config = yaml.safe_load(open(train_config_path, 'r'))
```

2. **Experiment Name and Logging Directory**. The experiment name is dynamically created based on the current date, and a logging directory is structured to include the environment ID and algorithm name:

```
exp_name = f"{train_config['experiment']['name']}_{str(datetime.date.today())}"
log_dir = (Path('experiments/') / train_config['environment']['id'] /
           train_config['training']['algorithm'] / exp_name)
```

3. **Creating the Environment**. A vectorized environment, which allows the training of multiple agents in parallel, is created using the *UC2Environment.create_vectorized_environment()* method, where the number of environments (*n_environments*) is determined from the configuration file:

```
n_environments = config["n_environments"]

vec_env = UC2Environment.create_vectorized_environment(
    n_environments=n_environments,
    return_type="stable-baselines",
    monitor=train_config['environment']['monitor']
)
```

4. **Video Recording** (Optional). If video recording is enabled in the configuration, the *VecVideoRecorder* is used to wrap the environment and record videos at every *video_trigger* step:

```
if train_config['environment'].get('video_wrapper'):
    vec_env = VecVideoRecorder(
        vec_env,
        video_folder=f"{str(log_dir / 'videos')}",
        record_video_trigger=lambda x: x % train_config.get('environment').get(
↪'video_trigger') == 0,
        video_length=train_config.get('environment').get('video_length')
    )
```

5. **Resetting the Environment**. The *vec_env.reset()* call resets the vectorized environment to its initial state before training begins to ensure that all agents start from a clean state.

```
vec_env.reset()
```

6. **Pre-trained Model Handling** (Optional). The path to a pre-trained model is obtained from the training configuration file to facilitate file operations, as long as the path is not set to *'None'*.

```
pm_path = train_config['training']['pretrained_model']
pretrained_model = None if pm_path == 'None' else Path(pm_path)
```

7. **Trainer Initialization and Execution**: The *RLTrainer* class is instantiated using the given environment (*vec_env*), training configuration, logging directory, optional pre-trained model, experiment name, and group information for tracking experiments via WandB.

```
trainer = RLTrainer(env=vec_env, config=train_config['training'], log_
↪dir=log_dir, pretrained_model=pretrained_model,
                    exp_name=exp_name, wandb_group=train_config['environment
↪']['id'])
```

Once initialized, the *run* method is called to start the training, with no external evaluation environment (which allows the model to be tested independently without influencing the ongoing training) or logger provided.

```
trainer.run(eval_env=None, logger=None)
```

8.5 Test Script for UC2 Environment

This script tests *UC2Environment* called by function *test_uc2()* by using: #. **test_gym_environment**: Tests a single environment. #. **test_vectorized_environment**: Tests a vectorized environment with multiple instances.

8.5.1 test_uc2()

This function is the entry point for testing both a single as well as a vectorized environment.

```
def test_uc2():
    # Run tests for both environments
    test_gym_environment()
    test_vectorized_environment()
```

8.5.2 test_gym_environment()

This function tests a single instance of the *UC2Environment* by creating a gym environment, resetting it, taking actions, and rendering it continuously until the environment is terminated or truncated.

1. **Create the Environment**. The function begins by creating an instance of *UC2Environment* using the method *create_gym_environment*. In this case, *environment_id=1* as it is User Case 2.

```
env = UC2Environment.create_gym_environment(environment_id=1)
```

2. A **Communication Monitor** is attached to the environment for debugging internal state information.

```
communication_monitor = CommunicationMonitor(env)
```

3. **Reset the Environment**, bringing it to its initial state before testing.

```
env.reset()
```

4. **Define Initial Action** by setting to an array of zeros corresponding to the already defined linear and angular velocity in Use Case 1, the capacity to fire/not fire and the angle of the turret: *[0.0, 0.0, 0.0, 0.0]*.

```
action = np.array([0.0, 0.0, 0.0, 0.0])
```

5. **Main Loop**. An infinite loop (*while True*) is used to repeatedly take actions in the environment, observe the rewards, and render the environment until the episode is terminated or truncated.

```
while True:
    observation, reward, terminated, truncated, info = env.step(action)
```

In each iteration, the action is updated using random actions that can be generated using *np.random.uniform(-1.0, 1.0, 2)*. Alternatively, it could be set manually with predefined values: Alternatively,

```
action = np.random.uniform(-1.0, 1.0, 2)
# action = np.array([1.0, 1.0, 0.0, 0.0])
```

After each action, the environment's state is rendered, allowing visual feedback of the simulation and if the environment reaches a terminal or truncated state, it is reset.

```
env.render()

if terminated or truncated:
    env.reset()
```

6. **Close the Environment** once the loop is manually stopped (e.g., by keyboard interrupt).

```
env.close()
```

8.5.3 test_vectorized_environment()

The *test_vectorized_environment* function is used to test a vectorized environment setup. This function initializes and interacts with multiple instances of the environment simultaneously. It loads configuration details from *config.yml* and uses a vectorized environment to execute actions across all instances.

1. **Loading the Configuration File config.yml** using the *yaml* library. This file holds general environment settings.

```
config_file_path = "config.yml"
config = yaml.safe_load(open(config_file_path))
```

2. **Creating the Vectorized Environment** using the *UC2Environment.create_vectorized_environment* method to obtain the specified number of environments running in parallel.

```
vec_env = UC2Environment.create_vectorized_environment(n_environments=n_
↪environments, return_type='gym')
```

3. **Resetting the Environment** to initialize all environments to their starting states to interact with them.

```
vec_env.reset()
```

4. **Defining Initial Actions** where each action (linear and angular velocity, capacity to fire/not fire and the angle of the turret) is set to *[0.0, 0.0, 0.0, 0.0]*.

```
actions = [[0.0, 0.0, 0.0, 0.0] for _ in range(vec_env.num_envs)]
```

5. **Interacting with the Environment** in a continuous loop, by calling *vec_env.step(actions)* method to perform actions in all environments. This method returns observations, rewards, termination flags, truncation flags, and additional information for each environment. After each step, new random actions are generated for the next iteration.

```
while True:

    observations, rewards, terminateds, truncateds, infos = vec_env.
↪step(actions)
    actions = [np.random.uniform(-1, 1, size=3) for _ in range(vec_env.num_
↪envs)]
```

6. **Closing the Environment** after the loop (which runs indefinitely in this example) to free up resources. In practice, a condition would be needed to break out of the loop when testing is complete.

```
vec_env.close()
```

8.6 Example

With *launch_unity_simulation.bat* we start the simulation of multiple Unity instances with configurable parameters based on a YAML configuration file.

```
launch_unity_simulation.bat
```

At this point we have no connection between ROS and Unity, so the arrows are red:

With *launch_ros_tcp_endpoint.bash* we enable the communications between ROS and Unity. It reads configuration details from a YAML file and launches multiple instances of a TCP endpoint node, each on a different port. This is useful for running several parallel environments of a server.

```
bash launch_ros_tcp_endpoint.bash
```

So the arrows become blue in both directions to indicate the communication is established.

Now, each time we are passing a message, the corresponding arrow indicating its direction will appear in yellow and we will see the tank moving through the environment:

CHAPTER

NINE

USER CASE 3

9.1 Target followed by bots

An example of the bots in action can be found in the following video:

https://youtu.be/oeQbDXxDUyY

CHAPTER

TEN

CUSTOMIZING ENVIRONMENTS

CHAPTER

ELEVEN

CUSTOMIZING MODELS

CHAPTER

TWELVE

TRAINING DEFINITION

CHAPTER

THIRTEEN

SAVING AND LOADING MODELS

CHAPTER

FOURTEEN

WANDB LOGGING

CHAPTER

FIFTEEN

INDRA-RL-LAB (DOCKER)

15.1 Project Structure

```
indra-rl-lab/
 └── [Volume]/
 |   └── [rl_pipeline]/
 |   |    └── [blocks]/
 |   |    |   ├── resnet.py/
 |   |    └── [configs]/
 |   |    |   ├── config.yml/
 |   |    └── [feature_extractors]/
 |   |    |   ├── base_extractor/
 |   |    |   ├── resnet_extractor/
 |   |    └── algorithm_registry/
 |   |    └── callbacks/
 |   |    └── rl_pipeline/
 |   |    └── schedulers/
 |   └── [ros_ws/src]/
 |   |     └── [interfaces_pkg]/
 |   |         └── [rl_pkg/msg]/
 |   |         |    ├── HealthInfo.msg
 |   |         |    └── ...
 |   |         └── [use_cases]/
 |   |             ├── [agents]/
 |   |             |   ├── TankAction.msg/
 |   |             |   ├── ...
 |   |             └── [uc1]/
 |   |             |    ├── UC1AgentAction.msg/
 |   |             |    ├── UC1EnvironmentStep.msg/
 |   |             |    ├── ...
 |   |             ├── [uc2]/
 |   |             └── [uc3]/
 |   └── [use_cases]/
 |        ├── [uc1]/
 |        |    ├── deploy/
 |        |    ├── environment/
 |        |    ├── train/
 |        |    ├── test/
 |        |
 |        └── [uc2]/
```

(continues on next page)

(continued from previous page)

```
└── Dockerfile
└── requirements.txt
```

15.2 RL Pipeline

- **algorithm_registry/**:
- **callbacks/**:
- **rl_trainer/**:
- **schedulers/**:

15.2.1 Blocks

15.2.2 Configs

15.2.3 Feature extractors

15.3 Ros_ws/src

15.3.1 interfaces_pkg

- **rl_pkg/msg: Contains the message definitions for the ROS topics.**
  - **HealthInfo**: Contains the use cases for the project.
  - **…**
- **use_cases/: Contains the use cases for the project.**
  - **agents: Contains the use cases for the project.**
    - **TankAction**: Contains the use cases for the project.
    - **…**
  - **uc1**
    - **UC1AgentAction**: Contains the use cases for the project.
    - **UC1EnvironmentStep**: Contains the use cases for the project.
    - **…**
  - **uc2**
    - **…**
  - **uc3**
    - **…**

15.3.2 rl_pkg

- **launch:**
  - **deploy_launch**:
  - **train_launch**:
  - **test_launch**:
- **resource**:
- **rl_pkg:**
  - **utils**:
  - **visualizers**:
  - **wrappers**:

15.3.3 ros_tcp_endpoint

- **launch:**
  - **ros_tcp_endpoint_launch**:
- **resource**:
- **ros_tcp_endpoint:**
  - **client.py**
  - **service.py**
  - **...**

15.4 Use cases

- **uc1:**
  - **deploy**:
  - **environment**:
  - **train**:
  - **test**:
- **uc2:**
  - **...**:
- **config.yml**:

15.5 Unity

If wanting to change the Unity environment by creating new scenarios, the following files should be modified Indra-RL-Lab-Unity

CHAPTER

SIXTEEN

FREQUENTLY ASKED QUESTIONS

16.1 What is the objective of Use Case 1?

Use Case 1 integrates with the Gym library to create a basic simulation environment for tank navigation.

Its objectives are:

- Provide a fundamental environment for tank navigation using ROS and Gym.
- Enable integration with reinforcement learning algorithms by specifying observation, reward, and state management methods.
- Implement a training script that uses the Stable Baselines 3 library to train agents in the environment. It uses configuration files to try different algorithms, architectures, and hyperparameters.
- Develop a test script to validate the environment's functionality and behavior.

16.2 What is the objective of Use Case 2?

Use Case 2 integrates with the Gym library to create a simulation environment for reach and shoot a static target. The environment is designed to be used with reinforcement learning algorithms, allowing agents to learn to navigate and shoot a target autonomously.

Its objectives are:

- Provide a customizable environment for reaching and shooting a target that does not move using ROS and Gym.
- Facilitate integration with reinforcement learning algorithms by defining observation, reward, and state management methods.
- Implement a training script that uses the Stable Baselines 3 library to train agents in the environment. It uses configuration files to try different algorithms, architectures, and hyperparameters.
- Develop a test script to validate the environment's functionality and behavior.

16.3 How are actions converted in the environment?

The method `convert_action_to_request` scales the Gym action values for ROS. For example:

- **Linear Velocity**: Scales action[0] to fit between the defined min and max velocities.
- **Yaw Rate**: Scales action[1] to fit the yaw rate limits.
- **Turret Target Angle**: Scales *action[2]* from the range [-1.0, 1.0] to the range *[0.0, 360.0]*.
- **Fire**: Sets the *fire* flag to *True* if *action[3]* is greater than 0.5, otherwise *False*.

16.4 What triggers termination or truncation of an episode?

- **Termination:**
  - In Use Case 1, it occurs when the tank reaches the target or its health reaches zero.
  - In Use Case 2, when the tank dies or the target is destroyed.
- **Truncation**: Happens if the episode exceeds the time limit.

16.5 What should I change to train a new algorithm?

In *indra-rl-lab/volume*:

1. Modify the both the training and testing configuration files in the *configs/* directory.

```yaml
algorithm: 'ppo'
pretrained_model: 'None'
use_wandb: true
algorithm_parameters:
    policy: 'MlpPolicy'
    learning_rate: 0.0003
    n_steps: 128
    batch_size: 32
    n_epochs: 5
    gamma: 0.99
    gae_lambda: 0.95
    clip_range: 0.2
    ent_coef: 0.0
    vf_coef: 0.5
    max_grad_norm: 0.5
```

2. In *utils/algorithm_registry.py*:
   - Add a new function to get the model's parameters.
   - Import from stable-baselines3 the model architecture.
   - Add the new model architecture and its parameters to *AVAILABLE_ALGORITHMS* dictionary.

16.6 What should I change to use another feature extractor?

It must be said that the same feature extractor must be used in training and testing.

To add a new feature extractor:

1. In *rl_pipeline/models*:
    - In *feature_extractors* -> Add the new feature extractor class in a new python file. It should inherit from the base extractor class.
    - In *blocks* -> Add the architecture of the neural network that will use the new feature extractor.
2. In *rl_pipeline/configs*: Update the configuration files of training and testing to use the new feature extractor. This means changing the following in the architecture section:

```
architecture:
    net_arch: {'pi': [128, 128], 'vf': [128, 128]}
    features_extractor_class: 'ResnetMLP'
    features_extractor_kwargs:
            features_dim: 128
    activation_fn: 'ReLU'
    share_features_extractor: false
```